MP3 Decoder on C64x+

User Guide

DAVINCI™
TEXAS INSTRUMENTS

Literature Number: SPRUEY5
April 2007

TEXAS INSTRUMENTS

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 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.

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.

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) MP3 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* (SPRU352) defines a set of requirements for DSP algorithms that, if followed, allow system integrators to quickly assemble production-quality systems from one or more such algorithms.
- *TMS320 DSP Algorithm Standard API Reference* (SPRU360) describes all the APIs that are defined by the TMS320 DSP Algorithm Interface Standard (also known as XDAIS) specification.
- *Technical Overview of eXpressDSP - Compliant Algorithms for DSP Software Producers* (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* (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* (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* (SPRA789) describes a C6x1x-optimized (C6211, C6711) ACPY2 library implementation and DMA Resource Manager.
- *TMS320C64x+ Megamodule* (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* (SPRU871) describes the C64x+ megamodule peripherals.
- *TMS320C64x to TMS320C64x+ CPU Migration Guide* (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* (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* (SPRU732) describes the CPU architecture, pipeline, instruction set, and interrupts of the C64x and C64x+ DSPs.
- *TMS320DM6446 Digital Media System-on-Chip* (SPRS283)
- *TMS320DM6446 Digital Media System-on-Chip Errata (Silicon Revision 1.0)* (SPRZ241) describes the known exceptions to the functional specifications for the TMS320DM6446 Digital Media System-on-Chip (DMSoC).
- *TMS320DM6443 Digital Media System-on-Chip* (SPRS282)
- *TMS320DM6443 Digital Media System-on-Chip Errata (Silicon Revision 1.0)* (SPRZ240) describes the known exceptions to the functional specifications for the TMS320DM6443 Digital Media System-on-Chip (DMSoC).
- *TMS320DM644x DMSoC DSP Subsystem Reference Guide* (SPRUE15) describes the digital signal processor (DSP) subsystem in the TMS320DM644x Digital Media System-on-Chip (DMSoC).
- *TMS320DM644x DMSoC ARM Subsystem Reference Guide* (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:

- *ISO/IEC IS 11172-3 Information Technology -- Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to about 1.5 Mbps -- Part 3: Audio*
- *ISO/IEC IS 13818-3 Information Technology -- Generic Coding of Moving Pictures and Associated Audio Information -- Part 3: Audio*

Abbreviations

The following abbreviations are used in this document:

Table 1-1. List of Abbreviations

Abbreviation	Description
API	Application Programming Interface
CBR	Constant Bit Rate
EVM	Evaluation Module
Kbps	Kilo bits per second
MP3	MPEG1 Layer 3
MPEG	Motion Picture Expert Group
PCM	Pulse Code Modulation
VBR	Variable Bit Rate
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 (MP3 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.

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 MP3 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

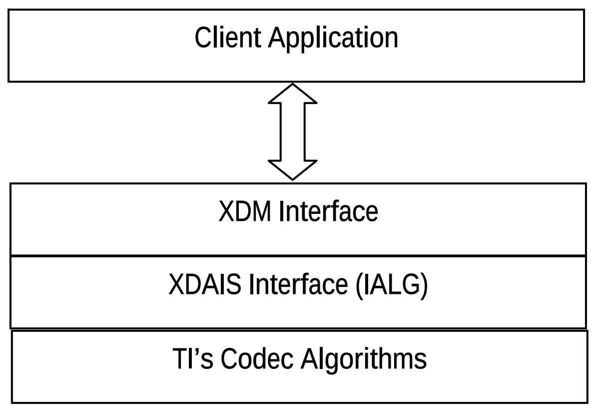
In the multimedia application space, you have the choice of integrating any codec into your multimedia system. For example, if you are building a video decoder system, you can use any of the available video decoders (such as MPEG4, H.263, or H.264) in your system. To enable easy integration with the 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 MP3 Decoder

MP3 is one of the most popular audio compression standards across wide spectrum of application ranging from portable player, cell phones, music systems, internet, and so forth.

1.3 Supported Services and Features

This user guide accompanies TI's implementation of MP3 Decoder on the C64x+ platform. This version of the codec has the following supported features:

- Supports ISO/IEC 11172-3 Layer 1, Layer 2, and Layer 3 compliant streams.
- Supports Variable Bit Rate (VBR) and Constant Bit Rate (CBR) modes. The VBR encoding provides a higher overall sound quality with smaller file size.
- Supports bit rates of 32 to 448 kbps for Layer 1, 32 to 384 kbps for Layer 2, and 8 to 320 kbps for Layer 3.
- Supports mono, stereo, and dual channel input streams.
- Outputs 16-bit raw Pulse Code Modulation (PCM) samples. If two channels of audio data are produced, the output can be either in interleaved or block format.
- Layer 1 and Layer 2 decoder is compliant only with ISO/IEC 11172-3 (MPEG1 audio) standard.
- Layer 3 decoder is compliant with the following standards:
  - ISO/IEC 11172-3 (MPEG 1) (48 KHz, 44.1 KHz, and 32 KHz)
  - ISO/IEC 13818-3 (MPEG 2) (24 KHz, 22.05 KHz, and 16 KHz)
  - MPEG 2.5 extension (12 KHz, 11.025 KHz, and 8 KHz) sampling rates
- Does not support free format streams.
- 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 DM6437 EVM with XDS560 USB.

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.
- **DSP/BIOS:** This project has been validated with DSP/BIOS version 5.21.

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 parent directory called 100_A_MP3_D_1_10_00, under which another directory named DM6437_L1L2L3 is created. Figure 2-1 shows the sub-directories created in DM6437_L1L2L3.

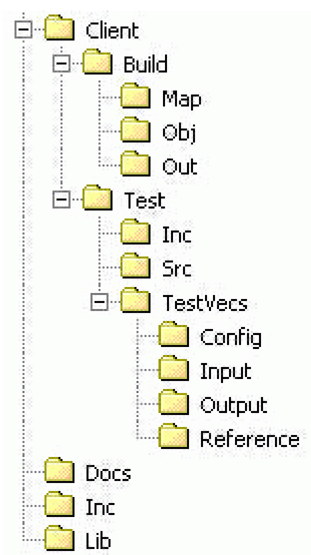


Figure 2-1. Component Directory Structure

Note:

If you are installing an evaluation version of this codec, the parent directory name will be 100E_A_MP3_D_1_10_00.

Table 2-1 provides a description of the sub-directories created in the DM6437_L1L2L3 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 manual, 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 .obj and/or .asm 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 5.21.

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 Compose 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

This codec is accompanied by a sample test application. This application will run in TI's Code Composer Studio development environment. To build and run the sample application in Code Composer Studio, follow these steps:

1) Verify that you have an installation of 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 mp3dec_tii_l1l2l3.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 the message "Decoder compliance test passed/failed" for each frame.

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
```

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

  The default setting of Testvecs.cfg file is for compliance checking.
- `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\Input\fl11.mp3
..\..\Test\TestVecs\Reference\fl11.pcm
0
..\..\Test\TestVecs\Input\fl11.mp3
..\..\Test\TestVecs\Output\fl11.pcm
```

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 message 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.

Note:

The comparison is valid only with a set of vectors provided as part of the release package

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, an audible tone will be heard occasionally.

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 IAUDDEC base class of the MP3 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.

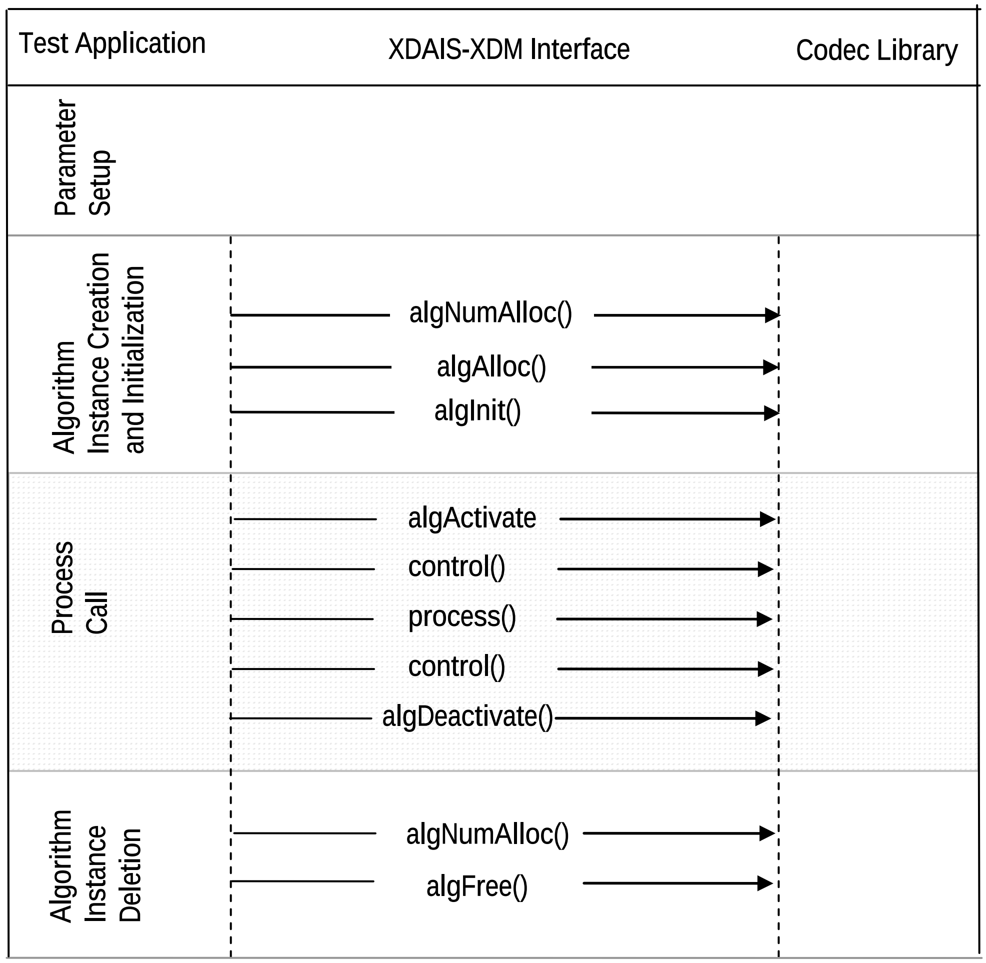


Figure 3-1. Test Application Sample Implementation

> **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 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 performs the following:

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

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

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 performs 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 `IAUDDEC_InArgs` and `IAUDDEC_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 etc., 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 etc., 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 must delete 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.

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
IAUDIO_ChannelId	IAUDIO_MONO	Single channel
	IAUDIO_STEREO	Two channel
	IAUDIO_THREE_ZERO	Three channel. Not supported in this version of MP3 Decoder
	IAUDIO_FIVE_ZERO	Five channel. Not supported in this version of MP3 Decoder
	IAUDIO_FIVE_ONE	5.1 channel. Not supported in this version of MP3 Decoder
	IAUDIO_SEVEN_ONE	7.1 channel. Not supported in this version of MP3 Decoder
IAUDIO_PcmFormat	IAUDIO_BLOCK	Left channel data followed by right channel data. **Note**: For single channel (mono), right channel data will be same as left channel data.
	IAUDIO_INTERLEAVED	Left and right channel data interleaved. **Note**: For single channel (mono), right channel data will be same as left channel data.
XDM_DataFormat	XDM_BYTE	Big endian stream
	XDM_LE_16	16-bit little endian stream
	XDM_LE_32	32-bit little endian stream
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

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	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. Not applicable for MP3 Decoder. Just returns IALG_EOK.
	XDM_GETBUFINFO	Query algorithm instance regarding the properties of input and output buffers
XDM_ErrorBit		The bit fields in the 32-bit error code are interpreted as shown.
	XDM_APPLIEDCONCEALMENT	Bit 9 ❑ 1 - Applied concealment ❑ 0 - Ignore Not applicable for MP3 Decoder.
	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 - Unsupported feature/parameter in input ❑ 0 - Ignore
	XDM_UNSUPPORTEDPARAM	Bit 14 ❑ 1 - Unsupported input parameter or configuration ❑ 0 - Ignore Not applicable for MP3 Decoder.
	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 - 15: Reserved
- Bit 0 - 7: Codec and implementation specific.

The MP3 Decoder uses a numerical value to define specific extended errors/warnings as follows:

- 0 - No error
- 1 - Sync word not found
- 2 - Stream is not layer 3
- 3 - Free format not supported
- 4 - Main data length invalid
- 5 - Joint stereo bound error
- 6 - Insufficient input data
- 7 - Invalid input data
- 8 - Bad PCM data warning
- 9 - Change in number of channels between frames
- 10 - Change in sampling frequency between frames
- 11 - Change in bitrate between frames
- 12 - Change in layer between frames
- 13 - Error in scalefactor decoding
- 14 - Error in Huffman decoding
- 15 - Error in inverse quantization
- 16 - Error in alias cancellation
- 17 - Error in inverse MDCT
- 18 - Error in polyphase synthesis
- 19 - Internal Pointer NULL error
- 20 - CRC check failed
- 21 - Input bitstream parameters not supported

The decoder has to be reset only in case of fatal errors. Otherwise the application can continue decoding without any problem.

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`
- `IAUDDEC_Fxns`
- `IAUDDEC_Params`
- `IAUDDEC_DynamicParams`
- `IAUDDEC_InArgs`
- `IAUDDEC_Status`
- `IAUDDEC_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 MP3 Decoder, the buffer details are:

- Number of input buffer required is 1.
- Number of output buffer required is 1.
- The size of the input buffer should be such that atleast one frame of encoded data is present in the input buffer. The input buffer size (in bytes) for worst case (Layer 2) is 2512 bytes.

> - The output buffer size (in bytes) for worst case (Layer 2) is 4608 bytes.
>
> These are the maximum buffer sizes but you can reconfigure depending on the format of the bit stream

4.2.1.3 IAUDDEC_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

4.2.1.4 IAUDDEC_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.
`maxSampleRate`	`XDAS_Int32`	Input	Maximum sampling frequency to be supported in Hertz (Hz). For example, if maximum sampling frequency is 44.1 kHz, set this field to 44100.
`maxBitrate`	`XDAS_Int32`	Input	Maximum bit rate to be supported in bits per second. For example, if maximum bit rate is 128 kbps, set this field to 128000.

Field	Datatype	Input/ Output	Description
maxNoOfCh	XDAS_Int32	Input	Maximum channels to be supported. See IAUDIO_ChannelId enumeration for details
dataEndianness	XDAS_Int32	Input	Endianness of input data. See XDM_DataFormat enumeration for details.

Note:

- Currently, the MP3 decoder implementation supports XDM_BYTE format.
- For the supported maxBitrate and maxSampleRate values, see the standards document listed in the Related Documentation section.
- The maxBitrate, maxSampleRate and maxNoOfCh are only used during intialization for checking the capability of the decoder by the application. During actual decoding the MP3 Decoder will decode all the sampling frequencies, bit-rate and channels as mentioned in this user manual

4.2.1.5 IAUDDEC_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.
outputFormat	XDAS_Int32	Input	To set interleaved/block format for output. See IAUDIO_PcmFormat enumeration for details.

4.2.1.6 IAUDDEC_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.

4.2.1.7 IAUDDEC_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.
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.
numChannels	XDAS_Int32	Output	Number of channels. See IAUDIO_ChannelId enumeration for details.
numLFEChannels	XDAS_Int32	Output	Number of Low Frequency Effects (LFE) channels in the stream

Field	Datatype	Input/ Output	Description
outputFormat	XDAS_Int32	Output	The output PCM format. See IAUDIO_PcmFormat enumeration for details.
autoPosition	XDAS_Int32	Output	Flag to indicate support for random position decoding, which means that a stream can be decoded from any point: ❑ 1 - Supports random position decoding ❑ 0 - Does not support random position decoding
fastFwdLen	XDAS_Int32	Output	Recommended Fast Forward length in bytes in case of random position decoding.
frameLen	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.

4.2.1.8 IAUDDEC_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

4.2.2 MP3 Decoder Data Structures

This section includes the following MP3 Decoder specific extended data structures:

- IMP3DEC_Params
- IMP3DEC_DynamicParams
- IMP3DEC_InArgs
- IMP3DEC_Status
- IMP3DEC_OutArgs

4.2.2.1 IMP3DEC_Params

|| Description

This structure defines the creation parameters and any other implementation specific parameters for the MP3 Decoder instance object. The creation parameters are defined in the XDM data structure, IAUDDEC_Params.

|| Fields

Field	Datatype	Input/ Output	Description
auddec_params	IAUDDEC_Params	Input	See IAUDDEC_Params data structure for details.
outputBitWidth	Int	Input	Bit width of output PCM sample. ❑ 16 - 16 bit PCM sample ❑ 24 - 24 bit PCM sample **Note**: This version of MP3 Decoder supports only 16-bit output width.

4.2.2.2 IMP3DEC_DynamicParams

|| Description

This structure defines the run time parameters and any other implementation specific parameters for the MP3 Decoder instance object. The run time parameters are defined in the XDM data structure, IAUDDEC_DynamicParams.

|| Fields

Field	Datatype	Input/ Output	Description
auddec_dynamicparams	IAUDDEC_DynamicParams	Input	See IAUDDEC_DynamicParams data structure for details.
MonoToStereoCopy	XDAS_Int32	Input	For mono streams, if this field is set to: ❑ 1 - Mono to stereo copy will happen ❑ 0 - Mono to stereo conversion will not happen This field is ignored for stereo streams.
stereoToMono	XDAS_UInt16	Input	For stereo streams, if this field is set to: ❑ 1 - Stereo stream will be converted to a mono perfomed by using M = 0.5(L + R) for every sample ❑ 0 - No stereo to mono conversion This field is ignored for mono streams.

4.2.2.3 IMP3DEC_InArgs

|| Description

This structure defines the run time input arguments for the MP3 Decoder instance object.

|| Fields

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

4.2.2.4 IMP3DEC_Status

|| Description

This structure defines parameters that describe the status of the MP3 Decoder and any other implementation specific parameters. The status parameters are defined in the XDM data structure, `IAUDDEC_Status`.

|| Fields

Field	Datatype	Input/ Output	Description
`auddec_status`	`IAUDDEC_Status`	Output	See `IAUDDEC_Status` data structure for details.
`layer`	`XDAS_Int32`	Output	Provides layer information: ❑ 1 - Layer 1 ❑ 2 - Layer 2 ❑ 3 - Layer 3
`isValid`	`XDAS_Int32`	Output	Flag indicating if the last decode call was successful: ❑ 1 - Indicates last decode call was successful and the values of the fields in the `IAUDDEC_Status` structure are valid. ❑ 0 - Indicates last decode call was not successful and the values are not valid.

4.2.2.5 IMP3DEC_OutArgs

|| Description

This structure defines the run time output arguments for the MP3 Decoder instance object.

|| Fields

Field	Datatype	Input/ Output	Description
`auddec_outArgs`	`IAUDDEC_OutArgs`	Output	See `IAUDDEC_OutArgs` data structure for details.
`layer`	`XDAS_Int32`	Output	Provides layer information: ❑ 1 - Layer 1 ❑ 2 - Layer 2 ❑ 3 - Layer 3
`crcErrCnt`	`XDAS_Int32`	Output	Indicates Cyclic Redundancy Check (CRC) error. A value of zero indicates there is no CRC error. A non-zero value indicates the number of contiguous frames with CRC error.

4.3 Interface Functions

This section describes the Application Programming Interfaces (APIs) used in the MP3 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) (IAUDDEC_Handle handle, IAUDDEC_Cmd
id, IAUDDEC_DynamicParams *params, IAUDDEC_Status
*status);
```

‖ Arguments

```
IAUDDEC_Handle handle; /* algorithm instance handle */

IAUDDEC_Cmd id; /* algorithm specific control commands*/

IAUDDEC_DynamicParams *params /* algorithm run time
parameters */

IAUDDEC_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 `IAUDDEC_DynamicParams` and `IAUDDEC_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()` and `algDeActivate()` 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)(IAUDDEC_Handle handle, XDM_BufDesc
*inBufs, XDM_BufDesc *outBufs, IAUDDEC_InArgs *inargs,
IAUDDEC_OutArgs *outargs);
```

‖ Arguments

```
IAUDDEC_Handle handle; /* algorithm instance handle */

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

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

IAUDDEC_InArgs *inargs /* algorithm runtime input
arguments */

IAUDDEC_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 `IAUDDEC_InArgs` data structure that defines the run time input arguments for an algorithm instance object.

The last argument is a pointer to the `IAUDDEC_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 al`gActivate()` and al`gDeActivate()` APIs.
- The input data for MP3 Decoder is in byte format. The decoder outputs 16-bit raw PCM samples in the little-endian format. The output data is either in block or interleaved format. In the block format, samples of the left channels are stored contiguously first, followed by right channel samples (that is, LLLLRRRR). In the interleaved format, left channel samples are stored followed by the right channel samples (that is, LRLRLRLR).

4.3.5 Termination API

Termination API is used to terminate the MP3 Decoder 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()`