WMA Version8 Encoder on C64x+

User Guide

Literature Number: SPRUF48
February 2008

TEXAS INSTRUMENTS

Preface

Read This First

About This Manual

This document describes how to install and work with Texas Instruments' (TI) WMA Version8 Encoder implementation on the C64x+ based SoCs. It also provides a detailed Application Programming Interface (API) reference and information on the sample application that accompanies this component.

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

Intended Audience

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

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

How to Use This Manual

This document includes the following chapters:

- **Chapter 1 - Introduction**, provides a brief introduction to the XDAIS and XDM standards. It also provides an overview of the codec and lists its supported features.
- **Chapter 2 - Installation Overview**, describes how to install, build, and run the codec.
- **Chapter 3 - Sample Usage**, describes the sample usage of the codec.
- **Chapter 4 - API Reference**, describes the data structures and interface functions used in the codec.
- **Appendix A - WMA Encoder Bit-rate and Sampling Frequency Combination,** contains the WMA Encoder bit-rate and sampling frequency combination.

- **Appendix B - Application Flow,** describes how the application calls WMA Encoder.

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+ Mega module* (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 Mega module Reference Guide* (literature number SPRU871) describes the C64x+ mega module peripherals.
- *TMS320C64x to TMS320C64x+ CPU Migration Guide* (literature number SPRAA84) describes migration from the Texas Instruments

TMS320C64x™ digital signal processor (DSP) to the TMS320C64x+™ DSP.

- *TMS320C6000 Optimizing Compiler v 6.0 Beta User's Guide* (literature number SPRU187N) explains how to use compiler tools such as compiler, assembly optimizer, standalone simulator, library-build utility, and C++ name demangler.
- *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide* (literature number SPRU732) describes the CPU architecture, pipeline, instruction set, and interrupts of the C64x and C64x+ DSPs.
- *TMS320DM6446 Digital Media System-on-Chip* (literature number SPRS283).
- *TMS320DM6446 Digital Media System-on-Chip Errata (Silicon Revision 1.0)* (literature number SPRZ241) describes the known exceptions to the functional specifications for the TMS320DM6446 Digital Media System-on-Chip (DMSoC).
- *TMS320DM6443 Digital Media System-on-Chip* (literature number SPRS282).
- *TMS320DM6443 Digital Media System-on-Chip Errata (Silicon Revision 1.0)* (literature number SPRZ240) describes the known exceptions to the functional specifications for the TMS320DM6443 Digital Media System-on-Chip (DMSoC).
- *TMS320DM644x DMSoC DSP Subsystem Reference Guide* (literature number SPRUE15) describes the digital signal processor (DSP) subsystem in the TMS320DM644x Digital Media System-on-Chip (DMSoC).
- *TMS320DM644x DMSoC ARM Subsystem Reference Guide* (literature number SPRUE14) describes the ARM subsystem in the TMS320DM644x Digital Media System on a Chip (DMSoC).
- *DaVinci Technology - Digital Video Innovation Product Bulletin (Rev. A)* (literature number SPRT378A).
- *The DaVinci Effect: Achieving Digital Video Without Complexity White Paper* (literature number SPRY079).
- *DaVinci Benchmarks Product Bulletin* (literature number SPRT379)
- *DaVinci Technology for Digital Video White Paper* (literature number SPRY067).
- *The Future of Digital Video White Paper* (literature number SPRY066).

Abbreviations

The following abbreviations are used in this document:

Table 1-1. List of Abbreviations

Abbreviation	Description
API	Application Programming Interface
ASF	Advanced Systems Format
CBR	Constant Bit Rate
DRM	Digital Rights Management
EVM	Evaluation Module
RCA	Raw Compressed Audio
WMA	Windows Media Audio
XDAIS	eXpressDSP Algorithm Interface Standard
XDM	eXpressDSP Digital Media

Text Conventions

The following conventions are used in this document:

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

Product Support

When contacting TI for support on this codec, quote the product name (WMA Version8 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

Code Composer Studio, the DAVINCI Logo, DAVINCI, DSP/BIOS, eXpressDSP, TMS320, TMS320C64x, TMS320C6000, TMS320DM644x, TNETV2685 and TMS320C64x+ are trademarks of Texas Instruments.

All trademarks are the property of their respective owners.

Contents

Read This First iii
About This Manual iii
Intended Audience iii
How to Use This Manual iii
Related Documentation From Texas Instruments iv
Abbreviations vi
Text Conventions vi
Product Support vi
Trademarks vi
Contents vii
Figures ix
Tables xi
Introduction 1-1
1.1 Overview of XDAIS and XDM 1-2
1.1.1 XDAIS Overview 1-2
1.1.2 XDM Overview 1-2
1.2 Overview of WMA Version8 Encoder 1-4
1.3 Supported Services and Features 1-4
Installation Overview 2-1
2.1 System Requirements 2-2
2.1.1 Hardware 2-2
2.1.2 Software 2-2
2.2 Installing the Component 2-2
2.3 Before Building the Sample Test Application 2-4
2.3.1 Installing DSP/BIOS 2-4
2.4 Building and Running the Sample Test Application 2-4
2.5 Configuration Files 2-5
2.5.1 Generic Configuration File 2-5
2.5.2 Encoder Configuration File 2-6
2.6 Standards Conformance and User-Defined Inputs 2-6
2.7 Uninstalling the Component 2-7
2.8 Evaluation Version 2-7
Sample Usage 3-1
3.1 Overview of the Test Application 3-2
3.1.1 Parameter Setup 3-3
3.1.2 Algorithm Instance Creation and Initialization 3-3
3.1.3 Process Call 3-4
3.1.4 Algorithm Instance Deletion 3-4
API Reference 4-1
4.1 Symbolic Constants and Enumerated Data Types 4-2
4.2 Data Structures 4-5
4.2.1 Common XDM Data Structures 4-5
4.2.2 WMA Encoder Data Structures 4-11
4.3 Interface Functions 4-14

4.3.1 Creation APIs 4-15
4.3.2 Initialization API...4-16
4.3.3 Control API 4-17
4.3.4 Data Processing API 4-19
4.3.5 Termination API 4-21

WMA Encoder Bit-rate and Sampling Frequency Combination A-1
Application Flow B-1

Figures

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

This page is intentionally left blank

Tables

Table 1-1. List of Abbreviations vi
Table 2-1. Component Directories 3
Table 4-1. List of Enumerated Data Types 2
Table 4-2. WMA Encoder Error Status 4
Table A-1. Sampling Rate and Bit-rate combinations A-1

This page is intentionally left blank

Chapter 1

Introduction

This chapter provides a brief introduction to XDAIS and XDM. It also provides an overview of TI's implementation of the WMA Version8 Encoder on the C64x+ based SoCs and its supported features.

Topic **Page**

Topic	Page
1.1 Overview of XDAIS and XDM	**1-2**
1.2 Overview of WMA Version8 Encoder	**1-4**
1.3 Supported Services and Features	**1-4**

1.1 Overview of XDAIS and XDM

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

1.1.1 XDAIS Overview

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

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

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

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

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

1.1.2 XDM Overview

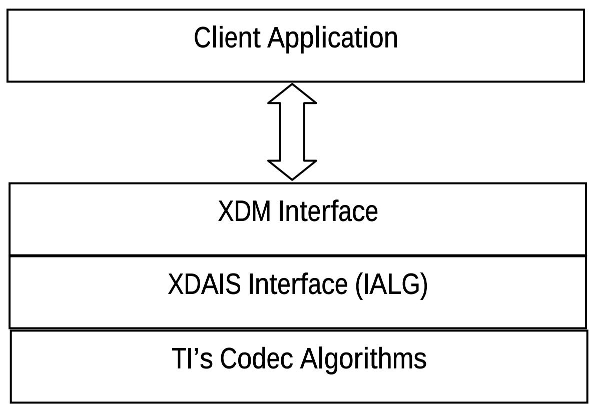
In the multimedia application space, you have the choice of integrating any codec into your multimedia system. For example, if you are building a video decoder system, you can use any of the available video decoders (such as MPEG4, H.263, or H.264) in your system. To enable easy integration with the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

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

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

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

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



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

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

1.2 Overview of WMA Version8 Encoder

The WMA Version8 Encoder is eXpressDSP Digital Media (XDM) compliant Encoder, which encodes and converts the wave files into Windows Media Audio files in the Advanced Systems Format (ASF).

From this point onwards, all references to WMA Encoder means WMA Version8 Encoder only.

1.3 Supported Services and Features

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

This version of the codec has the following supported features:

- Supports 16-bit PCM samples as input
- Supports full implementation, that is, Class 4 type of WMA Encoder
- Supports bit-rates from 5 kbps to 192 kbps
- Supports CBR (Constant Bit-Rate) mode only
- Supports 8 to 48 kHz output-sampling frequencies
- Supports mono and stereo channels
- Supports ASF format
- Compliant with Microsoft Acceptance Test criteria
- 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	2-2
2.2 Installing the Component	2-2
2.3 Before Building the Sample Test Application	2-4
2.4 Building and Running the Sample Test Application	2-4
2.5 Configuration Files	2-5
2.6 Standards Conformance and User-Defined Inputs	2-6
2.7 Uninstalling the Component	2-7
2.8 Evaluation Version	2-7

2.1 System Requirements

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

2.1.1 Hardware

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

This codec is supported on any C64x+ based device.

2.1.2 Software

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

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

2.2 Installing the Component

The codec component is released as a compressed archive. To install the codec, extract the contents of the zip file onto your local hard disk. The zip file extraction creates a directory called 100_A_WMA_E_1_13_00, under which another directory named C64XPLUS_ASF is created. Figure 2-1 shows the sub-directories created in C64XPLUS_ASF.

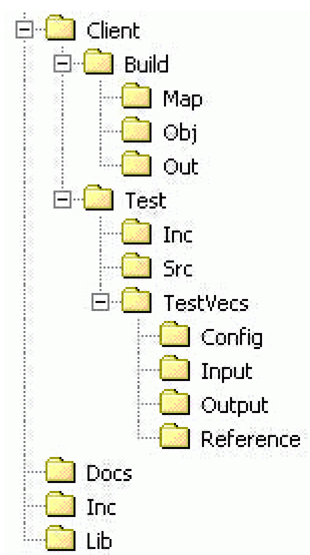


Figure 2-1. Component Directory Structure

> **Note:**
>
> If you are installing an evaluation version of this codec, the parent directory name will be 100E_A_WMA_E_1_13_00.

Table 2-1 provides a description of the sub-directories created in the C64XPLUS_ASF directory.

Table 2-1. Component Directories

Sub-Directory	Description
\Inc	Contains XDM related header files which allow interface to the codec library
\Lib	Contains the codec library file
\Docs	Contains user guide and datasheet
\Client\Build	Contains the sample test application project (.pjt) file
\Client\Build\Map	Contains the memory map generated on compilation of the code
\Client\Build\Obj	Contains the intermediate .asm and/or .obj file generated on compilation of the code
\Client\Build\Out	Contains the final application executable (.out) file generated by the sample test application
\Client\Test\Src	Contains application C files
\Client\Test\Inc	Contains header files needed for the application code
\Client\Test\TestVecs\Input	Contains input test vectors
\Client\Test\TestVecs\Output	Contains output generated by the codec
\Client\Test\TestVecs\Reference	Contains read-only reference output 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 wma8enc_tii_asf.l64P exists in the \Lib sub-directory.

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

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

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

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

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

7) The application displays one of the following messages either for each sub frame or after completion of output:

   - "Encoder compliance test failed in frame num = # at sample = # " (for compliance check failure mode)
   - "Encoder compliance test passed" (for compliance check pass mode)
   - "Encoder output dump completed" (for output dump mode)

2.5 Configuration Files

This codec is shipped along with:

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

2.5.1 Generic Configuration File

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

The format of the Testvecs.cfg file is:

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

where:

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

A sample Testvecs.cfg file is as shown.

```
1
..\\Test\\TestVecs\\Config\\Testparams.cfg
..\\Test\\TestVecs\\Input\\test3_44kHz_short.wav
..\\Test\\TestVecs\\Reference\\test3_short_v8_192kbps_44kHz_S.wma
0
..\\Test\\TestVecs\\Config\\Testparams.cfg
..\\Test\\TestVecs\\Input\\test3_44kHz_short.wav
..\\Test\\TestVecs\\Output\\test3_short_v8_192kbps_44kHz_S.wma
```

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
###########################################################
Parameters
###########################################################
192000 # Encoded output Bit rate
```

Any field in the `IAUDENC_Params` structure (see Section 4.2.1.4) can be set in the Testparams.cfg file using the syntax shown above. If you specify additional fields in the Testparams.cfg file, ensure to modify the test application appropriately to handle these fields.

2.6 Standards Conformance and User-Defined Inputs

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

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

- Copy the input files to the \Client\Test\TestVecs\Inputs sub-directory.
- Copy the reference files to the \Client\Test\TestVecs\Reference sub-directory.
- Edit the configuration file, Testvecs.cfg available in the \Client\Test\TestVecs\Config sub-directory. For details on the format of the Testvecs.cfg file, see Section 2.5.1.
- Execute the sample test application. On successful completion, the application displays one of the following messages for each frame:
  - "Encoder compliance test passed/failed" (if `X` is 1)
  - "Encoder output dump completed" (if `X` is 0)

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

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.

This page is intentionally left blank

Chapter 3

Sample Usage

This chapter provides a detailed description of the sample test application that accompanies this codec component.

3.1 Overview of the Test Application

The test application exercises the `IAUDENC` base class of the WMA 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.

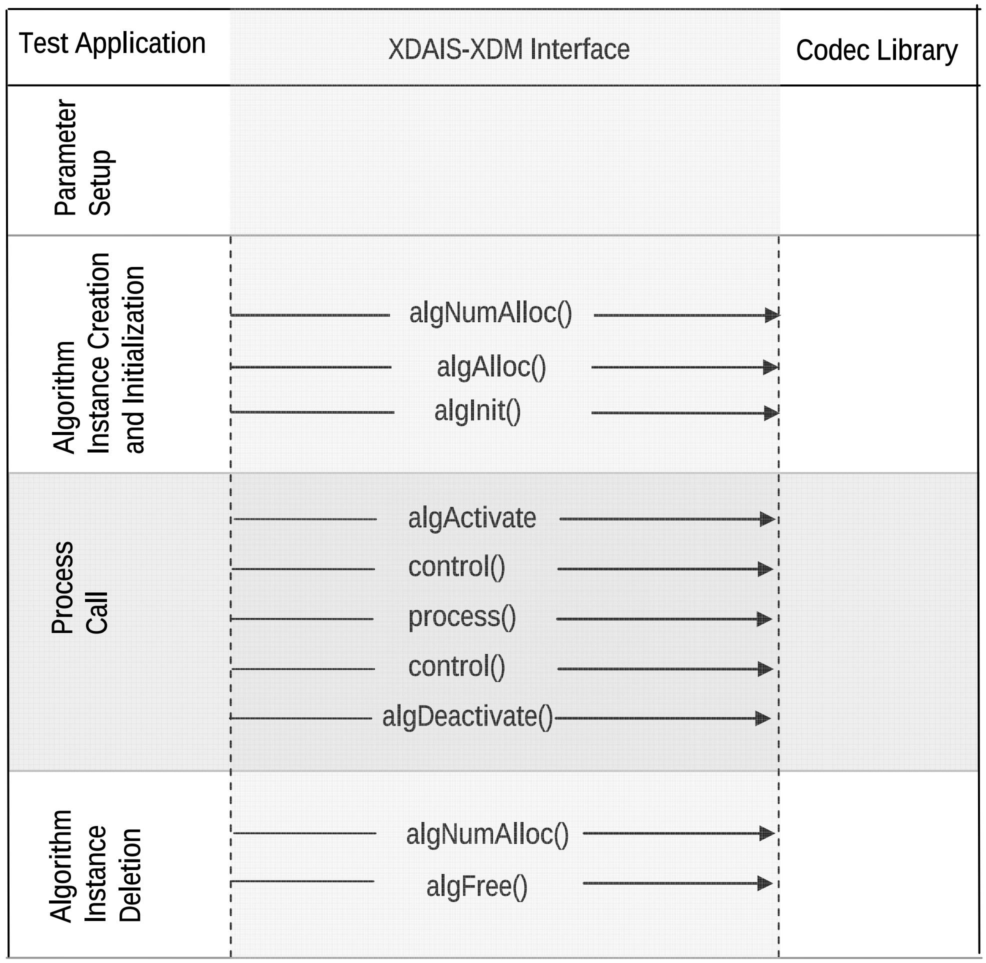


Figure 3-1. Test Application Sample Implementation

> **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, and so on. The test application obtains the required parameters from the Encoder configuration files.

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

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

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

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

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

4) 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 `IAUDENC_InArgs` and `IAUDENC_OutArgs` structures.

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

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

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

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

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

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

3.1.4 Algorithm Instance Deletion

Once encoding/decoding is complete, the test application 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.

Topic	Page
4.1 Symbolic Constants and Enumerated Data Types	4-2
4.2 Data Structures	4-5
4.3 Interface Functions	4-14

4.1 Symbolic Constants and Enumerated Data Types

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

Table 4-1. List of Enumerated Data Types

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
IAUDIO_ChannelId	IAUDIO_MONO	Single channel
	IAUDIO_STEREO	Two channel
	IAUDIO_THREE_ZERO	Three channel. Not supported by WMA standard Encoder.
	IAUDIO_FIVE_ZERO	Five channel. Not supported by WMA standard Encoder.
	IAUDIO_FIVE_ONE	5.1 channels. Not supported by WMA standard Encoder.
	IAUDIO_SEVEN_ONE	7.1 channel. Not supported by WMA standard Encoder.
IAUDIO_PcmFormat	IAUDIO_BLOCK	Left channel data followed by right channel data
	IAUDIO_INTERLEAVED	Left and right channel data interleaved
XDM_DataFormat	XDM_BYTE	Big endian stream
	XDM_LE_16	16-bit little endian stream
	XDM_LE_32	32-bit little endian stream
XDM_EncodingPreset	XDM_DEFAULT	Default setting of Encoder
	XDM_HIGH_QUALITY	High quality encoding
	XDM_HIGH_SPEED	High speed encoding
	XDM_USER_DEFINED	User defined configuration using advanced parameters
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. Just returns IALG_EOK.
	XDM_GETBUFINFO	Query algorithm instance regarding the properties of input and output buffers
	IWMAENC_GETSTATUS	Same as XDM_GETSTATUS
	IWMAENC_SETPARAMS	Same as XDM_SETPARAMS
	IWMAENC_RESET	Same as XDM_RESET
	IWMAENC_SETDEFAULT	Same as XDM_SETDEFAULT
	IWMAENC_FLUSH	Same as XDM_FLUSH
	IWMAENC_GETBUFINFO	Same as XDM_GETBUFINFO
	IWMAENC_CLOSE	Query algorithm to close the algorithm instance
	IWMAENC_ENCODE	Query algorithm instance to Encode the input data
XDM_ErrorBit		The bit fields in the 32-bit error code are interpreted as shown
	XDM_APPLIEDCONCEALMENT	Bit 9 ❑ 1 - Applied concealment ❑ 0 - Ignore
	XDM_INSUFFICIENTDATA	Bit 10 ❑ 1 - Insufficient input data ❑ 0 - Ignore
	XDM_CORRUPTEDDATA	Bit 11 ❑ 1 - Invalid data ❑ 0 - Ignore
	XDM_CORRUPTEDHEADER	Bit 12 ❑ 1 - Corrupted frame header ❑ 0 - Ignore
	XDM_UNSUPPORTEDINPUT	Bit 13 ❑ 1 - Unsupported feature/parameter ❑ 0 - Ignore

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	`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 WMA 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. WMA Encoder Error Status

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
`tWMFEncodeStatus`	`WMA_Succeeded`	0	Successful encoding.
	`WMA_EncPacketsDone`	1	This version of Encoder does not return this error.
	`WMA_BadOutputFile`	2	This version of Encoder does not return this error.
	`WMA_Failed`	3	Memory allocation failures, or invalid instance structure parameters value.
	`WMA_BadMemory`	4	Memory initialization failures.
	`WMA_EncodeFailed`	5	This version of Encoder does not return this error.
	`WMA_UnSupportedInputFormat`	6	The input parameters provided for encoding is not supported by the Encoder.

Group or Enumeration Class	Symbolic Constant Name	Value	Description or Evaluation
	WMA_UnSupportedCompressedFormat	7	This version of Encoder does not return this error.
	WMA_InValidArguments	8	Invalid pointers passed as arguments
	WMA_BadSource	9	This version of Encoder does not return this error.
	WMA_UnsupportedSamplingRate	10	The required sampling rate for encoding is not supported by the Encoder.
	WMA_InitFailed	11	Encoder Initialization failure
	WMA_UnSupportedParams	12	One or more of the input parameters supplied to the Encoder are not supported by this version.
	WMA_InsufficientData	13	Number of samples supplied to the Encoder is less than required number of samples per call.

4.2 Data Structures

This section describes the XDM defined data structures that are common across all 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 WMA Encoder, the buffer details are:

- Maximum number of input buffer required is 1
- Maximum number of output buffer required is 1
- Maximum input buffer size is 16384 bytes and maximum output buffer size is 10000 bytes
- Output buffer address needs to be 16-bit aligned

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 not sure 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. Not used by the present WMA Encoder library.
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:

- The `dataEndianness` field should be set to `XDM_LE_16`.
- The maximum sampling rate supported is 48000 Hz.
- The maximum bitrate supported is 192 kbps.
- The maximum channels supported is stereo.
- The `encodingPreset` field is not used by the present WMA Encoder library.
- The `maxBitrate`, `maxSampleRate` and, `maxNoOfCh` fields are used by the application only during intialization to check the capability of the Encoder. During actual encoding the WMA Encoder will Encode all the sampling frequencies, bit-rate and channels as mentioned in this user manual.

4.2.1.5 IAUDENC_DynamicParams

|| Description

This structure defines the run-time parameters for an algorithm instance object. Any change in these parameter values need to be done after RESET command. Set this data structure to NULL, if you are not sure of the values to be specified for these parameters.

|| Fields

Field	Datatype	Input/ Output	Description
size	XDAS_Int32	Input	Size of the basic or extended (if being used) data structure in bytes
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 input bits per output sample

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 WMA Encoder Data Structures

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

- `IWMAENC_Params`
- `IWMAENC_DynamicParams`
- `IWMAENC_InArgs`
- `IWMAENC_Status`
- `IWMAENC_OutArgs`

4.2.2.1 IWMAENC_Params

‖ Description

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

‖ Fields

Field	Datatype	Input/ Output	Description
audenc_params	IAUDENC_Params	Input	See IAUDENC_Params data structure for details.
fContentDescriptorPresent	XDAS_Int32	Input	❑ 1- indicates that content descriptors are present. ❑ 0 - indicates that content descriptors are absent
szContents	XDAS_Uint16 **	Input	Pointer array to content descriptors
srcChannelPattern	XDAS_Int32	Input	Channel pattern interleaved or non interleaved
pktDuration	XDAS_Uint16	Input	Required encoded WMA packet duration in milliseconds
samplingRate	XDAS_Int32	Input	Sampling rate for encoding
noOfChannels	XDAS_Int32	Input	Number of channels encoded
bitRate	XDAS_Int32	Input	Bit rate at which encoding is done
robustPadData	XDAS_Int32	Input	When set to 1 inserts a predefined pad data in the encoded bit stream. For any other value the pad data is not inserted in the encoded bit stream.

> **Note:**
>
> When `robustPadData` is 1, the Encoder pads extra bytes to the bit stream, and the conformance should not be checked with `robustPadData` equal to 1.

4.2.2.2 IWMAENC_DynamicParams

|| Description

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

|| Fields

Field	Datatype	Input/ Output	Description
`audenc_dynamicparams`	`IAUDENC_DynamicParams`	Input	Codec parameters that can be modified after creation via control calls. See `IAUDENC_DynamicParams` data structure for details.

4.2.2.3 IWMAENC_InArgs

|| Description

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

|| Fields

Field	Datatype	Input/ Output	Description
`audenc_inArgs`	`IAUDENC_InArgs`	Input	Run-time input arguments for process function. See `IAUDENC_InArgs` data structure for details.
`numSamples`	`XDAS_Int32`	Input	Input samples supplied per call

4.2.2.4 IWMAENC_Status

|| Description

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

|| Fields

Field	Datatype	Input/ Output	Description
`audenc_status`	`IAUDENC_Status`	Output	See `IAUDENC_Status` data structure for details.

4.2.2.5 IWMAENC_OutArgs

|| Description

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

|| Fields

Field	Datatype	Input/ Output	Description
`audenc_outArgs`	`IAUDENC_OutArgs`	Output	See `IAUDENC_OutArgs` data structure for details.
`ip_bytes_consumed`	`XDAS_Int16`	Output	Number of total bytes consumed by the Encoder from the input buffer

4.3 Interface Functions

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

- Pre-emption can happen only at frame boundaries and after `algDeactivate()` is called.
- Audio codecs do not use `algActivate(), algDeactivate()`, and `DMAN3_init()` APIs.

4.3.5 Termination API

Termination API is used to terminate the algorithm instance and free up the memory space that it uses.

‖ Name

`algFree()` - determine the addresses of all memory buffers used by the algorithm

‖ Synopsis

```
XDAS_Int32 algFree(IALG_Handle handle, IALG_MemRec
memTab[]);
```

‖ Arguments

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

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

‖ Return Value

```
XDAS_Int32; /* Number of buffers used by the algorithm */
```

‖ Description

`algFree()` determines the addresses of all memory buffers used by the algorithm. The primary aim of doing so is to free up these memory regions after closing an instance of the algorithm.

The first argument to `algFree()` is a handle to the algorithm instance.

The second argument is a table of memory records that describe the base address, size, alignment, type, and memory space of all buffers previously allocated for the algorithm instance.

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

‖ See Also

`algAlloc()`

Appendix A

WMA Encoder Bit-rate and Sampling Frequency Combination

The Encoder as per WMA standard supports the following bit-rate and sampling rate combinations.

Table A-1. Sampling Rate and Bit-rate combinations

Combination	Sampling Frequency	Channel	BitRate
1	48000	2	192000
2	48000	2	160000
3	48000	2	128000
1	44100	2	192000
2	44100	2	160000
3	44100	2	128000
4	44100	2	96000
5	44100	2	80000
6	44100	2	64000
7	44100	2	48000
8	44100	2	32000
9	44100	1	48000
10	44100	1	32000
1	32000	2	48000
2	32000	2	40000
3	32000	2	32000
4	32000	1	20000
1	22050	2	32000
2	22050	2	22000
3	22050	2	20000
4	22050	1	20000
5	22050	1	16000
1	16000	2	20000
2	16000	2	16000
3	16000	1	16000
4	16000	1	12000
5	16000	1	10000
1	11025	1	10000
2	11025	1	8000

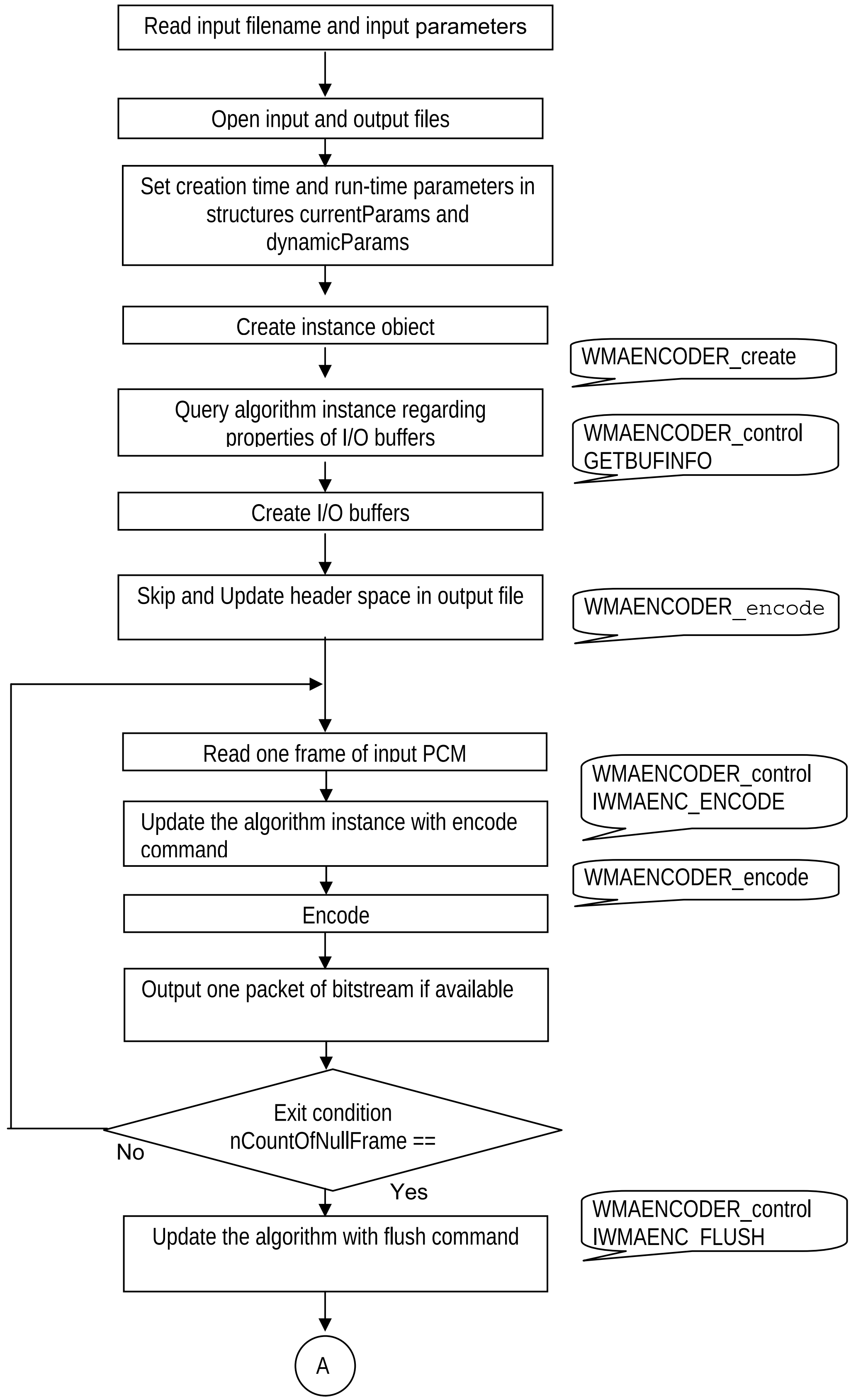
Combination	Sampling Frequency	Channel	BitRate
1	8000	2	12000
2	8000	1	8000
3	8000	1	6000
4	8000	1	5000

Appendix B

Application Flow

This section provides information on how the application should call the WMA Encoder

The following process flow diagram explains how application calls the WMA Encoder.



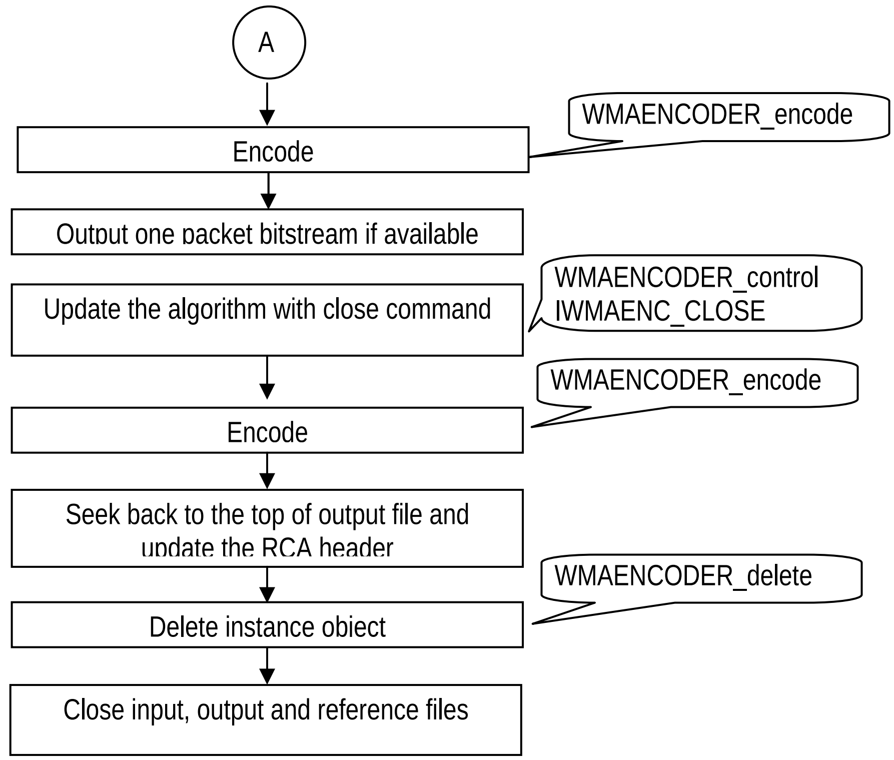


Figure B-1. Application Flow