WMA Version9 Decoder on C64x+

User's Guide

Literature Number: SPRUFN2
June 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
Copyright 2008, Texas Instruments Incorporated

Preface

Read This First

About This Manual

This document describes how to install and work with Texas Instruments' (TI) WMA Version9 Decoder implementation on the C64x+ based SoCs. 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+ based SoCs .

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

How to Use This Manual

This document includes the following chapters:

- **Chapter 1 - Introduction**, provides a brief introduction to the XDAIS and XDM standards. It also provides an overview of the codec and lists its supported features.
- **Chapter 2 - Installation Overview**, describes how to install, build, and run the codec.
- **Chapter 3 - Sample Usage**, describes the sample usage of the codec.
- **Chapter 4 - API Reference**, describes the data structures and interface functions used in the codec.
- **Appendix A - Data Fields for WMA RCA Header**, describes the data fields for WMA RCA file header section and WMA RCA Packet/Payload header section.

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).
- *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)
- *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)

Abbreviations

The following abbreviations are used in this document.

Table 1-1. List of Abbreviations

Abbreviation	Description
API	Application Programming Interface
ASF	Advanced Systems Format
DRM	Digital Rights Management
EVM	Evaluation Module
PCM	Pulse Coded Modulation
RCA	Raw Compressed Audio
WMA	Windows Media Audio
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 (WMA Version9 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, DM648, TNETV2685 and TMS320C64x+ are trademarks of Texas Instruments.

All trademarks are the property of their respective owners.

Contents

Read This First iii
About This Manual iii
Intended Audience iii
How to Use This Manual iii
Related Documentation From Texas Instruments iv
Abbreviations vi
Text Conventions vi
Product Support vi
Trademarks vi
Contents vii
Figures ix
Tables xi
Introduction 1-1
1.1 Overview of XDAIS and XDM 1-2
1.1.1 XDAIS Overview 1-2
1.1.2 XDM Overview 1-2
1.2 Overview of WMA Version9 Decoder 1-4
1.3 Supported Services and Features 1-4
Installation Overview 2-1
2.1 System Requirements 2-2
2.1.1 Hardware 2-2
2.1.2 Software 2-2
2.2 Installing the Component 2-2
2.3 Before Building the Sample Test Application 2-4
2.3.1 Installing DSP/BIOS 2-4
2.4 Building and Running the Sample Test Application 2-4
2.5 Configuration Files 2-5
2.5.1 Generic Configuration File 2-5
2.6 Standards Conformance and User-Defined Inputs 2-6
2.7 Uninstalling the Component 2-6
2.8 Evaluation Version 2-6
Sample Usage 3-1
3.1 Overview of the Test Application 3-2
3.1.1 Parameter Setup 3-3
3.1.2 Algorithm Instance Creation and Initialization 3-3
3.1.3 Process Call 3-3
3.1.4 Algorithm Instance Deletion 3-4
API Reference 4-1
4.1 Symbolic Constants and Enumerated Data Types 4-2
4.1.1 WMA Decoder Error Status 4-6
4.2 Data Structures 4-8
4.2.1 Common XDM Data Structures 4-8
4.2.2 WMA Decoder Data Structures 4-16
4.3 Interface Functions 4-20

4.3.1 Creation APIs 4-20
4.3.2 Initialization API..4-22
4.3.3 Control API..4-23
4.3.4 Data Processing API..4-25
4.3.5 Termination API 4-27

Data Fields for WMA RCA Header A-1

Figures

Figure 2-1. Component Directory Structure 2-2
Figure 3-1. Test Application Sample Implementation..3-2

This page is intentionally left blank

Tables

Table 1-1. List of Abbreviations vi
Table 2-1. Component Directories 2-3
Table 4-1. List of Enumerated Data Types 4-2
Table 4-2. WMA Decoder Error Status 4-6

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 WMA Version9 Decoder on the C64x+ based SoC and its supported features.

Topic	Page
1.1 Overview of XDAIS and XDM	**1-2**
1.2 Overview of WMA Version9 Decoder	**1-4**
1.3 Supported Services and Features	**1-4**

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

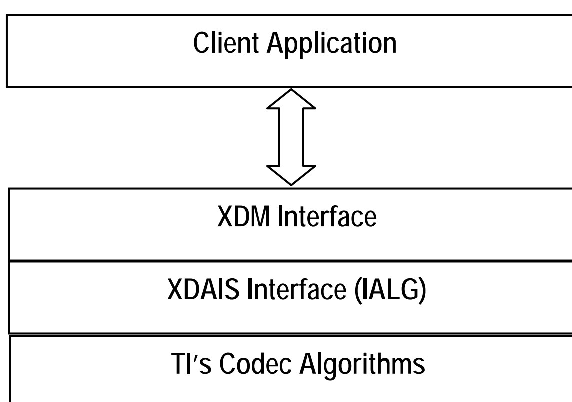
(for example audio, video, image, and speech). The XDM standard defines the following two APIs:

- `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 WMA Version9 Decoder

The WMA Version9 Decoder is WMA standard decoder that decodes Windows Media Audio files in the Raw Compressed Audio (RCA) format.

From this point onwards, all references to WMA Decoder mean WMA Version9 Decoder only.

1.3 Supported Services and Features

This user guide accompanies TI's implementation of WMA Decoder on the C64x+ based SoC.

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

- Supports all versions, namely V2, V7, V8, V9, V9 beta odd, and V9 NC
- Supports Class 4 implementation of WMA decoder
- Supports low, medium, and high bit-rates
- Supports Variable Bit Rate (VBR) mode
- Supports 8-48 kHz output sampling rates and 5-384 kbps input bit-rates
- Supports maximum of two channels
- Compliant with Microsoft Acceptance Test Criteria
- Supports Raw Compressed Audio (RCA) streams
- Outputs 16-bit PCM samples
- Does not support Digital Rights Management (DRM)
- eXpressDSP Digital Media (XDM 1.0 IAUDDEC1) 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	2-2
2.2 Installing the Component	2-2
2.3 Before Building the Sample Test Application	2-4
2.4 Building and Running the Sample Test Application	2-4
2.5 Configuration Files	2-5
2.6 Standards Conformance and User-Defined Inputs	2-6
2.7 Uninstalling the Component	2-6
2.8 Evaluation Version	2-6

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 DM6446 EVM with XDS560 emulator.

This codec supports any C64x+ based device.

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 3.2.37.12.
- **Code Generation Tools:** This project is compiled, assembled, archived, and linked using the code generation tools version 6.0.8.

2.2 Installing the Component

The codec component is released as a compressed archive. To install the codec, extract the contents of the zip file onto your local hard disk. The zip file extraction creates a top-level directory called 100_A_WMA_D_1_20_00, under which another directory named C64XPLUS_RCA is created.

Figure 2-1 shows the sub-directories created in C64XPLUS_RCA directory.

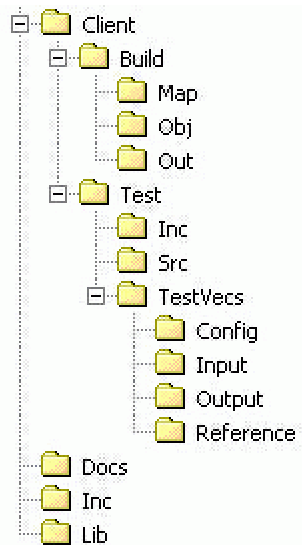


Figure 2-1. Component Directory Structure

> **Note:**
>
> If you are installing an evaluation version of this codec, the parent directory name will be 100E_A_WMA_D_1_20_00.

Table 2-1 provides a description of the sub-directories created in the C64XPLUS_RCA 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 files
\Docs	Contains user guide and datasheet
\Client\Build	Contains the sample test application project (.pjt) file
\Client\Build\Map	Contains the memory map generated on compilation of the code
\Client\Build\Obj	Contains the intermediate .asm and/or .obj file generated on compilation of the code
\Client\Build\Out	Contains the final application executable (.out) file generated by the sample test application
\Client\Test\Src	Contains application C files
\Client\Test\Inc	Contains header files needed for the application code
\Client\Test\TestVecs\Input	Contains input test vectors
\Client\Test\TestVecs\Output	Contains output generated by the codec
\Client\Test\TestVecs\Reference	Contains read-only reference output to be used for verifying 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.

This version of the codec has been validated with DSP/BIOS version 5.31.

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.

2.4 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 3.2.37.12 and code generation tools version 6.0.8.

2) Verify that the codec object library wma_tii_rca.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)
   - "Decoder output dump completed" (for output dump mode)

2.5 Configuration Files

This codec is shipped along with a generic configuration file (Testvecs.cfg) that specifies input and reference files for the sample test application.

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
Input
Output/Reference
Y
```

where:

- `X` may be set as:
  - 1 - for compliance checking, no output file is created
  - 0 - for writing the output to the output file
- `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)
- `Y` is the required channel mode and may be set as:
  - 0 – for mono output
  - 1 – for stereo output
  - 2 – for dual mono output

A sample Testvecs.cfg file is as shown:

```
0
..\..\Test\TestVecs\Input\test1_WMA_v8_5kbps_8kHz_1.rca
..\..\Test\TestVecs\Output\test1_WMA_v8_5kbps_8kHz_1.wav
0

1
..\..\Test\TestVecs\Input\test1_WMA_v8_5kbps_8kHz_1.rca
..\..\Test\TestVecs\Reference\test1_WMA_v8_5kbps_8kHz_1.wav
0
```

> **Note:**
>
> - If you are running the decoder in compliance mode, set the required `channel mode` same as the number of channels in the input file, else the compliance check will fail.
> - Compliance check is not applicable to evaluation version.

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 located 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:
   - "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 an audible tone heard every 300 frames.

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 IAUDDEC1 base class of the WMA Decoder library. The main test application file is TestAppDecoder.c. This file is available in the \Client\Test\Src sub-directory.

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

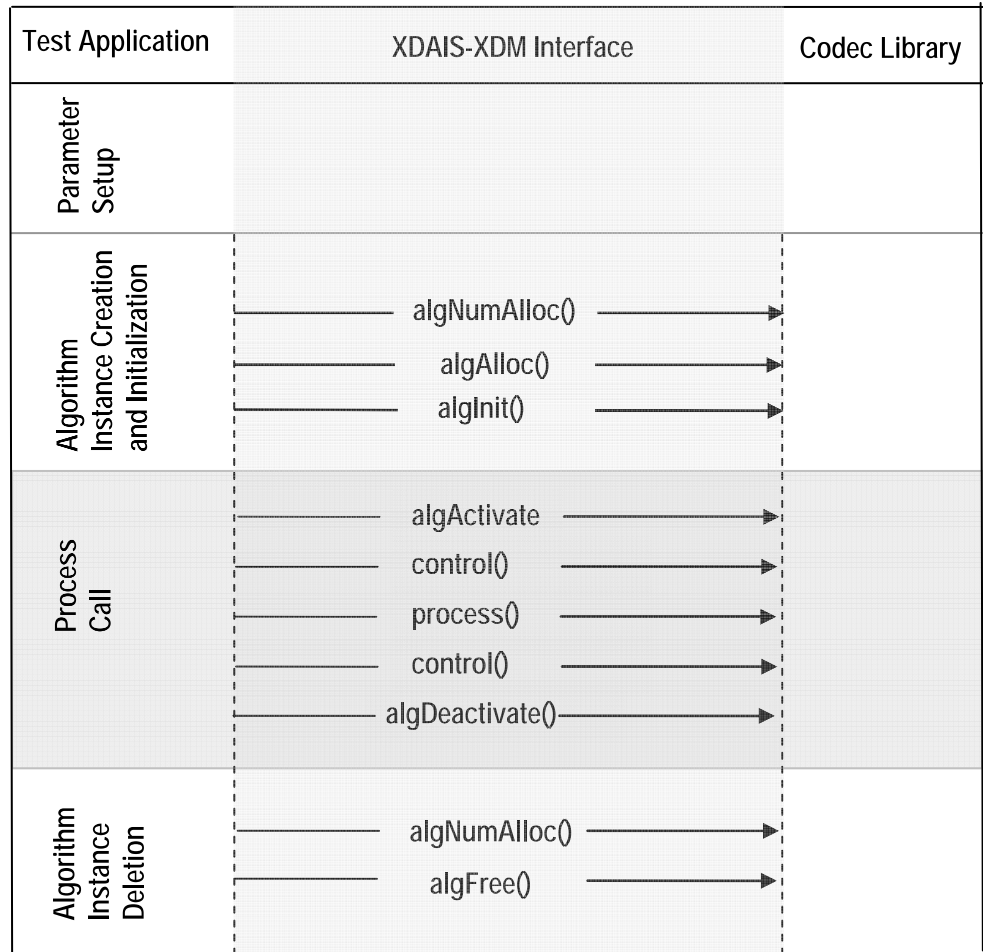


Figure 3-1. Test Application Sample Implementation

> **Note:**
>
> Audio codecs do not use `algActivate()` and `algDeactivate()` APIs.

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 during initialization. For example, a video codec requires parameters such as video height, video width, and so on. The test application obtains the required parameters from the Decoder configuration files.

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

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

2) 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.

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 `IAUDDEC1_InArgs` and `IAUDDEC1_OutArgs` structures.

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

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

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

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

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

In the sample test application, after calling `process()`, 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 deletes the current algorithm instance. The following APIs are called in sequence:

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

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

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.

Chapter 4

API Reference

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

Topic	Page
4.1 Symbolic Constants and Enumerated Data Types	4-2
4.2 Data Structures	4-8
4.3 Interface Functions	4-20

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
IAUDIO_ChannelMode		The following channel modes not only indicate the number of channels, but also the order in which the channels are placed in the input or output buffer.
	IAUDIO_1_0	Single channel (mono)
	IAUDIO_2_0	Two channel (stereo)
	IAUDIO_11_0	Two channel (dual mono)
	IAUDIO_3_0	Left, Right, Center. Not supported in this version of WMA Decoder.
	IAUDIO_2_1	Left, Right, Sur. Not supported in this version of WMA Decoder.
	IAUDIO_3_1	Left, Right, Center, Sur. Not supported in this version of WMA Decoder.
	IAUDIO_2_2	Left, Right, SurL, SurR. Not supported in this version of WMA Decoder.
	IAUDIO_3_2	Left, Right, Center, SurL, SurR. Not supported in this version of WMA Decoder.
	IAUDIO_2_3	Left, Right, SurL, SurR, SurC. Not supported in this version of WMA Decoder.
	IAUDIO_3_3	Left, Right, Center, SurL, SurR, SurC. Not supported in this version of WMA Decoder.
	IAUDIO_3_4	Left, Right, Center, SurL, SurR, sideL, sideR. Not supported in this version of WMA Decoder.
IAUDIO_DualMonoMode	IAUDIO_DUALMONO_LR	Play both left and right channel

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	IAUDIO_DUALMONO_LEFT	Play only left channel. Not supported in this version of WMA Decoder.
	IAUDIO_DUALMONO_RIGHT	Play only right channel Not supported in this version of WMA Decoder.
	IAUDIO_DUALMONO_LR_MIX	Mix and play. Not supported in this version of WMA Decoder.
IAUDIO_PcmFormat	IAUDIO_BLOCK	Left channel data followed by right channel data. Note: For single channel (mono), only one channel data is available.
	IAUDIO_INTERLEAVED	Left and right channel data interleaved. Note: For single channel (mono), only one channel data is available.
XDM_AccessMode	XDM_ACCESSMODE_READ	The algorithm reads from the buffer using the CPU.
	XDM_ACCESSMODE_WRITE	The algorithm writes to the buffer using the CPU.
XDM_DataFormat	XDM_BYTE	Big endian stream
	XDM_LE_16	16-bit little endian stream
	XDM_LE_32	32-bit little endian stream
	XDM_LE_64	64 bit little endian stream
	XDM_BE_16	16-bit big endian stream
	XDM_BE_32	32-bit big endian stream
	XDM_BE_64	64-bit big endian stream
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.

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	XDM_GETBUFINFO	Query algorithm instance regarding the properties of input and output buffers.
	XDM_GETVERSION	Query the algorithms version. Not supported in this version of WMA Decoder.
XDM_ErrorBit		The bit fields in the 32-bit error code are interpreted as shown.
	XDM_PARAMSCHANGE	Bit 8 ❑ 1 - Key parameter in input changed ❑ 0 - Ignore (Applicable only to transcoders, hence not implemented in WMA Decoder)
	XDM_APPLIEDCONCEALMENT	Bit 9 ❑ 1 - Applied concealment ❑ 0 - Ignore
	XDM_INSUFFICIENTDATA	Bit 10 ❑ 1 - Insufficient input data ❑ 0 - Ignore
	XDM_CORRUPTEDDATA	Bit 11 ❑ 1 - Invalid data ❑ 0 - Ignore
	XDM_CORRUPTEDHEADER	Bit 12 ❑ 1 - Corrupted frame header ❑ 0 - Ignore
	XDM_UNSUPPORTEDINPUT	Bit 13 ❑ 1 - Feature not supported/parameter in input ❑ 0 - Ignore
	XDM_UNSUPPORTEDPARAM	Bit 14 ❑ 1 - Input parameter that is not supported or configuration ❑ 0 - Ignore
	XDM_FATALERROR	Bit 15 ❑ 1 - Fatal error (stop decoding) ❑ 0 - Recoverable error

Note:

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

- Bit 16-32: Reserved
- Bit 8: Reserved
- Bit 0-7: Codec and implementation specific (see Table 4-2)

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

XDM_FLUSH is not applicable for WMA Decoder. Returns IALG_EOK.

4.1.1 WMA Decoder Error Status

The WMA 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. WMA Decoder Error Status

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
tWMAFileStatus	cWMA_NoErr	0	Successful decoding
	cWMA_Failed	1	Decoding failed due to NULL pointer or internal error.
	cWMA_BadArgument	2	Parameter passed in the function call is out of bounds.
	cWMA_BadHeader	3	Error in the header
	cWMA_BadPacketHeader	4	Error in the packet header
	cWMA_BrokenFrame	5	Encountered an error during frame decoding
	cWMA_NoMoreFrames	6	All input data has been decoded
	cWMA_BadSamplingRate	7	Sampling rate of the encoded file is not supported by this decoder
	cWMA_BadNumberOfChannels	8	Channel number of the encoded file is not supported
	cWMA_BadVersionNumber	9	Version number of the encoded file is not supported. Not returned by present library.
	cWMA_BadWeightingMode	10	The mode (a parameter of the encoded file is not supported). Not returned by present library.
	cWMA_BadPacketization	11	Not returned by present library.
	cWMA_BadDRMType	12	This source code does not support DRM. Not returned by the present library.
	cWMA_DRMFailed	13	DRM processing failed. Not returned by the present library.
	cWMA_DRMUnsupported	14	DRM is not supported
	cWMA_DemoExpired	15	The number of samples for the demo is greater than the limit defined. The present library does not support the demo API. Hence, this code is not returned by it.

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
	cWMA_BadState	16	Not returned by the present library
	cWMA_Internal	17	Decoder encountered an error in the bit-stream and further decoding is not possible
	cWMA_NoMoreDataThisTime	18	Not returned by the current library
	cWMA_HandleNotCreated	19	Not returned by the current library

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:

- XDM1_BufDesc
- XDM1_SingleBufDesc
- XDM_AlgBufInfo
- IAUDDEC1_Fxns
- IAUDDEC1_Params
- IAUDDEC1_DynamicParams
- IAUDDEC1_InArgs
- IAUDDEC1_Status
- IAUDDEC1_OutArgs

4.2.1.1 XDM1_BufDesc

|| Description

This structure defines an array of buffer descriptors for input and output buffers.

|| Fields

Field	Datatype	Input/ Output	Description
numBufs	XDAS_Int32	Input	Number of buffers
descs[XDM_MAX_IO_BUFFERS]	XDM1_SingleBufDesc	Input	Array of single buffer descriptors. See XDM1_SingleBufDesc data structure for more details

4.2.1.2 XDM1_SingleBufDesc

|| Description

This structure defines a single buffer descriptor.

|| Fields

Field	Datatype	Input/ Output	Description
*buf	XDAS_Int8	Input	Pointer to a buffer address
bufSize	XDAS_Int32	Input	Size of buf in 8-bit bytes
accessMask	XDAS_Int32	Output	Mask filled by the algorithm, declaring how the buffer was accessed by the algorithm. See XDM_AccessMode enumeration for details

4.2.1.3 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 WMA Decoder, the buffer details are:

- Number of input buffer required is 1
- Number of output buffer required is 1
- The input buffer size should be greater than the encoded data packet size. The maximum encoded data packet size can be upto 25 K-bytes.
- The output buffer size is 16348 bytes (4096 samples * 2 channels * 2 bytes/sample)

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

4.2.1.4 IAUDDEC1_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 IAUDDEC1_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 specify 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.
outputPCMWidth	XDAS_Int32	Input	Number of bits per output PCM sample.
pcmFormat	XDAS_Int32	Input	To set interleaved/block format for output. See IAUDIO_PcmFormat enumeration for details.
dataEndianness	XDAS_Int32	Input	Endianness of input data. See XDM_DataFormat enumeration for details.

Note:

- The current WMA Decoder implementation supports XDM_BYTE format. Input data is big endian as per standard and output is little endian.
- WMA Decoder supports only 16 bits per output PCM sample, hence

`outputPCMWidth` can only take a value of 16.

- Both interleaved and block output formats are supported.The default output format is interleaved.

4.2.1.6 IAUDDEC1_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.
downSampleSbrFlag	XDAS_Int32	Input	Flag to indicate downsampling for SBR Not supported in this version of WMA Decoder. This flag is ignored by the decoder.

4.2.1.7 IAUDDEC1_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	Number of valid input data (in bytes) in input buffer. For example, if number of valid input data in input buffer is 128 bytes, set this field to 128.
desiredChannelMode	XDAS_Int32	Input	Required channel configuration (see the `IAUDIO_ChannelMode` enumeration for details)
lfeFlag	XDAS_Int32	Input	Flag indicating whether LFE channel data is desired in the output. WMA Decoder does not support LFE channel and this field must be set to `XDAS_FALSE`. Any other value will return an error.

Note:

- If desiredChannelMode is set to IAUDIO_1_0, the decoder converts stereo input files to mono by averaging the left and right channel samples. Mono input files will remain unaffected.
- If desiredChannelMode is set to IAUDIO_2_0, the decoder converts mono input files to stereo by copying same data to the left and right channels. Stereo input files will remain unaffected.
- If desiredChannelMode is set to IAUDIO_11_0, the decoder will behave in the same way as noted for IAUDIO_2_0. The only difference is the channelMode member in IAUDDEC1_Status (Section 4.2.1.8) structure will be set to IAUDIO_11_0.

4.2.1.8 IAUDDEC1_Status

|| Description

This structure defines parameters that describe the status of the 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 enumeration for XDM compliant encoders and decoders. See XDM_ErrorBit enumeration for details.
data	XDM1_SingleBufDesc	Input/Output	Buffer descriptor for data passing. See the XDM1_SingleBufDesc data structure for more details Not supported in this version of WMA Decoder.
validFlag	XDAS_Int32	Output	Reflects the validity of this status structure. Set to XDAS_FALSE in case of an error, else normally set to XDAS_TRUE.
lfeFlag	XDAS_Int32	Output	Flag indicating the presence of LFE channel in output. Always returned as XDAS_FALSE by WMA Decoder.
bitRate	XDAS_Int32	Output	Bit-rate in bits per second. For example, if the value of this field is 128000, it indicates that bit-rate is 128 kbps.
sampleRate	XDAS_Int32	Output	Sampling frequency in Hertz (Hz). For example, if the value of this field is 44100, it indicates that the sample rate is 44.1kHz.

Field	Datatype	Input/ Output	Description
channelMode	XDAS_Int32	Output	Number of channels. See IAUDIO_ ChannelMode enumeration for details.
pcmFormat	XDAS_Int32	Output	The output PCM format. See IAUDIO_PcmFormat enumeration for details.
numSamples	XDAS_Int32	Output	Number of samples decoded per decode call
outputBitsPerSample	XDAS_Int32	Output	Number of output bits per output sample. For example, if the value of the field is 16, it indicates 16 output bits per PCM sample.
bufInfo	XDM_AlgBufInfo	Output	Input and output buffer information. See XDM_AlgBufInfo data structure for details.
dualMonoMode	XDAS_Int32	Output	Mode to indicate type of dual mono. Only used in case of dual mono output. WMA Decoder supports only IAUDIO_DUALMONO_LR mode. See the IAUDIO_DualMonoMode enumeration for details

Note:

WMA Decoder supports only IAUDIO_DUALMONO_LR mode.

4.2.1.9 IAUDDEC1_OutArgs

|| Description

This structure defines the run-time output arguments for the 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 enumeration for XDM compliant encoders and decoders. See XDM_ErrorBit data structure for details.
bytesConsumed	XDAS_Int32	Output	Bytes consumed during the process call
numSamples	XDAS_Int32	Output	Number of output samples per channel
channelMode	XDAS_Int32	Output	Number of channels. See IAUDIO_

Field	Datatype	Input/ Output	Description
			ChannelMode enumeration for details.
lfeFlag	XDAS_Int32	Output	Flag indicating the presence of LFE channel in output. Always returned as XDAS_FALSE by WMA Decoder.
dualMonoMode	XDAS_Int32	Output	Mode to indicate type of dual mono. Only used in case of dual mono output. See the IAUDIO_DualMonoMode enumeration for details
sampleRate	XDAS_Int32	Output	Sampling frequency in Hertz

Note:

WMA Decoder supports only IAUDIO_DUALMONO_LR mode.

4.2.2 WMA Decoder Data Structures

This section includes the following WMA Decoder specific extended data structures:

- IWMA_Params
- IWMA_DynamicParams
- IWMA_InArgs
- IWMA_Status
- IWMA_OutArgs
- tWMAFileHeader

4.2.2.1 IWMA_Params

|| Description

This structure defines the creation parameters and any other implementation specific parameters for the WMA Decoder instance object. The creation parameters are defined in the XDM data structure, `IAUDDEC1_Params`.

Fields

Field	Datatype	Input/ Output	Description
auddec_params	IAUDDEC1_Params	Input	See IAUDDEC1_Params data structure for details.

4.2.2.2 IWMA_DynamicParams

|| Description

This structure defines the run-time parameters and any other implementation specific parameters for the WMA Decoder instance object. The run-time parameters are defined in the XDM data structure, `IAUDDEC1_DynamicParams`.

|| Fields

Field	Datatype	Input/ Output	Description
auddec_dynamicparams	IAUDDEC1_DynamicParams	Input	See IAUDDEC1_DynamicParams data structure for details.

4.2.2.3 IWMA_InArgs

|| Description

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

|| Fields

Field	Datatype	Input/ Output	Description
auddec_inArgs	IAUDDEC1_InArgs	Input	See IAUDDEC1_InArgs data structure for details.

4.2.2.4 IWMA_Status

|| Description

This structure defines parameters that describe the status of the WMA Decoder and any other implementation specific parameters. The status parameters are defined in the XDM data structure, `IAUDDEC1_Status`.

|| Fields

Field	Datatype	Input/ Output	Description
auddec_status	IAUDDEC1_Status	Output	See IAUDDEC1_Status data structure for details.
hdr	tWMAFileHeader	Output	ASF header information. See Section 4.2.2.6 for details.

4.2.2.5 IWMA_OutArgs

|| Description

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

|| Fields

Field	Datatype	Input/ Output	Description
auddec_outArgs	IAUDDEC1_OutArgs	Output	See IAUDDEC1_OutArgs data structure for details.

4.2.2.6 tWMAFileHeader

|| Description

This structure defines the WMA Decoder file header information.

|| Fields

Field	Datatype	Input/ Output	Description
version	XDAS_UInt16	Output	Version of the WMA bit stream
sample_rate	XDAS_UInt32	Output	Sampling rate
num_channels	XDAS_UInt16	Output	Number of audio channels
duration	XDAS_UInt32	Output	Duration of the file in milliseconds
packet_size	XDAS_UInt32	Output	Size of an ASF packet

Field	Datatype	Input/ Output	Description
first_packet_offset	XDAS_UInt32	Output	Byte offset to the first ASF packet
last_packet_offset	XDAS_UInt32	Output	Byte offset to the last ASF packet
has_DRM	XDAS_UInt32	Output	Set to 1 if stream is DRM encrypted
LicenseLength	XDAS_UInt32	Output	License length in the header
bitrate	XDAS_UInt32	Output	Bit-rate of the WMA bit-stream
pcm_format_tag	XDAS_UInt16	Output	wFormatTag in pcm header
bits_per_sample	XDAS_UInt16	Output	Number of bits per sample of mono data (container size, always multiple of 8)
valid_bits_per_sample	XDAS_UInt16	Output	Actual valid bits per sample of mono data (less than or equal to bits_per_sample)
subformat_data1	XDAS_UInt16	Output	GUID information
subformat_data2	XDAS_UInt16	Output	GUID information
subformat_data3	XDAS_UInt16	Output	GUID information
subformat_data4[8]	XDAS_Uint8	Output	GUID information
channel_mask	XDAS_UInt32	Output	Data extracted from the ASF header. Required by core audio decoder for initialization.

4.3 Interface Functions

This section describes the Application Programming Interfaces (APIs) used in the WMA 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).

> **Note:**
>
> Audio codecs do not use `algActivate()` and `algDeactivate()` APIs.

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) (IAUDDEC1_Handle handle,
IAUDDEC1_Cmd id, IAUDDEC1_DynamicParams *params,
IAUDDEC1_Status *status);
```

|| Arguments

```
IAUDDEC1_Handle handle; /* algorithm instance handle */

IAUDDEC1_Cmd id; /* algorithm specific control commands*/

IAUDDEC1_DynamicParams *params /* algorithm run-time
parameters */

IAUDDEC1_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 IAUDDEC1_DynamicParams and IAUDDEC1_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:**
>
> Audio codecs do not use `algActivate(), algDeactivate()`, and `DMAN3_init()` APIs.

4.3.4 Data Processing API

Data processing API is used for processing the input data.

|| Name

process() – basic encoding/decoding call

|| Synopsis

```
XDAS_Int32 (*process)(IAUDDEC1_Handle handle, XDM1_BufDesc
*inBufs, XDM1_BufDesc *outBufs, IAUDDEC1_InArgs *inargs,
IAUDDEC1_OutArgs *outargs);
```

|| Arguments

```
IAUDDEC1_Handle handle; /* algorithm instance handle */

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

XDM1_BufDesc *outBufs; /* algorithm output buffer
descriptor */

IAUDDEC1_InArgs *inargs /* algorithm runtime input
arguments */

IAUDDEC1_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 XDM1_BufDesc data structure for details).

The fourth argument is a pointer to the IAUDDEC1_InArgs data structure that defines the run-time input arguments for an algorithm instance object.

The last argument is a pointer to the IAUDDEC1_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:**
>
> Audio codecs do not use `algActivate()`, `algDeactivate()`, and `DMAN3_init()` APIs.

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()`

Appendix A

Data Fields for WMA RCA Header

This section describes the data fields for WMA RCA header section and the data fields for WMA RCA packet/payload header section.

Table A-1. Data fields for WMA RCA file header section

ASF Header Object Name	Field Name	Field Type	Size (bits)	Field Description
File Properties	Data Packets Count	QWORD	64	Number of data packets.
File Properties	Play Duration	QWORD	64	Play Duration.
File Properties	Maximum Data Packet Size	DWORD	32	Maximum data packet size
Stream Properties	Stream Type	GUID	128	Stream Type = [Audio Media Type] GUID: F8699E40-5B4D-11CF-A8FD-00805F5C442B
Stream Properties	Type-Specific Data Length	DWORD	32	Data length is 28 for WMASTD.
Stream Properties	Stream Number	WORD	16	Stream Number
Stream Properties	Codec ID / Format Tag	WORD	16	Type Specific Data: Codec ID / Format Tag Format = 0x161 [Windows Media Audio]
Stream Properties	Number of Channels	WORD	16	Type Specific Data: Number of channels
Stream Properties	Samples Per Second	DWORD	32	Type Specific Data: Samples per second
Stream Properties	Average Number of Bytes Per Second	DWORD	32	Type Specific Data: Average number of bytes per second
Stream Properties	Block Alignment	WORD	16	Type Specific Data: Block alignment
Stream Properties	Bits Per Sample	WORD	16	Type Specific Data: Bits per sample

ASF Header Object Name	Field Name	Field Type	Size (bits)	Field Description
Stream Properties	iSize Wave Header	WORD	16	Used only for PRO and Lossless (Not used in WMASTD)
Stream Properties	Dw Samples Per Block	DWORD	32	Type Specific Data: Codec specific data: dwSamplesPerBlock (1-4th byte of codec specific data)
Stream Properties	Wencode Options	WORD	16	Type Specific Data: Codec specific data: wEncodeOptions (5-6th byte of Codec Specific Data)
Stream Properties	Channel Mask	DWORD	32	Determined based on number of channels.

Table A-2. Data fields for WMA RCA packet/payload header section

ASF Data Object Field.	Field Name	Field Type	Size (bits)
Payload Header	Replicated Data Length	BYTE	8
Payload Header	Audio Payload Size. (Replicated data field, for Non Compressed Payload or explicitly calculated Payload Size For Compressed Payload)	DWORD	32