AAC Encoder on C64x+

User Guide

Literature Number: SPRUEX8
May 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.

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
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

Read This First

About This Manual

This document describes how to install and work with Texas Instruments' (TI) AAC Encoder 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.
- **Appendix A – AAC Encoder Bit-rate and Sampling Frequency Combination**, contains the AAC Encoder bit rate and sampling frequency combination.

Related Documentation From Texas Instruments

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

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

The following documents describe TMS320 devices and related support tools:

- *Design and Implementation of an eXpressDSP-Compliant DMA Manager for C6X1X* (literature number SPRA789) describes a C6x1x-optimized (C6211, C6711) ACPY2 library implementation and DMA Resource Manager.
- *TMS320C64x+ Megamodule* (literature number SPRAA68) describes the enhancements made to the internal memory and describes the new features which have been added to support the internal memory architecture's performance and protection.
- *TMS320C64x+ DSP Megamodule Reference Guide* (literature number SPRU871) describes the C64x+ megamodule peripherals.
- *TMS320C64x to TMS320C64x+ CPU Migration Guide* (literature number SPRAA84) describes migration from the Texas Instruments TMS320C64x™ digital signal processor (DSP) to the TMS320C64x+™ DSP.
- *TMS320C6000 Optimizing Compiler v 6.0 Beta User's Guide* (literature number SPRU187N) explains how to use compiler tools

such as compiler, assembly optimizer, standalone simulator, library-build utility, and C++ name demangler.

- *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide* (literature number SPRU732) describes the CPU architecture, pipeline, instruction set, and interrupts of the C64x and C64x+ DSPs.
- TMS320DM6446 Digital Media System-on-Chip (literature number SPRS283)
- *TMS320DM6446 Digital Media System-on-Chip Errata (Silicon Revision 1.0)* (literature number SPRZ241) describes the known exceptions to the functional specifications for the TMS320DM6446 Digital Media System-on-Chip (DMSoC).
- TMS320DM6443 Digital Media System-on-Chip (literature number SPRS282)
- *TMS320DM6443 Digital Media System-on-Chip Errata (Silicon Revision 1.0)* (literature number SPRZ240) describes the known exceptions to the functional specifications for the TMS320DM6443 Digital Media System-on-Chip (DMSoC).
- *TMS320DM644x DMSoC DSP Subsystem Reference Guide* (literature number SPRUE15) describes the digital signal processor (DSP) subsystem in the TMS320DM644x Digital Media System-on-Chip (DMSoC).
- *TMS320DM644x DMSoC ARM Subsystem Reference Guide* (literature number SPRUE14) describes the ARM subsystem in the TMS320DM644x Digital Media System on a Chip (DMSoC).

Related Documentation

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

- *ISO/IEC IS 14496-3 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 13818-7 Information Technology -- Generic Coding of Moving Pictures and Associated Audio Information -- Part 7: Advanced Audio Coding*

Abbreviations

The following abbreviations are used in this document:

Table 1-1. List of Abbreviations

Abbreviation	Description
API	Application Programming Interface
ADIF	Audio Data Interchange Format
ADTS	Audio Data Transport Stream
CBR	Constant Bit Rate
CRC	Cyclic Redundancy Check
DMA	Direct Memory Access
DMAN3	DMA Manager
EVM	Evaluation Module
HE	High Efficiency
HEv2	High Efficiency with parametric stereo
Kbps	Kilo bits per second
LC	Low Complexity
MPEG	Moving Picture Experts Group
PCM	Pulse code modulation
PNS	Perceptual noise substitution
PS	Parametric Stereo
SBR	Spectral Band Replication
TNS	Tonal Noise Shaping
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 (AAC Encoder 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

This page is intentionally left blank

Contents

Figures

This page is intentionally left blank

Tables

This page is intentionally left blank

Chapter 1

Introduction

This chapter provides a brief introduction to XDAIS and XDM. It also provides an overview of TI's implementation of the AAC Encoder on the C64x+ platform and its supported features.

Topic **Page**

1.1 Overview of XDAIS and XDM

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

1.1.1 XDAIS Overview

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

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

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

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

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

1.1.2 XDM Overview

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

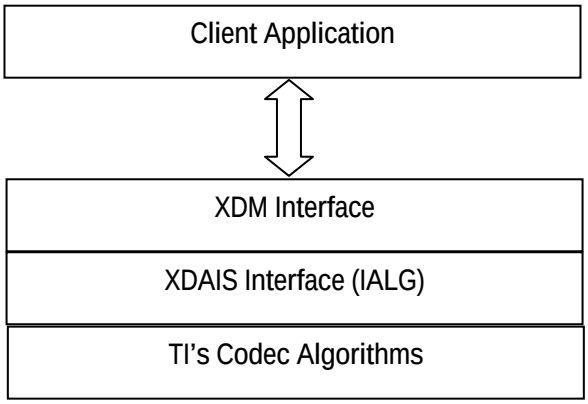
(for example audio, video, image, and speech). The XDM standard defines the following two APIs:

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

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

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

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



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

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

1.2 Overview of AAC Encoder

AAC 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 AAC Encoder on the C64x+ platform.

This version of the codec has the following supported features:

- Supports 16-bit and 32-bit PCM samples as input. In case of 32-bit PCM it takes the most significant 16-bits as input internally.
- Supports constant bit-rate (CBR) encoding and variable bit-rate (VBR) encoding
- Supports input sampling frequencies from 8 kHz to 96 kHz
- Supports only AAC-LC output format
- Supports mono, stereo and dual mono input files
- Supports bit rates based on sampling frequency and number of channels
- Supports Audio Data Interchange Format (ADIF), Audio Data Transport Stream (ADTS), and raw output format
- Compliant with the ISO/IEC 14496-3 (MPEG 4 AAC) and ISO/IEC 13818-7 (MPEG 2-AAC) standards
- eXpressDSP compliant
- eXpressDSP Digital Media (XDM) compliant

Chapter 2

Installation Overview

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

Topic **Page**

2.1 System Requirements

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

2.1.1 Hardware

This codec has been built and tested on the 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.

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_AAC_E_1_00_00, under which another directory named DM6437_LC is created.

Figure 2-1 shows the sub-directories created in the DM6437_LC directory.

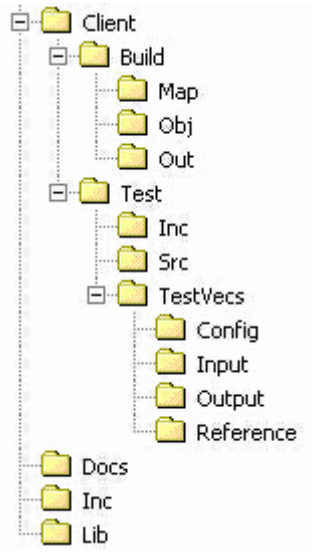


Figure 2-1. Component Directory Structure

Table 2-1 provides a description of the sub-directories created in the DM6437_LC directory.

Table 2-1. Component Directories

Sub-Directory	Description
\Inc	Contains XDM related header files which allow interface to the codec library
\Lib	Contains the codec library file
\Docs	Contains user guide, datasheet, and release notes
\Client\Build	Contains the sample test application project (.pjt) file
\Client\Build\Map	Contains the memory map generated on compilation of the code
\Client\Build\Obj	Contains the intermediate .asm and/or .obj file generated on compilation of the code
\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 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 aacenc_tii_lc.l64P exists in the \Lib sub-directory.

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

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

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

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

   The sample test application takes the input files stored in the \Client\Test\TestVecs\Input sub-directory, runs the codec, and dumps the output in the \Client\Test\TestVecs\Output directory.

2.5 Configuration Files

This codec is shipped along with:

- A generic configuration file (Testvecs.cfg) – specifies input and output files for the sample test application.

- A Encoder configuration file (Testparams.cfg) – specifies the configuration parameters used by the test application to configure the Encoder.

2.5.1 Generic Configuration File

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

The format of the Testvecs.cfg file is:

```
Input
Output
```

where:

- `Input` is the input file name (use complete path).
- `Output` is the output file name.

A sample Testvecs.cfg file is as shown:

```
..\\..\\Test\\TestVecs\\Input\\input.wav
..\\..\\Test\\TestVecs\\Output\\output.aac
```

2.5.2 Encoder Configuration File

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

A sample Testparams.cfg file is as shown:

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

-b 128000           /* bit rate */
-m 0                /* dual Mono */
-c 0                /* CRC flag */
-t 1                /* TNS flag */
-p 1                /* PNS flag */
-d 0                /* downmix flag */
-o 2                /* Output Object type */
                    /* 2 = LC */
                    /* 5 = HE */
                    /* 29 = HEv2 */
-f 2                /* Output file type */
                    /* 0 = Raw */
                    /* 1 = ADIF */
                    /* 2 = ADTS */
-v 1                /* 0 = CBR, 1 = VBR Mode 1 */
                    /* 2 = VBR Mode 2 */
                    /* 3 = VBR Mode 3 */
                    /* 4 = VBR Mode 4 */
                    /* 5 = VBR Mode 5 */
```

Any field in the `IAACENC_Params` structure (see Section 4.2.2.1) can be set in the Testparams.cfg file using the syntax shown above. If you specify additional fields in the Testparams.cfg file, ensure to modify the test application appropriately to handle these fields.

Note:

In case of VBR mode,

- VBR Mode 1 => Low quality
- VBR Mode 5 =>Very high quality.

The quality and the bit-rate increases from VBR Mode 1 to VBR Mode 5.

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.

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.

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 IAACENC base class of the AAC Encoder library. The main test application files are TestAppEncoder.c and TestAppEncoder.h. These files are available in the \Client\Test\Src and \Client\Test\Inc sub-directories respectively.

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

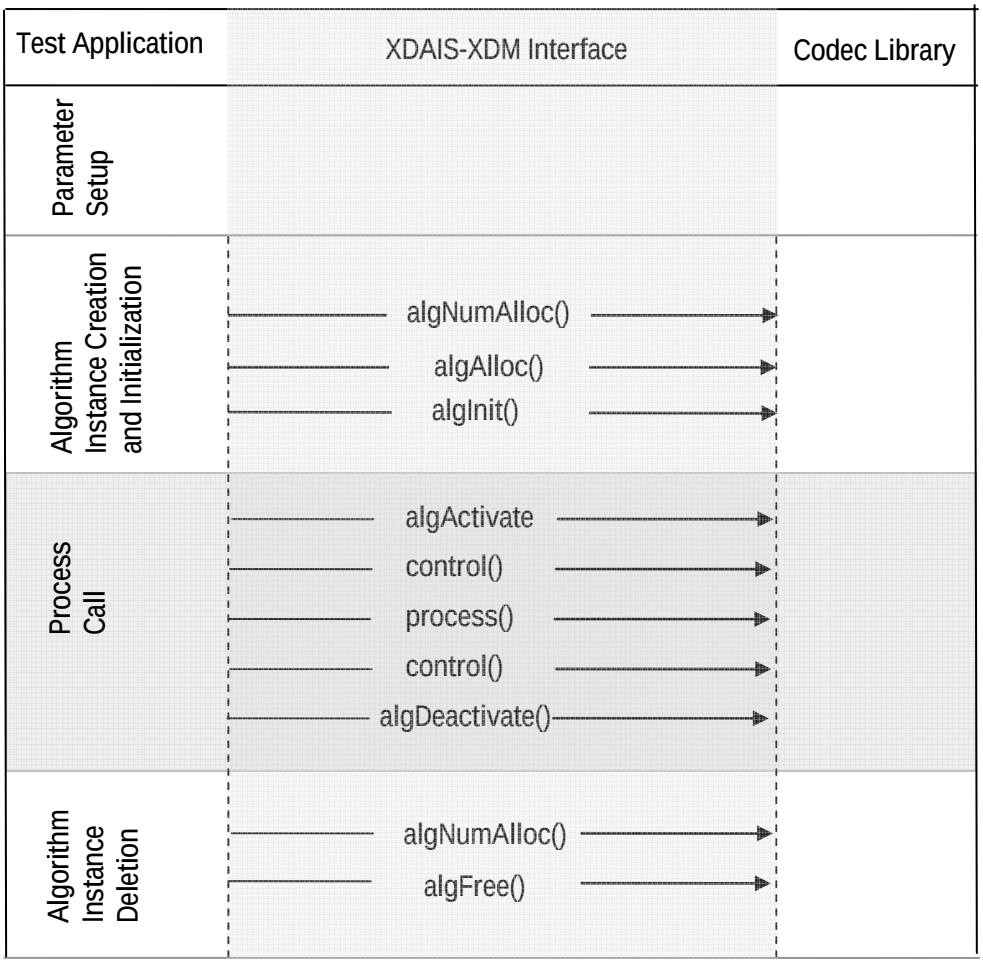


Figure 3-1. Test Application Sample Implementation

> **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 Encoder configuration files.

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

1) Opens the generic configuration file, config.txt and reads the compliance checking parameter, input file name, and output 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 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 4.2.1.5). The inputs to the `process()` function are input and output buffer descriptors, pointer to the `IAACENC_InArgs` and `IAACENC_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 Encoder 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.

This page is intentionally left blank

Chapter 4

API Reference

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

Topic **Page**

4.1 Symbolic Constants and Enumerated Data Types

This section summarizes all the symbolic constants specified as either #define macros and/or enumerated C data types. 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 channels
	IAUDIO_THREE_ZERO	Three channels. Not supported in this version of AAC Encoder.
	IAUDIO_FIVE_ZERO	Five channels. Not supported in this version of AAC Encoder.
	IAUDIO_FIVE_ONE	5.1 channels. Not supported in this version of AAC Encoder.
	IAUDIO_SEVEN_ONE	7.1 channels. Not supported in this version of AAC Encoder.
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

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	XDM_RESET	Reset the algorithm
	XDM_SETDEFAULT	Initialize all fields in `Params` structure to default values specified in the library
	XDM_FLUSH	Handle end of stream conditions. This command forces algorithm instance to output data without additional input.
	XDM_GETBUFINFO	Query algorithm instance regarding the properties of input and output buffers
AACENC_OBJ_TYP	AACENC_OBJ_TYP_LC	AAC low complexity. ❑ 2 - Low complexity
	AACENC_OBJ_TYP_HEAAC	AAC Encoder with SBR capability. ❑ 5 - SBR capability
	AACENC_OBJ_TYP_PS	AAC Encoder with SBR and PS. ❑ 29 - SBR and PS capability
AACENC_BITRATE_MODE	AACENC_BR_MODE_CBR	Constant Bit rate mode ❑ 0 – Constant bit-rate
	AACENC_BR_MODE_VBR_1	Variable bit-rate mode-1 ❑ 1 – VBR Mode 1
	AACENC_BR_MODE_VBR_2	Variable Bit-rate Mode-2 ❑ 2 – VBR Mode 2
	AACENC_BR_MODE_VBR_3	Variable Bit-rate Mode-3 ❑ 3 – VBR Mode 3
	AACENC_BR_MODE_VBR_4	Variable Bit-rate Mode-4 ❑ 4 – VBR Mode 4
	AACENC_BR_MODE_VBR_5	Variable Bit-rate Mode-5 ❑ 5 – VBR Mode 5
AACENC_TRANSPORT_TYPE	AACENC_TT_RAW	❑ 0 - Raw output format
	AACENC_TT_ADIF	❑ 1 - ADIF file format
	AACENC_TT_ADTS	❑ 2 - ADTS file format
AACENC_BOOL_TYPE	AACENC_FALSE	❑ 0 – False
	AACENC_TRUE	❑ 1 – True
XDM_ErrorBit		The bit fields in the 32-bit error code are interpreted as shown.

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	XDM_APPLIEDCONCEALMENT	Bit 9 ❑ 1 - Applied concealment ❑ 0 - Ignore Not applicable for AAC Encoder.
	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
	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.

The AAC Encoder 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. AAC Encoder Error Status

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
XDM_ErrorBit	AACENC_INVALID_PARAM	1	Invalid parameter for the encoder
	AACENC_INVALID_FREQ	2	Invalid input sampling frequency
	AACENC_INVALID_BITRATE	3	Invalid output bit rate
	AACENC_INVALID_CHANNELS	4	Invalid number of channels
	AACENC_INVALID_ELTYPE	5	Invalid element type
	AACENC_INSUFFICIENT_INPUT	6	Insufficient input PCM data
	AACENC_WRITEBITSTREAM_ERROR	7	Error during writing bit stream
	AACENC_ANCILLARYDATA_ERROR	8	Error in ancillary data parameters
	AACENC_INVALID_OUTFORMAT	9	Invalid output format
	AACENC_INVALID_BLOCK	10	Invalid block type
	AACENC_DIV_BY_ZERO	11	Divide by zero error
	AACENC_NULL_POINTER	12	Null pointer error
	AACENC_CRC_ERROR	13	Cyclic redundancy check error
	AACENC_TNS_ERROR	14	TNS error
	AACENC_INVALID_SFB	15	Invalid scale factor band
	AACENC_QUANTIZATION_ERROR	16	Quantization error
	AACENC_PSY_THRESHOLD_ERROR	17	Psychoacoustic threshold calculation error
	AACENC_DYNAMICBITCOUNT_ERROR	18	Dynamic bit count calculation error
	AACENC_WRITEADTSHEADER_ERROR	19	Error while writing ADTS header
	AACENC_WRITEADIFHEADER_ERROR	20	Error while writing ADIF header
	AACENC_OUTBUF_TOO_SMALL	21	Number of output bytes is greater than output buffer size
	AACENC_ERROR	22	Unspecified error

The following errors are fatal errors and the application has to reset the encoder with correct dynamic parameter values.

- AACENC_WRITEBITSTREAM_ERROR
- AACENC_DIV_BY_ZERO
- AACENC_NULL_POINTER
- AACENC_ERROR
- AACENC_INVALID_BLOCK
- AACENC_DIV_BY_ZERO
- AACENC_TNS_ERROR
- AACENC_INVALID_SFB
- AACENC_QUANTIZATION_ERROR
- AACENC_PSY_THRESHOLD_ERROR
- AACENC_DYNAMICBITCOUNT_ERROR
- AACENC_WRITEADTSHEADER_ERROR
- AACENC_WRITEADIFHEADER_ERROR

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`
- `IAUDENC_Fxns`
- `IAUDENC_Params`
- `IAUDENC_DynamicParams`
- `IAUDENC_InArgs`
- `IAUDENC_Status`
- `IAUDENC_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 AAC Encoder, 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 input PCM samples is present in the input buffer. The input buffer size (in words) is 1024 samples per channel.
- The output buffer size (in bytes) for worst case is 1536 bytes.

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

4.2.1.3 IAUDENC_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 IAUDENC_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.
encodingPreset	XDAS_Int32	Input	❑ XDM_DEFAULT - Default setting of encoder ❑ XDM_HIGH_QUALITY - High quality encoding ❑ XDM_HIGH_SPEED - High speed encoding ❑ XDM_USER_DEFINED - User defined configuration See XDM_EncodingPreset enumeration for details.
maxSampleRate	XDAS_Int32	Input	Maximum sampling frequency in Hertz.
maxBitrate	XDAS_Int32	Input	Maximum bit rate in bits per second.
maxNoOfCh	XDAS_Int32	Input	Maximum channels. See IAUDIO_ChannelId enumeration for details.
dataEndianness	XDAS_Int32	Input	Endianness of output data. See XDM_DataFormat enumeration for details.

Note:

- Currently, the AAC Encoder implementation supports XDM_LE_16 and XDM_LE_32 format.
- For the supported maxBitrate and maxSampleRate values, see the standard documents listed in the Related Documentation section.
- The supported maxBitrate is 576 kbps.
- The supported maxSampleRate is 96 kHz.
- Supports a maximum of two input channels.
- encodingPreset is not supported in this version of AAC Encoder. The value of encodingPreset is ignored by the encoder

4.2.1.5 IAUDENC_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.
`inputFormat`	`XDAS_Int32`	Input	Input PCM format. See `IAUDIO_PcmFormat` enumeration for details.
`bitRate`	`XDAS_Int32`	Input	Average bit rate in bits per second.
`sampleRate`	`XDAS_Int32`	Input	Sampling frequency in Hertz.
`numChannels`	`XDAS_Int32`	Input	Number of channels. See `IAUDIO_ChannelId` enumeration for details.
`numLFEChannels`	`XDAS_Int32`	Input	Number of LFE channels in the stream.
`inputBitsPerSample`	`XDAS_Int32`	Input	Number of bits per input PCM Sample

Note:

- Currently, the AAC Encoder does not support change in bit rate, sample rate, and number of channels in between frames. All these should be set at the start of encoding using `IAACENC_Params` structure or by using the reset command.
- Currently this encoder supports 16-bit per input sample and 32-bit per input sample. In case of 32-bits per input sample only the MSB 16-bits are considered
- `numLFEChannels` is ignored by the encoder

4.2.1.6 IAUDENC_InArgs

Description

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

Fields

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

4.2.1.7 IAUDENC_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.
frameLen	XDAS_Int32	Output	Number of samples encoded per encode call.
bufInfo	XDM_AlgBufInfo	Output	Input and output buffer information. See XDM_AlgBufInfo data structure for details.

4.2.1.8 IAUDENC_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 enumeration for details.
bytesGenerated	XDAS_Int32	Output	Bytes generated during the process call.

4.2.2 AAC Encoder Data Structures

This section includes the following AAC Encoder specific extended data structures:

- IAACENC_Params
- IAACENC_DynamicParams
- IAACENC_InArgs
- IAACENC_Status
- IAACENC_OutArgs

4.2.2.1 IAACENC_Params

Description

This structure defines the creation parameters and any other implementation specific parameters for the AAC Encoder instance object. The creation parameters are defined in the table below.

Fields

Field	Datatype	Input/ Output	Description
audenc_params	IAUDENC_Params	Input	See IAUDENC_Params data structure for details.
outObjectType	AACENC_OBJ_TYP	Input	Output object type LC, HE or HEv2. See AACENC_OBJ_TYP enumeration for details.
outFileFormat	AACENC_TRANSPORT_TYPE	Input	Output file format. See AACENC_TRANSPORT_TYPE enumeration for details.
useCRC	AACENC_BOOL_TYPE	Input	Flag for inserting CRC bits. See AACENC_BOOL_TYPE enumeration for details.
useTns	AACENC_BOOL_TYPE	Input	Flag for activating TNS feature. See AACENC_BOOL_TYPE enumeration for details.
usePns	AACENC_BOOL_TYPE	Input	Flag for activating PNS feature. See AACENC_BOOL_TYPE enumeration for details.
downMixFlag	AACENC_BOOL_TYPE	Input	Flag for enabling down mixing of channels. See AACENC_BOOL_TYPE enumeration for details.
bitRateMode	AACENC_BITRATE_MODE	Input	Flag for indicating CBR and VBR modes
bitRate	XDAS_Int32	Input	Input bit rate in bits per second.
sampleRate	XDAS_Int32	Input	Input sampling rate in Hertz.

Field	Datatype	Input/ Output	Description
nbInChannels	XDAS_Int32	Input	Enumerated type for number of input channels: ❑ IAUDIO_MONO - Single Channel ❑ IAUDIO_STEREO - Two Channels
dualMono	AACENC_BOOL_TYPE	Input	Flag to indicate dual mono stream. See AACENC_BOOL_TYPE enumeration for details.
ancFlag	AACENC_BOOL_TYPE	Input	Ancillary data flag. See AACENC_BOOL_TYPE enumeration for details.
ancRate	XDAS_Int32	Input	Ancillary data rate.

Note:

- outFileFormat supports Audio Data Interchange Format (ADIF), Audio Data Transport Stream (ADTS), and raw output format.
- In case of VBR mode,

  VBR Mode 1 => Low quality

  VBR Mode 5 =>Very high quality

  The quality and the bit-rate increases from VBR Mode 1 to VBR Mode 5.
- ancRate should be less than or equal to 15% of bitRate, subject to an absolute maximum value of 19199.

4.2.2.2 IAACENC_DynamicParams

‖ Description

This structure defines the run time parameters and any other implementation specific parameters for the AAC Encoder instance object. The run time parameters are defined in the table below.

‖ Fields

Field	Datatype	Input/ Output	Description
audenc_dynamicparams	IAUDENC_DynamicParams	Input	See IAUDENC_DynamicParams data structure for details.
dualMono	AACENC_BOOL_TYPE	Input	Flag to indicate dual Mono input. See AACENC_BOOL_TYPE enumeration for details.
useTns	AACENC_BOOL_TYPE	Input	Flag for activating TNS feature. See AACENC_BOOL_TYPE enumeration for details.

Field	Datatype	Input/ Output	Description
usePns	AACENC_BOOL_TYPE	Input	Flag for activating PNS feature. See AACENC_BOOL_TYPE enumeration for details.
useCRC	AACENC_BOOL_TYPE	Input	Flag for inserting CRC bits. See AACENC_BOOL_TYPE enumeration for details.
downMixFlag	AACENC_BOOL_TYPE	Input	Flag for enabling down sampling. See AACENC_BOOL_TYPE enumeration for details.
outObjectType	AACENC_OBJ_TYP	Input	Output object type LC/HE/HEv2. See AACENC_OBJ_TYP enumeration for details.
outFileFormat	AACENC_TRANSPORT_TYPE	Input	Output file format. See AACENC_TRANSPORT_TYPE enumeration for details.
ancFlag	AACENC_BOOL_TYPE	Input	Ancillary data flag. See AACENC_BOOL_TYPE enumeration for details.
ancRate	XDAS_Int32	Input	Ancillary data rate.

Note:

ancRate should be less than or equal to 15% of bitRate, subject to an absolute maximum value of 19199.

4.2.2.3 IAACENC_InArgs

‖ Description

This structure defines the run time input arguments for the AAC Encoder instance object.

‖ Fields

Field	Datatype	Input/ Output	Description
audenc_inArgs	IAUDENC_InArgs	Input	See IAUDENC_InArgs data structure for details.
numInSamples	XDAS_Int32	Input	Number of input PCM samples per channel.
*ancData	XDAS_UInt8	Input	Pointer to ancillary data.
numAncBytes	XDAS_Int32	Input	Number of ancillary bytes.

4.2.2.4 IAACENC_Status

Description

This structure defines parameters that describe the status of the AAC Encoder and any other implementation specific parameters. The status parameters are defined in the table below.

Fields

Field	Datatype	Input/ Output	Description
audenc_status	IAUDENC_Status	Output	See IAUDENC_Status data structure for details.
outputObjectType	AACENC_OBJ_TYP	Output	Output object type LC, HE or HEv2. See AACENC_OBJ_TYP enumeration for details.
bitRate	XDAS_Int32	Output	Output bit rate of the AAC stream.
outFileFormat	AACENC_TRANSPORT_TYPE	Output	Output file format. See AACENC_TRANSPORT_TYPE enumeration for details.
isValid	AACENC_BOOL_TYPE	Output	Flag to indicate status validity. See AACENC_BOOL_TYPE enumeration for details.

4.2.2.5 IAACENC_OutArgs

Description

This structure defines the run time output arguments for the AAC Encoder instance object.

Fields

Field	Datatype	Input/ Output	Description
audenc_outArgs	IAUDENC_OutArgs	Output	See IAUDENC_OutArgs data structure for details.

4.3 Interface Functions

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

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

You must call these APIs in the following sequence:

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

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

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

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) (IAUDENC_Handle handle, IAUDENC_Cmd
id, IAUDENC_DynamicParams *params, IAUDENC_Status
*status);
```

‖ Arguments

```
IAUDENC_Handle handle; /* algorithm instance handle */

IAUDENC_Cmd id; /* algorithm specific control commands*/

IAUDENC_DynamicParams *params /* algorithm run time
parameters */

IAUDENC_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 `IAUDENC_DynamicParams` and `IAUDENC_Status` data structures respectively.

Note:

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

‖ Preconditions

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

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

‖ Postconditions

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

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

‖ Example

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

‖ See Also

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

> **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)(IAUDENC_Handle handle, XDM_BufDesc
*inBufs, XDM_BufDesc *outBufs, IAUDENC_InArgs *inargs,
IAUDENC_OutArgs *outargs);
```

‖ Arguments

```
IAUDENC_Handle handle; /* algorithm instance handle */

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

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

IAUDENC_InArgs *inargs /* algorithm runtime input
arguments */

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

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

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

‖ Preconditions

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

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

‖ Postconditions

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

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

‖ Example

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

‖ See Also

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

Note:

- Audio codecs do not use `algActivate()`, `algDeactivate()`, and `DMAN3_init()` APIs.
- AAC Encoder supports 16-bit and 32-bit PCM samples in litte endian format as input.

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

AAC Encoder Bit-rate and Sampling Frequency Combination

The following table contains information on AAC Encoder bit rate and sampling frequency combination.

Sampling frequency	Mono/Stereo	Min-Bit rate	Max-Bit rate
8000	Mono	8000	42000
8000	Stereo	16000	84000
16000	Mono	8000	84000
16000	Stereo	16000	168000
22050	Mono	8000	116000
22050	Stereo	16000	232000
32000	Mono	8000	160000
32000	Stereo	16000	320000
44100	Mono	8000	160000
44100	Stereo	16000	320000
48000	Mono	8000	288000
48000	Stereo	16000	576000
96000	Mono	16000	288000
96000	Stereo	20000	576000