G.711 Encoder/Decoder on C64x+

User's Guide

Literature Number: SPRUEC9B
January 2009

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Read This First

About This Manual

This document describes how to install and work with Texas Instruments' (TI) G.711 Encoder/Decoder implementation on the C64x+ platform. It also provides a detailed Application Programming Interface (API) reference and information on the sample application that accompanies this component.

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

Intended Audience

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

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

How to Use This Manual

This document includes the following chapters:

- **Chapter 1 - Introduction**, provides a brief introduction to the XDAIS and XDM standards. It also provides an overview of the codec and lists its supported features.
- **Chapter 2 - Installation Overview**, describes how to install, build, and run the codec.
- **Chapter 3 - Sample Usage**, describes the sample usage of the codec.
- **Chapter 4 - API Reference**, describes the data structures and interface functions used in the codec.
- **Appendix A - Memory Placement Details,** describes the memory placement details for G.711.

- **Appendix B - Optimal Function Placement Order for Minimizing Instruction Cache Penalty**, describes optimal function placement order for minimizing cache penalty due to instruction cache misses.

- **Appendix C – Revision History**, highlights the changes made to SPRUEC9A codec specific user guide to make it SPRUEC9B.

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.
- *DaVinci Technology - Digital Video Innovation Product Bulletin (Rev. A)* (sprt378a.pdf)
- *The DaVinci Effect: Achieving Digital Video Without Complexity White Paper* (spry079.pdf)
- *DaVinci Benchmarks Product Bulletin* (sprt379.pdf)
- *DaVinci Technology for Digital Video White Paper* (spry067.pdf)
- *The Future of Digital Video White Paper* (spry066.pdf)

Related Documentation

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

- *ITU-T Recommendation G.711:Pulse code modulation (PCM) of voice frequencies*

Abbreviations

The following abbreviations are used in this document:

Table 1-1. List of Abbreviations

Abbreviation	Description
API	Application Programming Interface
EVM	Evaluation Module
Kbps	Kilo bits per second
PCM	Pulse Code Modulation
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 (G.711 Encoder/Decoder on C64x+) and version number. The version number of the codec is included in the Title of the Release Notes that accompanies this codec.

Trademarks

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

All trademarks are the property of their respective owners.

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 G.711 Encoder/Decoder on the C64x+ platform and its supported features.

1.1 Overview of XDAIS and XDM

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

1.1.1 XDAIS Overview

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

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

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

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

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

1.1.2 XDM Overview

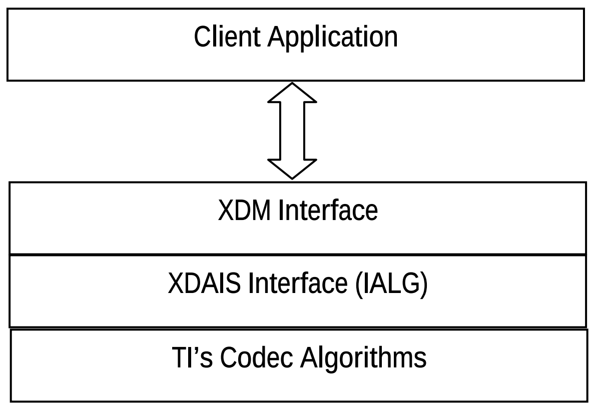
In the multimedia application space, you have the choice of integrating any codec into your multimedia system. For example, if you are building a video decoder system, you can use any of the available video decoders (such as MPEG4, H.263, or H.264) in your system. To enable easy integration with the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

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

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

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

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



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

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

1.2 Overview of G.711 Encoder/Decoder

G.711 is one of the earliest speech coders that convert 16-bit linear PCM samples to 8-bit compressed A-law or U-law samples to give a 64Kbps data rate in the encoder. Decoder expands 64Kbps bit-stream into linear PCM samples of 16-bits each at 8 Khz.

1.3 Supported Services and Features

This user guide accompanies TI's implementation of G.711 Encoder/Decoder on the C64x+ platform. This version of the codec has the following supported features:

- Supports both A-law and U-law compression (encoding) and decompression (decoding)
- Operates on sets of 8 samples
- Supports little endian mode of operation
- eXpressDSP Digital Media (XDM1.0 ISPHDEC1 and XDM1.0 ISPHENC1) 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.

2.1 System Requirements

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

2.1.1 Hardware

This codec has been built and tested on the DM644x EVM.

This codec can be used on any of TI's C64x+ based platforms such as DM644x, DM648, DM643x, DM646x, OMAP35xx and their derivatives.

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 3.3.38.2.
- **Code Generation Tools:** This project is compiled, assembled, archived, and linked using the code generation tools version 6.0.15.
- **Cygnus Cygwin V.B20**, which includes unix utilities as well as the 'make' program.

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_S_G711_X_ALL_C64XPLUS_1_12 under which another directory named G711_C64XPLUS is created.

Figure 2-1 shows the sub-directories created in G711_C64XPLUS directory.

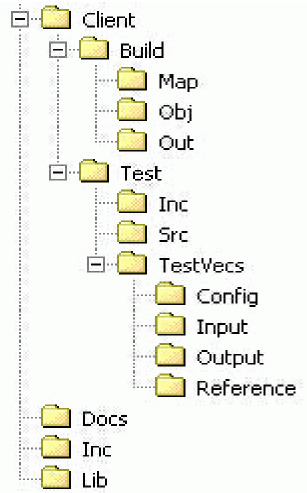


Figure 2-1. Component Directory Structure

Table 2-1 provides a description of the sub-directories created in the G711_C64XPLUS 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. Separate libraries are provided for encoder and decoder.
\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_v3.3

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

- Header file, bcache.h available in the <install directory>\CCStudio_v3.3\<bios_directory>\packages\ti\bios\include directory.
- Library file, biosDM420.a64P available in the <install directory>\CCStudio_v3.3\<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 3.3.38.2 and code generation tools version 6.0.15.

2) Verify that one of the codec object libraries g711dec_tii.l64P or g711enc_tii.l64P exists in the \Lib sub-directory.

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

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

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

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

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

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

   - o “Encoder compliance test passed/failed” or “Decoder compliance test passed/failed” (for compliance check mode)
   - o “Encoder output dump completed” or “Decoder output dump completed” (for output dump mode)

2.5 Configuration Files

This codec is shipped along with:

- ❑ Generic configuration files (Testvecs_enc.cfg and Testvecs_dec.cfg) – specifies input and reference files for the sample test application.
- ❑ Encoder configuration file (Testparams_enc.cfg) – specifies the configuration parameters used by the test application to configure the Encoder.
- ❑ Decoder configuration file (Testparams_dec.cfg) – specifies the configuration parameters used by the test application to configure the Decoder.

2.5.1 Generic Configuration File

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

The format of the Testvecs_enc.cfg and Testvecs_dec.cfg file is:

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

where:

- ❑ `X` may be set as:
   - o 1 - for compliance checking, no output file is created
   - o 0 - for writing the output to the output file
- ❑ `Config` is the Encoder or Decoder configuration file. For details, see Section 2.5.2 and 2.5.3
- ❑ `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_enc.cfg file is as shown:

```
1
..\..\Test\TestVecs\Config\Testparams_enc.cfg
..\..\Test\TestVecs\Input\input_alaw.pcm
..\..\Test\TestVecs\Reference\alaw.bit
0
..\..\Test\TestVecs\Config\Testparams_enc.cfg
..\..\Test\TestVecs\Input\input_alaw.pcm
..\..\Test\TestVecs\Output\alaw.bit
```

A sample Testvecs_dec.cfg file is as shown:

```
1
..\..\Test\TestVecs\Config\Testparams_dec.cfg
..\..\Test\TestVecs\Input\alaw.bit
..\..\Test\TestVecs\Reference\output_alaw.pcm
0
..\..\Test\TestVecs\Config\Testparams_dec.cfg
..\..\Test\TestVecs\Input\alaw.bit
..\..\Test\TestVecs\Output\output_alaw.pcm
```

2.5.2 Encoder Configuration File

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

A sample Testparams_enc.cfg file is as shown:

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

CompandingLaw   = 1   # 1 => A law,
                        2 => U law
```

Any field in the `ISPHENC1_Params` structure (see Section 4.2.1.3) can be set in the Testparams_enc.cfg file using the syntax shown above. If you specify additional fields in the Testparams_enc.cfg file, ensure to modify the test application appropriately to handle these fields.

2.5.3 Decoder Configuration File

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

A sample Testparams_dec.cfg file is as shown:

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

CompandingLaw   = 1   # 1 => A law,
                        2 => U law
```

Any field in the `ISPHDEC1_Params` structure (see Section 4.3.1.3) can be set in the Testparams_dec.cfg file using the syntax shown above. If you specify additional fields in the Testparams_dec.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:

1) Copy the input files to the \Client\Test\TestVecs\Inputs sub-directory.

2) Copy the reference files to the \Client\Test\TestVecs\Reference sub-directory.

3) Edit the configuration file, Testvecs_enc.cfg or Testvecs_dec.cfg available in the \Client\Test\TestVecs\Config sub-directory. For details on the format of the Testvecs_enc.cfg and Testvecs_dec.cfg files, see Section 2.5.1.

4) Execute the sample test application. On successful completion, the application displays one of the following messages for each frame:

   o "Encoder compliance test passed/failed" or "Decoder compliance test passed/failed" (if `X` is 1)

   o "Encoder output dump completed" or "Decoder output dump completed" (if `X` is 0)

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

Note:

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

2.7 Uninstalling the Component

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

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 `ISPHENC1` or `ISPHDEC1` base class of the G.711 Encoder or Decoder library. The main test application files are TestAppEncoder.c or TestAppDecoder.c and TestAppEncoder.h or TestAppDecoder.h. These files are available in the \Client\Test\Src and \Client\Test\Inc sub-directories respectively.

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

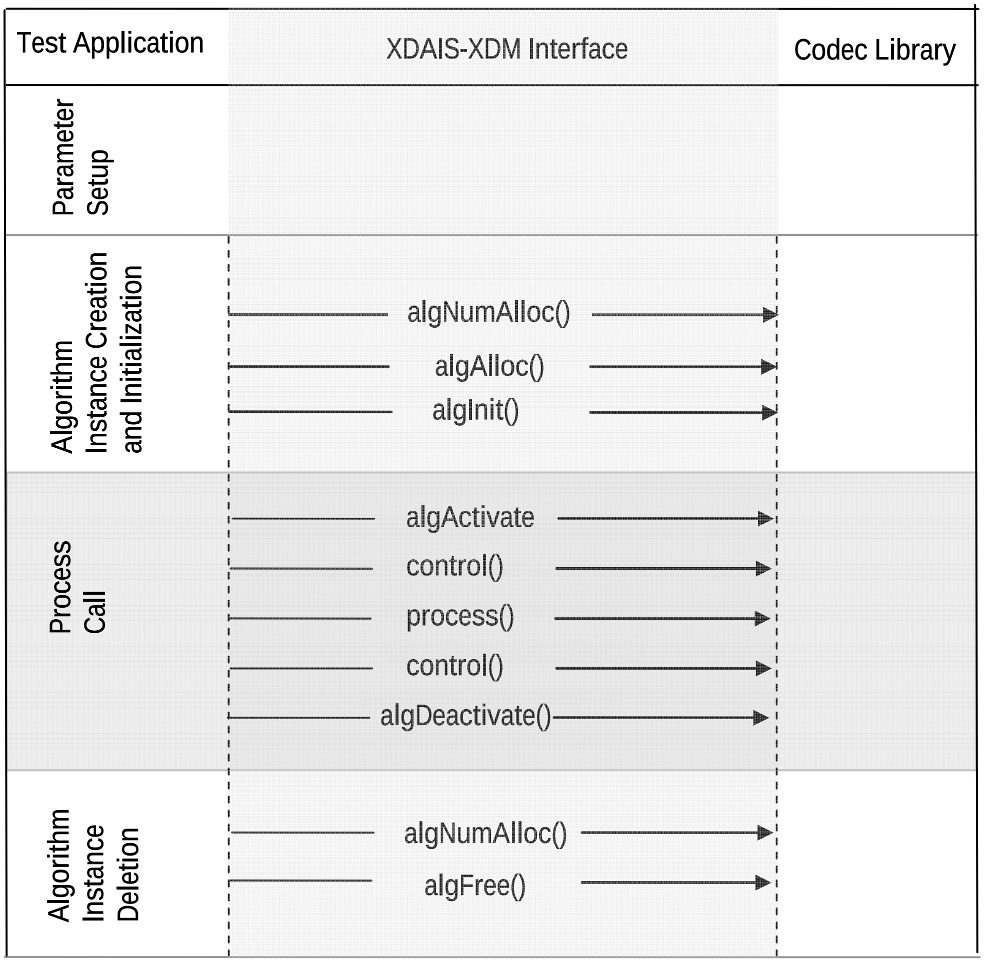


Figure 3-1. Test Application Sample Implementation

> **Note:**
>
> Speech codecs do not use `algActivate()`, `algDeactivate()`, and `DMAN3_init()` 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 or Decoder configuration files.

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

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

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

2) Reads the input bit-stream into the application input buffer.

After successful completion of the above steps, the test application does the algorithm instance creation and initialization.

3.1.2 Algorithm Instance Creation and Initialization

In this logical block, the test application accepts the various initialization parameters and returns an algorithm instance pointer. The following APIs are called in sequence:

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

2) `algAlloc()` - To query the algorithm about the memory requirement to be filled in the memory records.

3) `algInit()` - To initialize the algorithm with the memory structures provided by the application.

A sample implementation of the create function that calls `algNumAlloc()`, `algAlloc()`, and `algInit()` in sequence is provided in the `ALG_create()` function implemented in the alg_create.c file.

3.1.3 Process Call

After algorithm instance creation and initialization, the test application 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. The inputs to the process function are input and output buffer descriptors, pointer to the `ISPHENC1_InArgs` or `ISPHDEC1_InArgs` and `ISPHENC1_OutArgs` or `ISPHDEC1_OutArgs` structures.

4) Size of input speech samples for the encoder (frame size) and input buffer size for decoder are specified using the `bufSize` field in `XDM1_SingleBufDesc`.

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 or Decoder with appropriate input/output buffer and arguments information.

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

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

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

3.1.4 Algorithm Instance Deletion

Once encoding/decoding is complete, the test application must delete the current algorithm instance. The following APIs are called in sequence:

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

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

A sample implementation of the delete function that calls `algNumAlloc()` and `algFree()` in sequence is provided in the `ALG_delete()` function implemented in the alg_create.c file.

Chapter 4

API Reference

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

4.1 Symbolic Constants and Enumerated Data Types

This section summarizes all the symbolic constants specified as either #define macros and/or enumerated C data types. 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
XDM_AccessMode	XDM_ACCESSMODE_READ	Read from the buffer using the CPU
	XDM_ACCESSMODE_WRITE	Write to the buffer using the CPU
XDM_CmdId	XDM_GETSTATUS	Query to the algorithm instance to fill Status structure
	XDM_SETPARAMS	Set run-time dynamic parameters through the DynamicParams structure.
	XDM_RESET	Reset the algorithm.
	XDM_SETDEFAULT	Initialize all fields in Params structure to default values specified in the library
	XDM_FLUSH	Handle end of stream conditions. This command forces algorithm instance to output data without additional input.
	XDM_GETBUFINFO	Query algorithm instance regarding the properties of input and output buffers
	XDM_GETVERSION	Query the algorithms version.
ISPHDEC1_FrameType	ISPHDEC1_FTYPE_SPEECHGOOD	Regular speech frame received without error or loss
	ISPHDEC1_FTYPE_SIDUPDATE	SID update frames
	ISPHDEC1_FTYPE_NODATA	Untransmitted frame for codecs that support internal DTX
	ISPHDEC1_FTYPE_SPEECHLOST	Complete loss of speech frame
	ISPHDEC1_FTYPE_DEGRADED	Speech frame with a correct CRC or invalid data
	ISPHDEC1_FTYPE_BAD	Speech frame invalid
	ISPHDEC1_FTYPE_SIDFIRST	First frame of comfort noise

Group or Enumeration Class	Symbolic Constant Name	Description or Evaluation
	ISPHDEC1_FTYPE_SIDBAD	Corrupt SID update frame
	ISPHDEC1_FTYPE_ONSET	Frames which precede the first speech (frame of a speech burst)
ISPHENC1_FrameType	ISPHENC1_FTYPE_SPEECH	Speech frame
	ISPHENC1_FTYPE_SIDFRAME	SID frames for codecs which support DTX
	ISPHENC1_FTYPE_NODATA	Un-transmitted frame for codecs that support DTX
ISPEECH1_PCM_CompandingLaw	ISPEECH1_PCM_COMPAND_LINEAR	Linear
	ISPEECH1_PCM_COMPAND_ALAW	A-law
	ISPEECH1_PCM_COMPAND_ULAW	Mu-law

Note:

The constants XDM_SETPARAMS, XDM_RESET, XDM_FLUSH are not applicable for G.711 Encoder/Decoder. Also, the constant ISPEECH1_PCM_COMPAND_LINEAR is not applicable for G.711 Encoder/Decoder.

4.2 Encoder Data Structures

This section describes the XDM defined data structures that are common across codec classes. These XDM data structures can be extended to define any implementation specific parameters for a codec component.

4.2.1 Common XDM Data Structures

This section includes the following common XDM data structures:

- XDM1_SingleBufDesc
- ISPHENC1_Fxns
- ISPHENC1_Params
- ISPHENC1_DynamicParams
- ISPHENC1_OutArgs
- ISPHENC1_Status
- ISPHENC1_InArgs

4.2.1.1 XDM1_SingleBufDesc

|| Description

This structure defines the buffer descriptor for input and output buffers. The access field is filled by the algorithm, declaring how the algorithm processor accessed the buffer.

|| Fields

Field	Datatype	Input/ Output	Description
*buf	XDAS_Int8	Input	Pointer to a buffer address.
bufSize;	XDAS_Int32	Input	Size of buffer in bytes
accessMask	XDAS_Int32	Output	Mask filled by the algorithm, declaring how the buffer was accessed by the algorithm processor

4.2.1.2 ISPHENC1_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 (*) (ISPHENC1_Handle , XDM1_SingleBufDesc *inBuf, XDM1_SingleBufDesc *outBuf, ISPHENC1_InArgs *inArgs, ISPHENC1_OutArgs *outArgs)`	Input	Pointer to the `process()` function
`*control`	`XDAS_Int32 (*) (ISPHENC1_Handle , ISPHENC1_Cmd id, ISPHENC1_DynamicParams *dParams, ISPHENC1_Status *status)`	Input	Pointer to the `control()` function

4.2.1.3 ISPHENC1_Params

‖ Description

This structure defines the creation parameters for an algorithm instance object. You can 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_Int16`	Input	Size of the basic or extended data structure in bytes.
`compandingLaw`	`XDAS_Int16`	Input	Companding law (used for selection) ❑ 1 - A law(Default) ❑ 2 - U law

> **Note:**
>
> The parameters not specified in the table should be ignored with respect to G.711.

4.2.1.4 ISPHENC1_DynamicParams

‖ Description

This structure defines the run-time parameters for an algorithm instance object. Set this data structure to `NULL`, if you are not sure of the values to be specified for these parameters

‖ Fields

Field	Datatype	Input/ Output	Description
`size`	`XDAS_Int16`	Input	Size of the basic or extended (if being used) data structure in bytes.

> **Note:**
>
> There are no dynamic parameters for G.711. This section is not applicable for G.711 codec.

4.2.1.5 ISPHENC1_OutArgs

|| Description

This structure defines run-time output arguments used for an algorithm instance object.

|| Fields

Field	Datatype	Input/ Output	Description
size	XDAS_Int16	Input	Size of the basic or extended (if being used) data structure in bytes.
frameType	XDAS_Int16	Output	Frame type is set to ISPHENC1_FTYPE_SPEECH always.

> **Note:**
>
> The parameters not specified in the table should be ignored with respect to G.711 Codec.

4.2.1.6 ISPHENC1_Status

|| Description

This structure defines parameters that describe the status of an algorithm instance object.

|| Fields

Field	Datatype	Input/ Output	Description
size	XDAS_Int16	Input	Size of the basic or extended (if being used) data structure in bytes.
compandingLaw	XDAS_Int16	Output	Companding law (used for selection) ❑ 1 - A law ❑ 2 - U law

> **Note:**
>
> The parameters not specified in the table should be ignored with respect to G.711 Codec.

4.2.1.7 ISPHENC1_InArgs

‖ Description

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

‖ Fields

Field	Datatype	Input/ Output	Description
`size`	`XDAS_Int16`	Input	Size of the basic or extended (if being used) data structure in bytes.

> **Note:**
>
> The parameters not specified in the table should be ignored with respect to G.711 Codec.

4.3 Decoder Data Structures

This section describes the XDM defined data structures which are common across codec classes. These XDM data structures can be extended to define any implementation specific parameters for a codec component.

4.3.1 Common XDM Data Structures

This section includes the following common XDM data structures used across all the codec classes:

- `XDM1_SingleBufDesc`
- `ISPHDEC1_Fxns`
- `ISPHDEC1_Params`
- `ISPHDEC1_DynamicParams`
- `ISPHDEC1_InArgs`
- `ISPHDEC1_Status`
- `ISPHDEC1_OutArgs`

4.3.1.1 XDM1_SingleBufDesc

|| Description

This structure defines the buffer descriptor for input and output buffers.The access field is filled by the algorithm, declaring how the algorithm processor accessed the buffer.

|| Fields

Field	Datatype	Input/ Output	Description
`*buf`	`XDAS_Int8`	Input	Pointer to a buffer address.
`bufSize;`	`XDAS_Int32`	Input	Size of buffer in bytes
`accessMask`	`XDAS_Int32`	Output	Mask filled by the algorithm, declaring how the buffer was accessed by the algorithm processor

4.3.1.2 ISPHDEC1_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 (*) (ISPHDEC1_Handle, XDM1_SingleBufDesc *inBuf, XDM1_SingleBufDesc *outBuf, ISPHDEC1_InArgs *inArgs, ISPHDEC1_OutArgs *outArgs)`	Input	Pointer to the `process()` function
`*control`	`XDAS_Int32 (*) (ISPHDEC1_Handle , ISPHDEC1_Cmd id, ISPHDEC1_DynamicParams *dParams, ISPHDEC1_Status *status)`	Input	Pointer to the `control()` function

4.3.1.3 ISPHDEC1_Params

|| Description

This structure defines the creation time parameters for all algorithm instance objects. This structure will be extended by the IMOD interface for advanced (codec and implementation specific) parameters

|| Fields

Field	Datatype	Input/ Output	Description
`size`	`XDAS_Int16`	Input	Size of the basic or extended (if being used) data structure in bytes.
`compandingLaw`	`XDAS_Int16`	Input	Companding law (used for selection): ❑ 1 - A law (Default) ❑ 2 - U law

> **Note:**
>
> The parameters not specified in the table should be ignored with respect to G.711 Codec.

4.3.1.4 ISPHDEC1_DynamicParams

|| Description

This structure defines the run-time parameters for algorithm instance object. Set this data structure to `NULL`, if you are not sure of the values to be specified for these parameters.

|| Fields

Field	Datatype	Input/ Output	Description
`size`	`XDAS_Int16`	Input	Size of the basic or extended (if being used) data structure in bytes.

> **Note:**
>
> There are no dynamic parameters for G.711. This section is not applicable for G.711 codec.

4.3.1.5 ISPHDEC1_InArgs

|| Description

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

|| Fields

Field	Datatype	Input/ Output	Description
`size`	`XDAS_Int16`	Input	Size of the basic or extended data structure in bytes.

Note:

The parameters not specified in the table should be ignored with respect to G.711 Codec.

4.3.1.6 ISPHDEC1_Status

|| Description

This structure defines parameters that describe the status of the algorithm instance object

|| Fields

Field	Datatype	Input/ Output	Description
`size`	`XDAS_Int16`	Input	Size of the basic or extended (if being used) data structure in bytes.
`compandingLaw`	`XDAS_Int16`	Output	Companding law (used for selection): ❑ 1 - A law ❑ 2 - U law

Note:

The parameters not specified in the table should be ignored with respect to G.711 Codec.

4.3.1.7 ISPHDEC1_OutArgs

‖ Description

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

‖ Fields

Field	Datatype	Input/ Output	Description
`size`	`XDAS_Int16`	Input	Size of the basic or extended (if being used) data structure in bytes.

> **Note:**
>
> The parameters not specified in the table should be ignored with respect to G.711 Codec.

4.4 Interface Functions

This section describes the Application Programming Interfaces (APIs) used in the G.711 Encoder/Decoder. The APIs are logically grouped into the following categories:

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

You must call these APIs in the following sequence:

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

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

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

Note:

Speech codecs do not use `algActivate()`, `algDeactivate(), and DMAN3_init()` APIs.

4.4.1 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()`

4.4.2 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()`

4.4.3 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 of the encoder

|| Synopsis

```
XDAS_Int32 (*control) (ISPHENC1_Handle handle,
ISPHENC1_Cmd id, ISPHENC1_DynamicParams *params,
ISPHENC1_Status *status);
```

|| Arguments

```
ISPHENC1_Handle handle; /* algorithm instance handle */

ISPHENC1_Cmd id; /* algorithm specific control commands*/

ISPHENC1_DynamicParams *params /* algorithm run-time
parameters */

ISPHENC1_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 command controls XDM_FLUSH, XDM_RESET and XDM_SETPARAMS are not applicable for G.711 Encoder.

The third and fourth arguments are pointers to the ISPHENC1_DynamicParams and ISPHENC1_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:**
>
> Speech codecs do not use `algActivate()`, `algDeactivate()`, and `DMAN3_init()` APIs.

|| Name

control() – change run-time parameters and query the status of the decoder

|| Synopsis

```
XDAS_Int32 (*control) (ISPHDEC1_Handle handle,
ISPHDEC1_Cmd id, ISPHDEC1_DynamicParams *params,
ISPHDEC1_Status *status);
```

|| Arguments

```
ISPHDEC1_Handle handle; /* algorithm instance handle */

ISPHDEC1_Cmd id; /* algorithm specific control commands*/

ISPHDEC1_DynamicParams *params /* algorithm run-time
parameters */

ISPHDEC1_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 command controls XDM_FLUSH, XDM_RESET and XDM_SETPARAMS are not applicable for G.711 Decoder.

The third and fourth arguments are pointers to the ISPHDEC1_DynamicParams and ISPHDEC1_Status data structures respectively.

Note:

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

|| Preconditions

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

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

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

|| Postconditions

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

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

|| Example

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

|| See Also

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

> **Note:**
>
> Speech codecs do not use `algActivate()`, `algDeactivate()`, `and DMAN3_init()` APIs.

4.4.4 Data Processing API

Data processing API is used for processing the input data.

|| Name

`process()` – basic encoding call

|| Synopsis

```
XDAS_Int32 (*process)(ISPHENC1_Handle handle,
XDM1_SingleBufDesc *inBufs, XDM1_SingleBufDesc *outBufs,
ISPHENC1_InArgs *inargs, ISPHENC1_OutArgs *outargs);
```

|| Arguments

```
ISPHENC1_Handle handle; /* algorithm instance handle */

XDM1_SingleBufDesc *inBufs; /* algorithm input buffer
descriptor */

XDM1_SingleBufDesc *outBufs; /* algorithm output buffer
descriptor */

ISPHENC1_InArgs *inargs /* algorithm runtime input
arguments */

ISPHENC1_OutArgs *outargs /* algorithm runtime output
arguments */
```

|| Return Value

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

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

|| Description

This function does the basic encoding/decoding. The first argument to `process()` is a handle to an algorithm instance.

The second and third arguments are pointers to the input and output buffer descriptor data structures respectively (see `XDM1_SingleBufDesc` data structure for details).

The fourth argument is a pointer to the `ISPHENC1_InArgs` data structure that defines the run-time input arguments for an algorithm instance object.

The last argument is a pointer to the `ISPHENC1_OutArgs` data structure that defines the run-time output arguments for an algorithm instance object.

|| 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()`.
- `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.
- `bufSize` field of `outBufs` is updated with the number of bytes generated by encoder.

‖ Example

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

‖ See Also

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

> **Note:**
>
> Speech codecs do not use `algActivate()`, `algDeactivate()`, and `DMAN3_init()` APIs.

‖ Name

`process()` – basic decoding call

‖ Synopsis

```
XDAS_Int32 (*process)(ISPHDEC1_Handle handle,
XDM1_SingleBufDesc *inBufs, XDM1_SingleBufDesc *outBufs,
ISPHDEC1_InArgs *inargs, ISPHDEC1_OutArgs *outargs);
```

‖ Arguments

```
ISPHDEC1_Handle handle; /* algorithm instance handle */

XDM1_SingleBufDesc *inBufs; /* algorithm input buffer
descriptor */

XDM1_SingleBufDesc *outBufs; /* algorithm output buffer
descriptor */

ISPHDEC1_InArgs *inargs /* algorithm runtime input
arguments */

ISPHDEC1_OutArgs *outargs /* algorithm runtime output
arguments */
```

‖ Return Value

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

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

‖ Description

This function does the basic encoding/decoding. The first argument to `process()` is a handle to an algorithm instance.

The second and third arguments are pointers to the input and output buffer descriptor data structures respectively (see `XDM1_SingleBufDesc` data structure for details).

The fourth argument is a pointer to the `ISPHDEC1_InArgs` data structure that defines the run-time input arguments for an algorithm instance object.

The last argument is a pointer to the `ISPHDEC1_OutArgs` data structure that defines the run-time output arguments for an algorithm instance object.

‖ 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.
- `bufSize` field of `outBufs` is updated with the number of bytes generated by decoder.

‖ Example

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

‖ See Also

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

> **Note:**
>
> Speech codecs do not use `algActivate()`, `algDeactivate()`, and `DMAN3_init()` APIs.

4.4.5 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

Memory Placement Details

This appendix contains the memory placement details for G.711. Ensure that you take care of the buffer detail information provided in this section.

- The `algAlloc` API returns a pointer to a memory descriptor array. The numbering and ordering of buffers is same as that in this memory descriptor array.
- The alignment of a buffer is specified as module-N, For example, if a buffer requires modulo-8 alignment, then the alignment is given as 8. This is as per the XDAIS Guidelines.

A.1 G.711 Encoder Buffers

This section describes buffer number, attribute, buffer size, and alignment details for G.711 decoder.

Table A-1. G.711 Encoder Buffers.

Buffer Number	Attribute	Size in bytes	Alignment
0	STATIC	8	8

A.2 G.711 Decoder Buffers

This section describes buffer number, attribute, buffer size, and alignment details for G.711 decoder.

Table A-2. G.711 Decoder Buffers.

Buffer Number	Attribute	Size in bytes	Alignment
0	STATIC	8	8

This page is intentionally left blank

Optimal Function Placement Order for Minimizing Instruction Cache Penalty

This appendix contains the optimal function placement order for minimizing cache penalty due to instruction cache misses. This is also available in the sample linker command file.

The application linker command file contains the following code.

```
GROUP :
{
   .text:_G711ENC_TII_numAlloc
   .text:_G711ENC_TII_alloc
   .text:_G711ENC_TII_initObj
   .text:_G711ENC_TII_g711Encode
   .text:_G711_TII_Encoder
   .text:_G711ENC_TII_free
   .text:_G711ENC_TII_g711Control
   .text:_G711DEC_TII_numAlloc
   .text:_G711DEC_TII_alloc
   .text:_G711DEC_TII_initObj
   .text:_G711DEC_TII_g711Decode
   .text:_G711_TII_Decoder
   .text:_G711DEC_TII_g711Control
   .text:_G711DEC_TII_free

} > EMEM
```

This page is intentionally left blank

Appendix C

Revision History

This user guide revision history highlights the changes made to SPRUEC9A codec specific user guide to make it SPRUEC9B.

Table C-1. Revision History of G711 Encoder/Decoder on C64x+

Section	Additions/Modifications/Deletions
Global	❑ Modified code generation version to 6.0.15 ❑ Modified DSP/BIOS version to 5.31
Section 2.1.1	Hardware: ❑ Updated supported platforms
Section 2.2	Installing the Component: ❑ Updated top-level directory name
Section 4.2.1.6	`ISPHENC1_Status`: Added the following field: ❑ `compandingLaw`
Section 4.3.1.6	`ISPHDEC1_Status`: Added the following field: ❑ `compandingLaw`
Section 4.4.4	Data Processing API: ❑ Modified all occurrences of `XDM_BufDesc` to `XDM1_SingleBufDesc`