Windows Media VC-1 Advanced Profile Decoder on C64x+

User Guide

Literature Number: SPRUEI3A
March 2007

IMPORTANT NOTICE

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
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
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
Low Power Wireless	www.ti.com/lpw	Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

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

Preface

Read This First

About This Manual

This document describes how to install and work with Texas Instruments' (TI) VC-1 Advanced Profile Decoder implementation on the C64x+ platform. It also provides a detailed Application Programming Interface (API) reference and information on the sample application that accompanies this component.

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

Intended Audience

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

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

How to Use This Manual

This document includes the following chapters:

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

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.
- *TMS320DM6446 Digital Media System-on-Chip* (literature number SPRS283)
- *TMS320DM6446 Digital Media System-on-Chip Errata (Silicon Revision 1.0)* (literature number SPRZ241) describes the known exceptions to the functional specifications for the TMS320DM6446 Digital Media System-on-Chip (DMSoC).
- *TMS320DM6443 Digital Media System-on-Chip* (literature number SPRS282)
- *TMS320DM6443 Digital Media System-on-Chip Errata (Silicon Revision 1.0)* (literature number SPRZ240) describes the known exceptions to the functional specifications for the TMS320DM6443 Digital Media System-on-Chip (DMSoC).
- *TMS320DM644x DMSoC DSP Subsystem Reference Guide* (literature number SPRUE15) describes the digital signal processor (DSP) subsystem in the TMS320DM644x Digital Media System-on-Chip (DMSoC).
- *TMS320DM644x DMSoC ARM Subsystem Reference Guide* (literature number SPRUE14) describes the ARM subsystem in the TMS320DM644x Digital Media System on a Chip (DMSoC).
- *DaVinci Technology - Digital Video Innovation Product Bulletin (Rev. A)* (sprt378a.pdf)
- *The DaVinci Effect: Achieving Digital Video Without Complexity White Paper* (spry079.pdf)
- *DaVinci Benchmarks Product Bulletin* (sprt379.pdf)
- *DaVinci Technology for Digital Video White Paper* (spry067.pdf)
- *The Future of Digital Video White Paper* (spry066.pdf)

Related Documentation

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

- *Windows Media Video V9 Decoding Specification* (Revision 87)
- *Proposed SMPTE Standard for Television: VC-1 Compressed Video Bitstream Format and Decoding Process* (Final committee Draft 1, revision 5)

Abbreviations

The following abbreviations are used in this document:

Table 1-1. List of Abbreviations

Abbreviation	Description
AP	Advanced Profile
API	Application Programming Interface
ASF	Advanced Systems Format
DMA	Direct Memory Access
DMAN3	DMA Manager
DSP	Digital Signal Processing
EVM	Evaluation Module
MP	Main Profile
MPML	Main Profile at Main Level
RCV	Raw Compressed Video
SP	Simple Profile
SMPTE	Society of Motion Picture and Television Engineers
WMV	Windows Media Video
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, please quote the product name (VC-1 Advanced Profile Decoder on C64x+) and version number. The version number of the codec is included in the Title of the Release Notes that accompanies this codec.

Trademarks

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

All trademarks are the property of their respective owners.

This page is intentionally left blank

Contents

Figures

This page is intentionally left blank

Tables

This page is intentionally left blank

Chapter 1

Introduction

This chapter provides a brief introduction to XDAIS and XDM. It also provides an overview of TI's implementation of the VC-1 Advanced Profile Decoder on the C64x+ platform and its supported features.

1.1 Overview of XDAIS and XDM

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

1.1.1 XDAIS Overview

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

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

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

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

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

1.1.2 XDM Overview

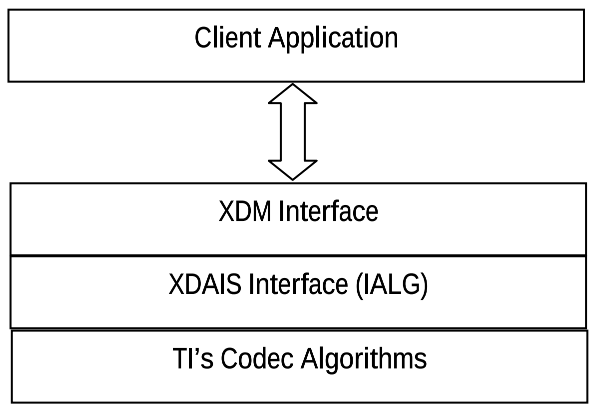
In the multimedia application space, you have the choice of integrating any codec into your multimedia system. For example, if you are building a video decoder system, you can use any of the available video decoders (such as MPEG4, H.263, or H.264) in your system. To enable easy integration with the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

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

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

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

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



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

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

1.2 Overview of VC-1 Advanced Profile Decoder

VC-1 is the Society of Motion Picture and Television Engineers (SMPTE) standardized video decoder. VC-1 consists of three profiles namely, simple, main, and advanced. Simple and main profiles were originally developed for use in lower-bit-rate networked computing environments. The advanced profile adds extensive in-band metadata support to allow for optimized experience on a wide range of display devices. This decoder supports Simple, Main, and Advance profile up to PAL D1 (720x576) resolution.

From this point onwards, all references to VC-1 Decoder means VC-1 Advanced Profile Decoder only.

1.3 Supported Services and Features

This user guide accompanies TI's implementation of VC-1 Decoder on the C64x+ platform.

This version of the codec has the following supported features:

- Supports the advanced profile features of the SMPTE FCD1r6 at level 1 (AP@L1) standard
- Supports elementary input streams
- Supports YUV 420 and YUV 422 interleaved output formats
- Considers Advanced Systems Format (ASF) parser as an application layer
- Main profile is bit exact with the reference decoder provided by Microsoft®
- Advanced profile (VC-1) is bit exact with the SMPTE test cases
- eXpressDSP compliant
- eXpressDSP Digital Media (XDM) compliant

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 DM644x EVM hardware with XDS510 USB (Code Composer Studio version 3.2.39.5 with patch 3.2.30.0).

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 3.2.39.5.
- **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 directory called 100_V_VC1_D_1_01.

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

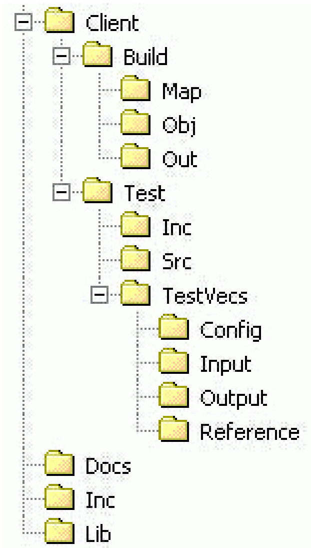


Figure 2-1. Component Directory Structure

> **Note:**
>
> If you are installing an evaluation version of this codec, the directory name will be 100E_V_VC1_D_1_01.

Table 2-1 provides a description of the sub-directories created in the 100_V_VC1_D_1_01 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, datasheet, and release notes
\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

Sub-Directory	Description
\Client\Build\Out	Contains the final application executable (.out) file generated by the sample test application
\Client\Test\Src	Contains application C files
\Client\Test\Inc	Contains header files needed for the application code
\Client\Test\TestVecs\Input	Contains input test vectors
\Client\Test\TestVecs\Output	Contains output generated by the codec
\Client\Test\TestVecs\Reference	Contains read-only reference output to be used for cross-checking against 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 5.21 and Framework Component (FC) version 1.02.

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_v3.2

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

- Header file, bcache.h available in the <install directory>\CCStudio_v3.2\<bios_directory>\packages\ti\bios\include directory.
- Library file, biosDM420.a64P available in the <install directory>\CCStudio_v3.2\<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_v3.2

The test application uses the following DMAN3 files:

- Library file, dman3.a64P available in the <install directory>\CCStudio_v3.2\<fc_directory>\packages\ti\sdo\fc\dman3 directory.
- Header file, dman3.h available in the <install directory>\CCStudio_v3.2\<fc_directory>\packages\ti\sdo\fc\dman3 directory.
- Header file, idma3.h available in the <install directory>\CCStudio_v3.2\<fc_directory>\packages\ti\sdo\fc\acpy3 directory.

2.4 Building and Running the Sample Test Application

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

1) Verify that you have an installation of TI's Code Composer Studio version 3.2.39.5 and code generation tools version 6.0.7.

2) Verify that the codec object library vc1dec_ti.l64P exists in the \Lib sub-directory.

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

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

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

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

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

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

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

- o “Decoder output dump completed” (for output dump mode)

2.5 Configuration Files

This codec is shipped along with:

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

2.5.1 Generic Configuration File

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

The format of the Testvecs.cfg file is:

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

where:

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

A sample Testvecs.cfg file is as shown:

```
1
..\..\Test\TestVecs\Config\Testparams.cfg
..\..\Test\TestVecs\Input\SA00050.vc1
..\..\Test\TestVecs\Output\SA00050_REF_OUT.yuv
0
..\..\Test\TestVecs\Config\Testparams.cfg
..\..\Test\TestVecs\Input\SA00050.vc1
..\..\Test\TestVecs\Output\SA00050_OUT.yuv
```

2.5.2 Decoder Configuration File

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

A sample Testparams.cfg file is as shown:

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

ImageWidth = 720     # Image width in Pels, must be
                      multiples of 16
ImageHeight = 576    # Image height in Pels, must be
                      multiples of 16
ChromaFormat = 1     # 1 => XDM_YUV_420P,
                       4 => XDM_YUV_422ILE
FramesToDecode = 110 # Number of frames to be coded
```

Note:

- For the RCV streams an additional Testparams file, Testparams_RCVStreams.cfg is included, which has the parameter `IsElementaryStream` determining VC1/RCV streams in addition to the parameters already specified in the Testparams file.

Any field in the `IVIDDEC_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:

- Copy the input files to the \Client\Test\TestVecs\Inputs sub-directory.
- Copy the reference files to the \Client\Test\TestVecs\Reference sub-directory.
- 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.
- Execute the sample test application. On successful completion, the application displays one of the following messages for each frame:
  - “Decoder compliance test passed/failed” (if `X` is 1)
  - “Decoder output dump completed” (if `X` is 0)

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

2.7 Uninstalling the Component

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

2.8 Evaluation Version

If you are using an evaluation version of this codec there will be a 10-minute limit on usage. A Texas Instruments logo will be visible in the output.

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 `IVIDDEC` base class of the VC-1 Decoder library. The main test application files are TestAppDecoder.c and TestAppDecoder.h. These files are available in the \Client\Test\Src and \Client\Test\Inc sub-directories respectively.

Figure 3-1 depicts the sequence of APIs exercised in the sample test application.

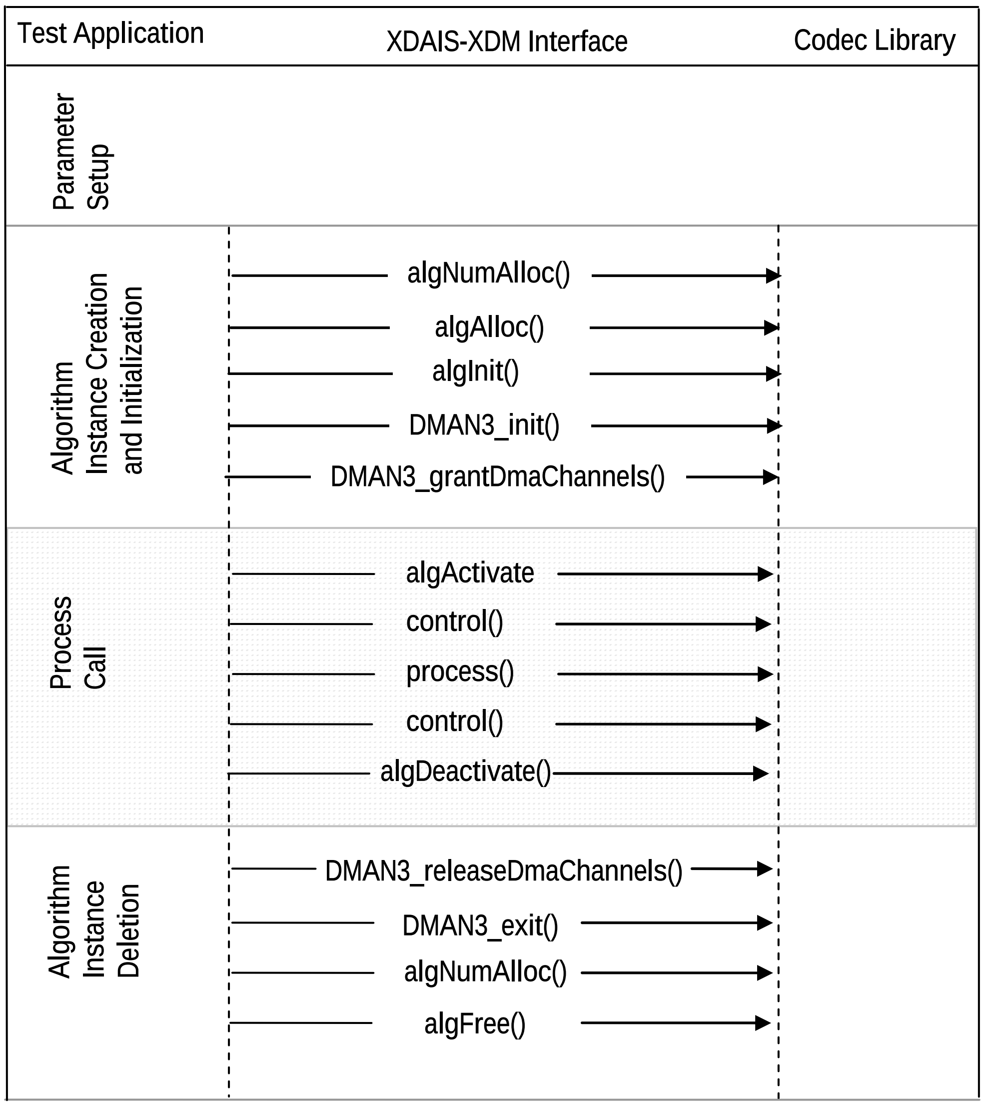


Figure 3-1. Test Application Sample Implementation

The test application is divided into four logical blocks:

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

3.1.1 Parameter Setup

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

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

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

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

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

3) Sets the `IVIDDEC_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 `IVIDDEC_InArgs` and `IVIDDEC_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 etc., using the six available control commands.
3) `process()` - To call the Decoder with appropriate input/output buffer and arguments information.
4) `control()` (optional) - To query the algorithm on status or setting of dynamic parameters etc., 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 as well. The test application does a cache invalidate for the valid input buffers before `process()` and a cache write back invalidate for output buffers after `process()`.

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

3.1.4 Algorithm Instance Deletion

Once decoding/encoding is complete, the test application 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.

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. Described alongside the macro or enumeration is the semantics or interpretation of the same in terms of what value it stands for and what it means.

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
	IVIDEO_INTERLACED	Interlaced video content
IVIDEO_FrameSkip	IVIDEO_NO_SKIP	Do not skip the current frame. Not supported in this version of VC-1 Decoder
	IVIDEO_SKIP_P	Skip forward inter coded frame. Not supported in this version of VC-1 Decoder
	IVIDEO_SKIP_B	Skip bi-directional inter coded frame. Not supported in this version of VC-1 Decoder
	IVIDEO_SKIP_I	Skip intra coded frame. Not supported in this version of VC-1 Decoder
XDM_DataFormat	XDM_BYTE	Big endian stream.
	XDM_LE_16	16-bit little endian stream. Not supported in this version of VC-1 Decoder
	XDM_LE_32	32-bit little endian stream. Not supported in this version of VC-1 Decoder
XDM_ChromaFormat	XDM_YUV_420P	YUV 4:2:0 planar
	XDM_YUV_422P	YUV 4:2:2 planar. Not supported in this version of VC-1 Decoder

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	XDM_YUV_422IBE	YUV 4:2:2 interleaved (big endian). Not supported in this version of VC-1 Decoder.
	XDM_YUV_422ILE	YUV 4:2:2 interleaved (little endian)
	XDM_YUV_444P	YUV 4:4:4 planar. Not supported in this version of VC-1 Decoder.
	XDM_YUV_411P	YUV 4:1:1 planar. Not supported in this version of VC-1 Decoder.
	XDM_GRAY	Gray format. Not supported in this version of VC-1 Decoder.
	XDM_RGB	RGB color format. Not supported in this version of VC-1 Decoder.
XDM_CmdId	XDM_GETSTATUS	Query algorithm instance to fill `Status` structure
	XDM_SETPARAMS	Set run-time dynamic parameters via the `DynamicParams` structure
	XDM_RESET	Reset the algorithm. All fields in the internal data structures are reset and all internal buffers are flushed. After you have reset the decoder, the decoder will start decoding from the point where it detects a sequence header in the bit stream. Any external buffers passed by the system to the decoder are assumed to be freed by the decoder after a reset.
	XDM_SETDEFAULT	Initialize all fields in `Params` structure to default values specified in the library
	XDM_FLUSH	Flushes the last output frame data for display. `control()` API should be called with the `XDM_FLUSH` command only after the decode of the last frame in the sequence.
	XDM_GETBUFINFO	Query algorithm instance regarding the properties of input and output buffers
XDM_DecMode	XDM_DECODE_AU	Decode entire frame including all the headers

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	XDM_PARSE_HEADER	Decode only the sequence/entry point frame header. Not supported in this version of VC-1 Decoder.
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 (see Table 4-2)

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

The VC-1 Decoder specific error status messages are listed in Table 4-2. The Value column indicates the decimal value of the last 8-bits reserved for codec specific error statuses.

Table 4-2. VC-1 Decoder Error Status

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
WMVDEC_STATUS	WMVDEC_invalidBitstreamAddress	1	Bit stream points to NULL
	WMVDEC_nullBitstreamAddress	2	The input pointer is set to NULL
	WMVDEC_insufficientData	3	Input bitstream incomplete to decode a frame
	WMVDEC_corruptedHeader	4	Corrupted header to the decoder
	WMVDEC_corruptedBitStream	5	Corrupted bit stream to the decoder
	WMVDEC_duringInitialization	6	Sequence header not found or insufficient data in the stream
	WMVDEC_invalidCodecVersion	7	Unsupported WMV version
	WMVDEC_allocFailure	8	Memory allocation error
	WMVDEC_systemError	9	Memory Related error
	WMVDEC_skippedFrame_USF	12	Indicates that skipped frame is not supported
	WMVDEC_unsupported_BS	13	Unsupported codec version of WMV
	WMVDEC_internalErrorBBOR	14	Error during Bounding box computations
	WMVDEC_internalErrorMBDec1	15	Failed to decode MB overhead in frame
	WMVDEC_internalErrorMBDec2	16	Failed in MB decode
	WMVDEC_nullWmvDecObject	17	Instance points to NULL
	WMVDEC_insufficientMemoryDuringInit	19	Insufficient memory to handle decoding process
	WMVDEC_internalErrorHWFE1	21	Error at Huffman decoding
	WMVDEC_internalErrorHWFE2	22	Error at Huffman decoding
	WMVDEC_unsupportedFeature	24	Indicates unsupported feature
	WMVDEC_internalErrorWQ	27	Wrong QP
	WMVDEC_unsupportedFeatureInterlace	28	Indicates Interlace decoding of main profile stream not supported
	WMVDEC_unsupportedFeatureSprite	29	Indicates SpriteMode not supported

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
	WMVDEC_unsupportedFeatureX8IIntra	30	Unsupported header bits for transform type
	WMVDEC_unsupportedFeatureMultiRes	31	Multi resolution video not supported
	WMVDEC_unsupportedFeatureBeta	33	Indicates non RTM compatible bit-stream is not supported
	WMVDEC_unsupportedFeatureComplex	34	Indicate WMV3_PC_PROFILE not supported
	WMVDEC_startCodeNotFound	35	Start code not found
	WMVDEC_invalidStartcode	36	Inappropriate start code
	WMVDEC_flushBitNotFound	37	Flush bit not found

Note:

The missing constants are not applicable for this version of VC-1 Decoder.

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`
- `IVIDDEC_Fxns`
- `IVIDDEC_Params`
- `IVIDDEC_DynamicParams`
- `IVIDDEC_InArgs`
- `IVIDDEC_Status`
- `IVIDDEC_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 VC-1 Decoder, the buffer details are:

- Number of input buffer required is 1
- Number of output buffer required is 1 for YUV 422ILE and 3 for YUV420P

There is no restriction on input buffer size except that it should contain at least one frame of encoded data.

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 IVIDDEC_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 IVIDDEC_Params

‖ Description

This structure defines the creation parameters for an algorithm instance object. Set this data structure to NULL, if you are unsure of the values to be specified for these parameters.

‖ Fields

Field	Datatype	Input/ Output	Description
size	XDAS_Int32	Input	Size of the basic or extended (if being used) data structure in bytes.
maxHeight	XDAS_Int32	Input	Maximum video height to be supported in pixels
maxWidth	XDAS_Int32	Input	Maximum video width to be supported in pixels
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. For example, if bit rate is 10 Mbps, set this field to 10485760.
dataEndianness	XDAS_Int32	Input	Endianness of input data. See XDM_DataFormat enumeration for details.
forceChromaFormat	XDAS_Int32	Input	Sets the output to the specified format. For example, if the output should be in YUV 4:2:2 interleaved (little endian) format, set this field to XDM_YUV_422ILE. See XDM_ChromaFormat enumeration for details.

Note:

- VC-1 Decoder does not use the maxFrameRate, maxBitRate, and dataEndianess fields for creating the algorithm instance.
- Only XDM_BYTE is supported for DataEndianness.
- Maximum video height and width supported are 576 pixels and 720 pixels respectively.

4.2.1.6 IVIDDEC_DynamicParams

|| Description

This structure defines the run time parameters for an algorithm instance object. Set this data structure to NULL, if you are unsure of the values to be specified for these parameters.

|| Fields

Field	Datatype	Input/ Output	Description
size	XDAS_Int32	Input	Size of the basic or extended (if being used) data structure in bytes.
decodeHeader	XDAS_Int32	Input	Number of access units to decode: ❑ 0 (XDM_DECODE_AU) - Decode entire frame including all the headers ❑ 1 (XDM_PARSE_HEADER) - Decode only one NAL unit
displayWidth	XDAS_Int32	Input	If the field is set to: ❑ 0 - Uses decoded image width as pitch ❑ If any other value greater than the decoded image width is given, then this value in pixels is used as pitch.
frameSkipMode	XDAS_Int32	Input	Frame skip mode. See IVIDEO_FrameSkip enumeration for details.

> **Note:**
>
> The fields frameSkipMode is not supported in this version of VC-1 Decoder.
>
> XDM_PARSE_HEADER is not supported in this version of VC-1 Decoder.

4.2.1.7 IVIDDEC_InArgs

‖ Description

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

‖ Fields

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

4.2.1.8 IVIDDEC_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.
outputHeight	XDAS_Int32	Output	Output height in pixels
outputWidth	XDAS_Int32	Output	Output width in pixels
frameRate	XDAS_Int32	Output	Average frame rate in fps * 1000. The average frame rate for all video decoders is 30 fps.
bitRate	XDAS_Int32	Output	Average bit rate in bits per second
contentType	XDAS_Int32	Output	Video content. See IVIDEO_ContentType enumeration for details.
outputChromaFormat	XDAS_Int32	Output	Output chroma format. See XDM_ChromaFormat enumeration for details.
bufInfo	XDM_AlgBufInfo	Output	Input and output buffer information. See XDM_AlgBufInfo data structure for details.

Note:

- The decoder, for few erroneous test streams under some rare circumstances may not set the extendedError parameter.
- For VC-1 Decoder, the bitRate is expressed in Kilo bits per second.

4.2.1.9 IVIDDEC_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.
bytesConsumed	XDAS_Int32	Output	Bytes consumed per decode call
decodedFrameType	XDAS_Int32	Output	Decoded frame type. See IVIDEO_FrameType enumeration for more details.
outputID	XDAS_Int32	Output	Output ID. See inputID field description in IVIDDEC_InArgs data structure for details.
displayBufs	IVIDEO_Buf Desc	Output	Decoder fills this structure to denote the buffer pointers for current frames. In case of sequences having I and P frames only, these values are identical to the output buffers (outBufs) passed using the process call.

Note:

The field decodedFrametype gives the frame type of the current frame decoded and it is -1 when RCV header is decoded. A value of -1 indicates the decoded frame type is Not Applicable (NA) for the frame decoded. This may appear sometimes for the erroneous bit streams.

4.2.2 VC-1 Decoder Data Structures

This section includes the following VC-1 Decoder specific extended data structures:

- IVC1DEC_Params
- IVC1DEC_DynamicParams
- IVC1DEC_InArgs
- IVC1DEC_Status
- IVC1DEC_OutArgs

4.2.2.1 IVC1DEC_Params

|| Description

This structure defines the creation parameters and any other implementation specific parameters for a VC-1 Decoder instance object. The creation parameters are defined in the XDM data structure, `IVIDDEC_Params`.

|| Fields

Field	Datatype	Input/ Output	Description
`viddecParams`	`IVIDDEC_Params`	Input	See `IVIDDEC_Params` data structure for details.

4.2.2.2 IVC1DEC_DynamicParams

|| Description

This structure defines the run time parameters and any other implementation specific parameters for a VC-1 Decoder instance object. The run time parameters are defined in the XDM data structure, `IVIDDEC_DynamicParams`.

|| Fields

Field	Datatype	Input/ Output	Description
`viddecDynamicParams`	`IVIDDEC_DynamicParams`	Input	See `IVIDDEC_DynamicParams` data structure for details.
`bIsElementaryStream`	`XDAS_UInt8`	Input	Flag indicating the type of input stream: ❑ 1 - Indicates that the stream is an elementary stream ❑ 2 - Indicates that the stream is an RCV stream

> **Note:**
>
> The Test Wrapper supports both VC1 and RCV streams and only `bIsElementaryStream` variable needs to be set properly.

4.2.2.3 IVC1DEC_InArgs

|| Description

This structure defines the run time input arguments for a VC-1 Decoder instance object.

‖ **Fields**

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

4.2.2.4 IVC1DEC_Status

‖ **Description**

This structure defines parameters that describe the status of the VC-1 Decoder and any other implementation specific parameters. The status parameters are defined in the XDM data structure, IVIDDEC_Status.

‖ **Fields**

Field	Datatype	Input/ Output	Description
viddecStatus	IVIDDEC_Status	Input	See IVIDDEC_Status data structure for details.

4.2.2.5 IVC1DEC_OutArgs

‖ **Description**

This structure defines the run time output arguments for the VC-1 Decoder instance object.

‖ **Fields**

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

4.3 Interface Functions

This section describes the Application Programming Interfaces (APIs) used in the VC-1 Decoder. The APIs are logically grouped into the following categories:

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

You must call these APIs in the following sequence:

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

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

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

4.3.1 Creation APIs

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

‖ Name

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

‖ Synopsis

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

‖ Arguments

```
Void
```

‖ Return Value

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

‖ Description

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

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

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

‖ See Also

`algAlloc()`

‖ Name

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

‖ Synopsis

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

‖ Arguments

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

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

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

‖ Return Value

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

‖ Description

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

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

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

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

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

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

‖ See Also

`algNumAlloc(), algFree()`

4.3.2 Initialization API

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

‖ Name

`algInit()` – initialize an algorithm instance

‖ Synopsis

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

‖ Arguments

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

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

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

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

‖ Return Value

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

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

‖ Description

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

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

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

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

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

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

‖ See Also

`algAlloc()`, `algMoved()`

4.3.3 Control API

Control API is used for controlling the functioning of the algorithm instance during run-time. This is done by changing the status of the controllable parameters of the 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) (IVIDDEC_Handle handle, IVIDDEC_Cmd
id, IVIDDEC_DynamicParams *params, IVIDDEC_Status
*status);
```

‖ Arguments

```
IVIDDEC_Handle handle; /* algorithm instance handle */

IVIDDEC_Cmd id; /* algorithm specific control commands*/

IVIDDEC_DynamicParams *params /* algorithm run time
parameters */

IVIDDEC_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 IVIDDEC_DynamicParams and IVIDDEC_Status data structures respectively.

Note:

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

‖ Preconditions

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

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

‖ Postconditions

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

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

‖ Example

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

‖ See Also

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

> **Note:**
>
> For VC1/RCV streams, the last frame can be obtained only by calling the Control API with Flush command and then calling the process API for getting the output in output buffers.

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)(IVIDDEC_Handle handle, XDM_BufDesc
*inBufs, XDM_BufDesc *outBufs, IVIDDEC_InArgs *inargs,
IVIDDEC_OutArgs *outargs);
```

‖ Arguments

```
IVIDDEC_Handle handle; /* algorithm instance handle */

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

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

IVIDDEC_InArgs *inargs /* algorithm runtime input
arguments */

IVIDDEC_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 IVIDDEC_InArgs data structure that defines the run time input arguments for an algorithm instance object.

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

Note:

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

‖ Preconditions

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

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

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

‖ Postconditions

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

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

‖ Example

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

‖ See Also

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

Note:

- A video encoder or decoder cannot be preempted 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.
- Any codec specific notes for process function.

‖ 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()`