MP3 Encoder

Programmer's Guide

For HiFi DSPs and Fusion F1 DSP

.

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Version: 1.5
Last Updated: August 2022

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Contents

Figures

Tables

Document Change History

Version	Changes
1.5	• Added performance and memory data for HiFi1 in the section 1.4 HiFi MP3 Encoder Performance.
1.4	• Added performance data for Fusion F1 and HiFi 3z/4/5 in the section 1.4 HiFi MP3 Encoder Performance.
1.3	• Added memory data for the HiFi Mini in the section 1.4.1 Memory. • Added timing data for the HiFi Mini in the section 1.4.2 Timings
1.2	• CPU load of the encoder has been significantly optimized. • Added new performance data for HiFi 3. • Changed generic references from “HiFi 2” to “HiFi”. • Deleted references to Diamond 330HiFi.

1. Introduction to the HiFi MP3 Encoder

The HiFi MP3 Encoder implements the audio codec standard specified as part of the MPEG-1 and MPEG-2 Specifications. *These standards come from the Moving Picture Experts Group (MPEG) which is a working group of ISO/IEC.*[1] [2].

Note	For this document, HiFi DSPs include Fusion F1 DSP.

1.1 MP3 Description

"MPEG-1 Audio Layer III, also known as MP3, has been implemented in manifold ways. Many software packages exist to rip a track from a CD Audio and compress it in MP3. This has given rise to innovative ways of consuming music, such as the ability to create compilations to one's liking that can then downloaded to light non-mechanical MP3 players. With the arrival of MP3 the music world has been changed without recognition."

Quote From: Moving Picture Experts Group

The first MP3 player arrived in the year 1998. MP3 is extended by MPEG-2 to support further sampling rates and encoded bit rates. It was a success due to better compression achieved as compared to the CD format.

The target of 1.33 bits per sample results in a bitstream of 128 kbit/s for stereo 48 kHz audio. This compression is 11.7 times over the uncompressed PCM Audio.

1.2 Document Overview

This document provides all the information required to integrate the HiFi audio codecs into an application. The HiFi codec libraries implement a simple API to encapsulate the complexities of the coding operations and simplify the application and system implementation. Parts of the API described in this document are common to all the HiFi codecs. This document also details the features, provides HiFi MP3 Encoder specific information, and describes an example test bench.

1.3 HiFi MP3 Encoder Specifications

The HiFi DSP MP3 Encoder from Cadence Tensilica implements the following features in conformance to the MPEG-1 and MPEG-2 specifications:

- MPEG-2 Layer 3 Encoding
- All sampling frequencies from 16KHz to 48 kHz according to MPEG-1/2 [1][2]
- Bit rates 16-320 kbps
- Supports both mono and stereo data
- Constant bit rate (CBR)
- Joint Stereo (MS only)

The HiFi DSP MP3 Encoder does not support the following features:

- Intensity Stereo
- MPEG 2.5
- Variable bit rate (VBR)
- Average bit rate (ABR)
- Bit rate at 8 kbps

The HiFi DSP MP3 Encoder supports MP3 CBR files with a limited number of sample and data rates.

- For MPEG-1 sample rates of 32000Hz, 44100Hz and 48000Hz, the bit rates (in kbps) are 32, 40, 48, 56, 64, 80, 96, 112, 128 160, 192, 224, 256, 320.
- For MPEG-2 sample rates of 16000Hz, 22050Hz and 24000Hz, the bit rates (in kbps) are 16, 24, 32, 40, 48, 56, 64, 80, 96, 112, 128, 144, 160.

The HiFi DSP MP3 Encoder implements Cadence Audio Codec API.

1.4 HiFi MP3 Encoder Performance

The HiFi DSP MP3 Encoder from Cadence Tensilica is characterized on the HiFi 5-stage DSP. In the following sections, the memory usage and performance figures are provided for design reference.

1.4.1 Memory

Text (Kbytes)						Data
Fusion F1	HiFi 1	HiFi 3	HiFi 3z	HiFi 4	HiFi 5	Kbytes
48.5	49.2	47.7	52.9	54.8	77.1	14.2

Runtime Memory (Kbytes)				
Persistent	Scratch	Stack	Input	Output
23.1	14.0	3.2	4.5	2.0

1.4.2 Timings

Rate kHz	Channels	Bit Rate kbps	Average CPU Load (MHz)					
			Fusion F1	HiFi 1	HiFi 3	HiFi 3z	HiFi 4	HiFi 5
44.1	2	128	15.3	14.6	15.2	12.8	11.8	11.1
44.1	2	320	16.8	16.1	16.8	13.7	12.6	12.1

Note Performance specification measurements are carried on a cycle-accurate simulator assuming an ideal memory system, that is, one with zero memory wait states. This is equivalent to running with all code and data in local memories or using an infinite-size, pre-filled cache model.

Note The MCPS numbers for HiFi 3/HiFi 3z/HiFi 4/Fusion F1 are obtained by running the test with binary, that is, based on 24-bit optimized source code.

Note The MCPS numbers for HiFi 5 are obtained by running the test with binary, that is, recompiled from the HiFi 1's optimized source code in HiFi 5 configuration. No specific optimization is done for HiFi 5.

Note All the memory and MCPS numbers are captured with XT-CLANG compiler and RI-2021.8 tools.

2. Generic HiFi Audio Codec API

This section describes the API that is common to all the HiFi audio codec libraries. The API facilitates any codec that works in the overall method shown in the following diagram.

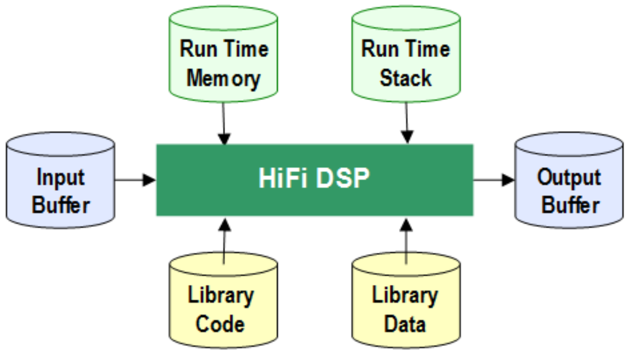

Figure 1 HiFi Audio Codec Interfaces

Section 2.1 discusses all the types of run time memory required by the codecs. There is no state information held in static memory, therefore a single thread can perform time division processing of multiple codecs. Additionally, multiple threads can perform concurrent codec processing. The API is implemented so that the application does not need to consider the codec implementation.

Through the API, the codec requests the minimum sizes required for the input and output buffers. Prior to executing the codec execution command, the codec requires that the input buffer is filled with data up to the minimum size for the input buffer. However, the codec may not consume all of the data in the input buffer. Therefore, the application must check the amount of input data consumed, copy downwards any unused portion of the input buffer, and then continue to fill the rest of the buffer with new data until the input buffer is again filled to the minimum size. The codec will produce data in the output buffer. The output data must be removed from the output buffer after the codec operation.

Applications that use these libraries should not make any assumptions about the size of the PCM “chunks” of data that each call to a codec produces or consumes. Although normally the chunks are the exact size of the underlying frame of the specified codec algorithm, they will vary between codecs and also between different operating modes of the same codec. The application should provide enough data to fill the input buffer. However, some codecs do provide information, after the initialization stage, to adjust the number of bytes of PCM data they need.

2.1 Memory Management

The HiFi audio codec API supports a flexible memory scheme and a simple interface that eases the integration into the final application. The API allows the codecs to request the required memory for their operations during run time.

The runtime memory requirement consists primarily of the scratch and persistent memory. The codecs also require an input buffer and output buffer for the passing of data into and out of the codec.

2.1.1 API Object

The codec API stores its data in a small structure that is passed via a handle that is a pointer to an opaque object from the application for each API call. All state information and the memory tables that the codec requires are referenced from this structure.

2.1.2 API Memory Table

During the memory allocation, the application is prompted to allocate memory for each of the following memory areas. The reference pointer to each memory area is stored in this memory table. The reference to the table is stored in the API object.

2.1.3 Persistent Memory

This is also known as static or context memory. This is the state or history information that is maintained from one codec invocation to the next within the same thread or instance. The codecs expect that the contents of the persistent memory be unchanged by the system apart from the codec library itself for the complete lifetime of the codec operation.

2.1.4 Scratch Memory

This is the temporary buffer used by the codec for processing. The contents of this memory region should be unchanged if the actual codec execution process is active; that is, if the thread running the codec is inside any API call. This region can be used freely by the system between successive calls to the codec.

2.1.5 Input Buffer

This is the buffer used by the algorithm for accepting input data. Before the call to the codec, the input buffer must be completely filled with input data.

2.1.6 Output Buffer

This is the buffer in which the algorithm writes the output. This buffer must be made available for the codec before its execution call. The output buffer pointer can be changed by the application between calls to the codec. This allows the codec to write directly to the required output area. The codec will never write more data than the requested size of the output buffer.

2.2 C Language API

A single interface function is used to access the codec, with the operation specified by command codes. The actual API C call is defined per codec library and is specified in the codec-specific section. Each library has a single C API call.

The C parameter definitions for every codec library are identical and are specified in the following table.

Table 2-1 Codec API

xa_<codec>	
Description	This C API is the only access function to the audio codec.

Syntax	`XA_ERRORCODE xa_<codec>(`
	`      xa_codec_handle_t  p_xa_module_obj,`
	`      WORD32 i_cmd,`
	`      WORD32 i_idx,`
	`      pVOID  pv_value);`

Parameters	`p_xa_module_obj`
	Pointer to the opaque API structure.
	
	`i_cmd`
	Command.
	
	`i_idx`
	Command subtype or index.
	
	`pv_value`
	Pointer to the variable used to pass in, or get out properties from the state structure.

Returns	Error Code based on the success or failure of the API command

The types used for the C API call are defined in the supplied header files as:

```
typedef signed int              WORD32;
typedef void                   *pVOID;
```

Each time the C API for the codec is called, a pointer to a private allocated data structure is passed as the first argument. This argument is treated as an opaque handle as there is no requirement by the application to look at the data within the structure. The size of the structure is supplied by a specific API command so that the application can allocate the required memory. Do not use `sizeof()` on the type of the opaque handle.

Some command codes are further divided into subcommands. The command and its subcommand are passed to the codec via the second and third arguments, respectively.

When a value must be passed to a particular API command or an API command returns a value, the value expected or returned is passed through a pointer, which is given as the fourth argument to the C API function. In the case of passing a pointer value to the codec, the pointer is just cast to `pVOID`. It is incorrect to pass a pointer to a pointer in these cases. An example would be when the application is passing the codec a pointer to an allocated memory region.

Due to the similarities of the operations required to decode or encode audio streams, the HiFi DSP API allows the application to use a common set of procedures for each stage. By maintaining a pointer to the single API function and passing the correct API object, the same code base can be used to implement the operations required for any of the supported codecs.

2.3 Generic API Errors

The error code returned is of type `XA_ERRORCODE`, which is of type `signed int`. The format of the error codes is defined in the following table.

Table 2-2 Error Codes Format

31	30-15	14 – 11	10 – 6	5 – 0
Fatal	Reserved	Class	Codec	Sub code

The errors that can be returned from the API are subdivided into those that are fatal, which require the restarting of the entire codec, and those that are nonfatal and are provided for information to the application.

The class of an error can be API, Config, or Execution. The API errors are concerned with the incorrect use of the API. The Config errors are produced when the codec parameters are incorrect or outside the supported usage. The Execution errors are returned after a call to the main encoding or decoding process and indicate situations that have arisen due to the input data.

2.4 Commands

This section covers the commands associated with the following command sequence overview flow chart. For each stage of the flow chart there is a section that lists the required commands in the order they should occur. For individual commands, definitions, and examples refer to Section 2.6. The codecs have a common set of generic API commands that are represented by the white stages. The yellow stages are specific to each codec.

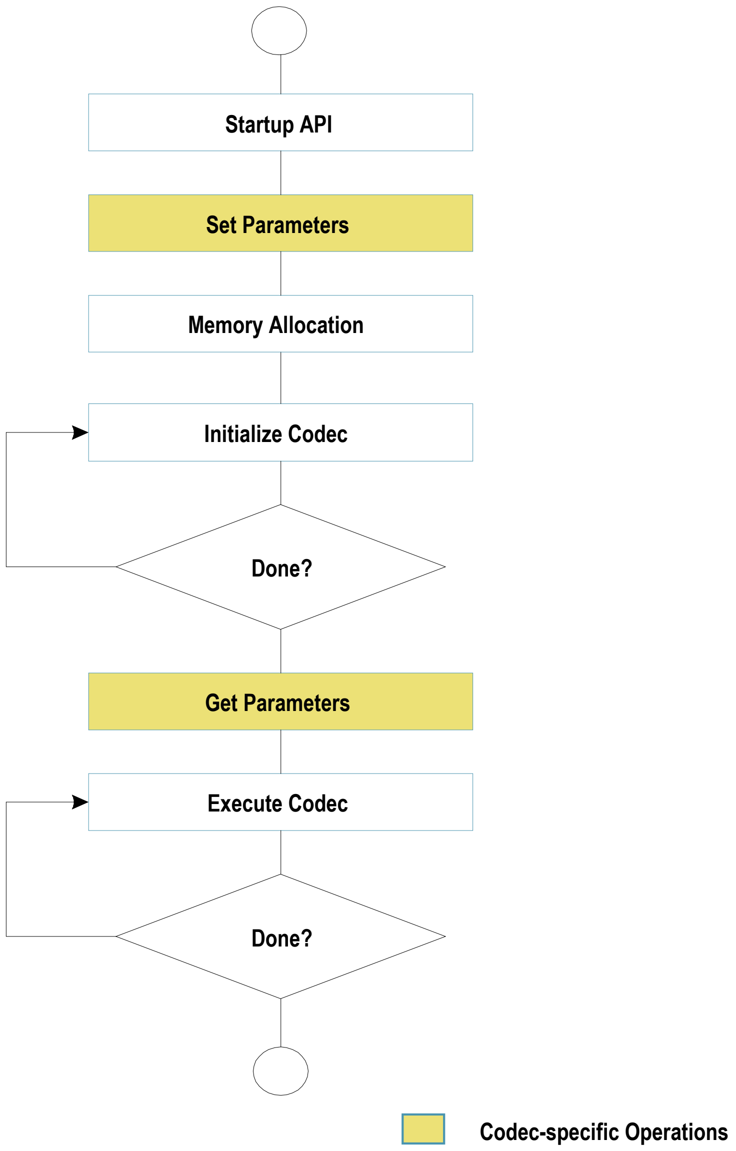

Figure 2 API Command Sequence Overview

2.4.1 Start-up API Stage

The following commands should be executed once each during start-up. The commands to get the various identification strings from the codec library are for information only and are optional. The command to get the API object size is mandatory as the real object type is hidden in the library and therefore there is no type available to use with `sizeof()`.

Table 2-3 Commands for Initialization

Command / Subcommand	Description
`XA_API_CMD_GET_LIB_ID_STRINGS` `XA_CMD_TYPE_LIB_NAME`	Get the name of the library.
`XA_API_CMD_GET_LIB_ID_STRINGS` `XA_CMD_TYPE_LIB_VERSION`	Get the version of the library.
`XA_API_CMD_GET_LIB_ID_STRINGS` `XA_CMD_TYPE_API_VERSION`	Get the version of the API.
`XA_API_CMD_GET_API_SIZE`	Get the size of the API structure.
`XA_API_CMD_INIT` `XA_CMD_TYPE_INIT_API_PRE_CONFIG_PARAMS`	Set the default values of all the configuration parameters.

2.4.2 Set Codec-Specific Parameters Stage

Refer to the specific codec section for the parameters that can be set. These parameters either control the encoding process or determine the output format of the decoder PCM data.

Table 2-4 Commands for Setting Parameters

Command / Subcommand	Description

`XA_API_CMD_SET_CONFIG_PARAM`
`XA_<codec>_CONFIG_PARAM_<param_name>`
	Set the codec-specific parameter. See the codec-specific section for parameter definitions.

2.4.3 Memory Allocation Stage

The following commands should be executed once only after all the codec-specific parameters have been set. The API is passed the pointer to the memory table structure (MEMTABS) after it is allocated by the application to the size specified. After the codec specific parameters are set, the initial codec setup is completed by performing the post-configuration portion of the initialization to determine the initial operating mode of the codec and assign sizes to the blocks of memory required for its operation. The application then requests a count of the number of memory blocks.

Table 2-5 Commands for Initial Table Allocation

Command / Subcommand	Description
`XA_API_CMD_GET_MEMTABS_SIZE`	Get the size of the memory structures to be allocated for the codec tables.
`XA_API_CMD_SET_MEMTABS_PTR`	Pass the memory structure pointer allocated for the tables.
`XA_API_CMD_INIT` `XA_CMD_TYPE_INIT_API_POST_CONFIG_PARAMS`	Calculate the required sizes for all the memory blocks based on the codec-specific parameters.
`XA_API_CMD_GET_N_MEMTABS`	Obtain the number of memory blocks required by the codec.

The following commands should then be executed in a loop to allocate the memory. The application first requests all the attributes of the memory block and then allocates it. It is important to abide by the alignment requirements. Finally, the pointer to the allocated block of memory is passed back through the API. For the input and output buffers it is not necessary to assign the correct memory at this point. The input and output buffer locations must be assigned before their first use in the EXECUTE stage. The type field refers to the memory blocks, for example input or persistent, as described in Section 2.1.

Table 2-6 Commands for Memory Allocation

Command / Subcommand	Description
`XA_API_CMD_GET_MEM_INFO_SIZE`	Get the size of the memory type being referred to by the index.
`XA_API_CMD_GET_MEM_INFO_ALIGNMENT`	Get the alignment information of the memory-type being referred to by the index.
`XA_API_CMD_GET_MEM_INFO_TYPE`	Get the type of memory being referred to by the index.
`XA_API_CMD_GET_MEM_INFO_PRIORITY`	Get the allocation priority of memory being referred to by the index.
`XA_API_CMD_SET_MEM_PTR`	Set the pointer to the memory allocated for the referred index to the input value.

2.4.4 Initialize Codec Stage

The following commands should be executed in a loop during initialization. These commands should be called until the initialization is completed as indicated by the `XA_CMD_TYPE_INIT_DONE_QUERY` command. In general, decoders can loop multiple times until the header information is found. However, encoders will perform exactly one call before they signal they are done.

There is a major difference between encoding Pulse Code Modulated (PCM) data and decoding stream data. During the initialization of a decoder, the initialization task reads the input stream to discover the parameters of the encoding. However, for an encoder there is no header information in PCM data. Even so, the encoder application is still required to perform the initialization described in this stage. However, encoders will not consume data during initialization. Furthermore, this has an implication in that some encoders provide parameters that can be used to modify the input buffer data requirements after the initialization stage. These modifications will always be a reduction in the size. The application only needs to provide the reduced amount per execution of the main codec process.

In general, the application will signal to the codec the number of bytes available in the input buffer and signal if it is the last iteration. It is not normal to hit the end of the data during initialization, but in the case of a decoder being presented with a corrupt stream it will allow a graceful termination. After the codec initialization is called, the application will ask for the number of bytes consumed. The application can also ask if the initialization is complete, it is advisable to always ask, even in the case of encoders that require only a single pass. A decoder application must keep iterating until it is complete.

Table 2-7 Commands for Initialization

Command / Subcommand	Description
`XA_API_CMD_SET_INPUT_BYTES`	Set the number of bytes available in the input buffer for initialization.
`XA_API_CMD_INPUT_OVER`	Signal to the codec the end of the bitstream.
`XA_API_CMD_INIT` `XA_CMD_TYPE_INIT_PROCESS`	Search for the valid header, does header decoding to get the parameters and initializes state and configuration structures.
`XA_API_CMD_INIT` `XA_CMD_TYPE_INIT_DONE_QUERY`	Check if the initialization process has completed.
`XA_API_CMD_GET_CURIDX_INPUT_BUF`	Get the number of input buffer bytes consumed by the last initialization.

2.4.5 Get Codec-Specific Parameters Stage

Finally, after the initialization, the codec can supply the application with information. In the case of decoders this would be the parameters it has extracted from the encoded header in the stream.

Table 2-8 Commands for Getting Parameters

Command / Subcommand	Description

`XA_API_CMD_GET_CONFIG_PARAM`
`XA_<codec>_CONFIG_PARAM_<param_name>`
	Get the value of the parameter from the codec. See the codec-specific section for parameter definitions.

2.4.6 Execute Codec Stage

The following commands should be executed continuously until the data is exhausted or the application wants to terminate the process. This is similar to the initialization stage, but includes support for the management of the output buffer. After each iteration, the application requests how much data is written to the output buffer. This amount is always limited by the size of the buffer requested during the memory block allocation. (To alter the output buffer position use `XA_API_CMD_SET_MEM_PTR` with the output buffer index.)

Table 2-9 Commands for Codec Execution

Command / Subcommand	Description
`XA_API_CMD_INPUT_OVER`	Signal the end of bitstream to the library.
`XA_API_CMD_SET_INPUT_BYTES`	Set the number of bytes available in the input buffer for the execution.
`XA_API_CMD_EXECUTE` `XA_CMD_TYPE_DO_EXECUTE`	Execute the codec thread.
`XA_API_CMD_EXECUTE` `XA_CMD_TYPE_DONE_QUERY`	Check if the end of stream has been reached.
`XA_API_CMD_GET_OUTPUT_BYTES`	Get the number of bytes output by the codec in the last frame.
`XA_API_CMD_GET_CURIDX_INPUT_BUF`	Get the number of input buffer bytes consumed by the last call to the codec.

2.5 Files Describing the API

Following are the common include files (`include`):

- `xa_apicmd_standards.h`

  The command definitions for the generic API calls

- `xa_error_standards.h`

  The macros and definitions for all the generic errors

- `xa_memory_standards.h`

  The definitions for memory block allocation

- `xa_type_def.h`

  All the types required for the API calls

2.6 HiFi API Command Reference

In this section, the different commands are described along with their associated subcommands. The only commands missing are those specific to a single codec. The particular codec commands are generally the SET and GET commands for the operational parameters.

The commands are listed below in sections based on their primary commands type (`i_cmd`). Each section contains a table for every subcommand. In the case of no subcommands the one primary command is presented.

The commands are followed by an example C call. Along with the call there is a definition of the variable types used. This is to avoid any confusion over the type of the fourth argument. The examples are not complete C code extracts as there is no initialization of the variables before they are used.

The errors returned by the API are detailed after each of the command definitions. However, there are a few errors that are common to all the API commands; these are listed in Section 2.6.1. All the errors possible from the codec-specific commands will be defined in the codec-specific sections. Furthermore, the codec-specific sections also cover the Execution errors that occur during the initialization or execution calls to the API.

2.6.1 Common API Errors

These errors are fatal and should not be encountered during normal application operation. They signal that a serious error has occurred in the application that is calling the codec.

- XA_API_FATAL_MEM_ALLOC

  `p_xa_module_obj` is `NULL`

- XA_API_FATAL_MEM_ALIGN

  `p_xa_module_obj` is not aligned to 4 bytes

- XA_API_FATAL_INVALID_CMD

  `i_cmd` is not a valid command

- XA_API_FATAL_INVALID_CMD_TYPE

  `i_idx` is invalid for the specified command `(i_cmd)`

2.6.2 XA_API_CMD_GET_LIB_ID_STRINGS

Table 2-10 XA_CMD_TYPE_LIB_NAME subcommand

Subcommand	`XA_CMD_TYPE_LIB_NAME`
Description	This command obtains the name of the library in the form of a string. The maximum length of the string that the library will provide is 30 bytes. Therefore the application shall pass a pointer to a buffer of a minimum size of 30 bytes. This command is optional.

Actual Parameters	`p_xa_module_obj`
	`NULL`
	
	`i_cmd`
	`XA_API_CMD_GET_LIB_ID_STRINGS`
	
	`i_idx`
	`XA_CMD_TYPE_LIB_NAME`
	
	`pv_value`
	`process name` – Pointer to a character buffer in which the name of the library is returned

Restrictions	None

Note No codec object is required due to the name being static data in the codec library.

Example

```
char process_name[30];
res = (*api_func)(NULL,
       XA_API_CMD_GET_LIB_ID_STRINGS,
       XA_CMD_TYPE_LIB_NAME,
       (pVOID) process_name);
```

Errors

- XA_API_FATAL_MEM_ALLOC

  This error is suppressed as `p_xa_module_obj` is `NULL`

- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

Table 2-11 XA_CMD_TYPE_LIB_VERSION subcommand

Subcommand	`XA_CMD_TYPE_LIB_VERSION`
Description	This command obtains the version of the library in the form of a string. The maximum length of the string that the library will provide is 30 bytes. Therefore, the application shall pass a pointer to a buffer of a minimum size of 30 bytes. This command is optional.

Actual Parameters	`p_xa_module_obj`
	`NULL`
	
	`i_cmd`
	`XA_API_CMD_GET_LIB_ID_STRINGS`
	
	`i_idx`
	`XA_CMD_TYPE_LIB_VERSION`
	
	`pv_value`
	`lib_version` – Pointer to a character buffer in which the version of the library is returned

Restrictions	None

Note No codec object is required due to the version being static data in the codec library.

Example

```
char lib_version[30];
res = (*api_func)(NULL,
        XA_API_CMD_GET_LIB_ID_STRINGS,
        XA_CMD_TYPE_LIB_VERSION,
        (pVOID) lib_version);
```

Errors

- XA_API_FATAL_MEM_ALLOC

  This error is suppressed as `p_xa_module_obj` is `NULL`

- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

Table 2-12 XA_CMD_TYPE_API_VERSION subcommand

Subcommand	`XA_CMD_TYPE_API_VERSION`
Description	This command obtains the version of the API in the form of a string. The maximum length of the string that the library will provide is 30 bytes. Therefore, the application shall pass a pointer to a buffer of a minimum size of 30 bytes. This command is optional.

Actual Parameters	`p_xa_module_obj`
	`NULL`
	
	`i_cmd`
	`XA_API_CMD_GET_LIB_ID_STRINGS`
	
	`i_idx`
	`XA_CMD_TYPE_API_VERSION`
	
	`pv_value`
	`api_version` – Pointer to a character buffer in which the version of the API is returned

Restrictions	None

Note No codec object is required due to the version being static data in the codec library.

Example

```
char api_version[30];
res = (*api_func)(NULL,
        XA_API_CMD_GET_LIB_ID_STRINGS,
        XA_CMD_TYPE_API_VERSION,
        (pVOID) api_version);
```

Errors

- XA_API_FATAL_MEM_ALLOC

  This error is suppressed as `p_xa_module_obj` is `NULL`

- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.3 XA_API_CMD_GET_API_SIZE

Table 2-13 XA_API_CMD_GET_API_SIZE command

Subcommand	None
Description	This command is used to obtain the size of the API structure, in order to allocate memory for the API structure. The pointer to the API size variable is passed and the API returns the size of the structure in bytes. The API structure is used for the interface and is persistent.

Actual Parameters	`p_xa_module_obj`
	`NULL`
	
	`i_cmd`
	`XA_API_CMD_GET_API_SIZE`
	
	`i_idx`
	`NULL`
	
	`pv_value`
	`&api_size` – Pointer to the API size variable

Restrictions	The application will allocate memory with an alignment of 4 bytes.

Note No codec object is required due to the size being fixed for the codec library.

Example

```
unsigned int api_size;
res = (*api_func)(NULL,
        XA_API_CMD_GET_API_SIZE,
        0,
        (pVOID) &api_size);
```

Errors

- XA_API_FATAL_MEM_ALLOC

  This error is suppressed as `p_xa_module_obj` is `NULL`

- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.4 XA_API_CMD_INIT

Table 2-14 XA_CMD_TYPE_INIT_API_PRE_CONFIG_PARAMS subcommand

Subcommand	`XA_CMD_TYPE_INIT_API_PRE_CONFIG_PARAMS`
Description	This command is used to set the default value of the configuration parameters. The configuration parameters can then be altered by using one of the codec-specific parameters setting commands. Refer to the codec-specific section.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_INIT`
	
	`i_idx`
	`XA_CMD_TYPE_INIT_API_PRE_CONFIG_PARAMS`
	
	`pv_value`
	`NULL`

Restrictions	None

Example

```
res = (*api_func)(api_obj,
        XA_API_CMD_INIT,
        XA_CMD_TYPE_INIT_API_PRE_CONFIG_PARAMS,
        NULL);
```

Errors

- Common API Errors

Table 2-15 XA_CMD_TYPE_INIT_API_POST_CONFIG_PARAMS subcommand

Subcommand	`XA_CMD_TYPE_INIT_API_POST_CONFIG_PARAMS`
Description	This command is used to calculate the sizes of all the memory blocks required by the application. It should occur after the codec-specific parameters have been set.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_INIT`
	
	`i_idx`
	`XA_CMD_TYPE_INIT_API_POST_CONFIG_PARAMS`
	
	`pv_value`
	`NULL`

Restrictions	None

Example

```
res = (*api_func)(api_obj,
      XA_API_CMD_INIT,
      XA_CMD_TYPE_INIT_API_POST_CONFIG_PARAMS,
      NULL);
```

Errors

- Common API Errors

Table 2-16 XA_CMD_TYPE_INIT_PROCESS subcommand

Subcommand	`XA_CMD_TYPE_INIT_PROCESS`
Description	This command initializes the codec. No output data is created during initialization.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_INIT`
	
	`i_idx`
	`XA_CMD_TYPE_INIT_PROCESS`
	
	`pv_value`
	`NULL`

Restrictions	None

Example

```
res = (*api_func)(api_obj,
      XA_API_CMD_INIT,
      XA_CMD_TYPE_INIT_PROCESS,
      NULL);
```

Errors

- Common API Errors
- See the codec-specific section for execution errors

Table 2-17 XA_CMD_TYPE_INIT_DONE_QUERY subcommand

Subcommand	`XA_CMD_TYPE_INIT_DONE_QUERY`
Description	This command checks to see if the initialization process has completed. If it has, the flag value is set to 1; otherwise it is set to zero. A pointer to the flag variable is passed as an argument.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_INIT`
	
	`i_idx`
	`XA_CMD_TYPE_INIT_DONE_QUERY`
	
	`pv_value`
	`&init_done` – Pointer to a flag that indicates the completion of initialization process

Restrictions	None

Example

```
unsigned int init_done;
res = (*api_func)(api_obj,
      XA_API_CMD_INIT,
      XA_CMD_TYPE_INIT_DONE_QUERY,
      (pVOID) &init_done);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.5 XA_API_CMD_GET_MEMTABS_SIZE

Table 2-18 XA_API_CMD_GET_MEMTABS_SIZE command

Subcommand	None
Description	This command is used to obtain the size of the table used to hold the memory blocks required for the codec operation. The API returns the total size of the required table. A pointer to the size variable is sent with this API command and the codec writes the value to the variable.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_GET_MEMTABS_SIZE`
	
	`i_idx`
	`NULL`
	
	`pv_value`
	`&proc_mem_tabs_size` – Pointer to the memory size variable

Restrictions	The application shall allocate memory with an alignment of 4 bytes.

Example

```
unsigned int proc_mem_tabs_size;
res = (*api_func)(api_obj,
        XA_API_CMD_GET_MEMTABS_SIZE,
        0,
        (pVOID) &proc_mem_tabs_size);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.6 XA_API_CMD_SET_MEMTABS_PTR

Table 2-19 XA_API_CMD_SET_MEMTABS_PTR command

Subcommand	None
Description	This command is used to set the memory structure pointer in the library to the allocated value.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_SET_MEMTABS_PTR`
	
	`i_idx`
	`NULL`
	
	`pv_value`
	`alloc` – Allocated pointer

Restrictions	The application will allocate memory with an alignment of 4 bytes.

Example

```
int * alloc; //alloc is a pointer to the allocated memory
res = (*api_func)(api_obj,
      XA_API_CMD_SET_MEMTABS_PTR,
      0,
      (pVOID) alloc);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_API_FATAL_MEM_ALIGN

  `pv_value` is not aligned to 4 bytes

2.6.7 XA_API_CMD_GET_N_MEMTABS

Table 2-20 XA_API_CMD_GET_N_MEMTABS command

Subcommand	None
Description	This command obtains the number of memory blocks needed by the codec. This value is used as the iteration counter for the allocation of the memory blocks. A pointer to each memory block will be placed in the previously allocated memory tables. The pointer to the variable is passed to the API and the codec writes the value to this variable.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_GET_N_MEMTABS`
	
	`i_idx`
	`NULL`
	
	`pv_value`
	`&n_mems` – Number of memory blocks required to be allocated

Restrictions	None

Example

```
int n_mems;
res = (*api_func)(api_obj,
      XA_API_CMD_GET_N_MEMTABS,
      0,
      (pVOID) &n_mems);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.8 XA_API_CMD_GET_MEM_INFO_SIZE

Table 2-21 XA_API_CMD_GET_MEM_INFO_SIZE command

Subcommand	Memory index
Description	This command obtains the size of the memory type being referred to by the index. The size in bytes is returned in the variable pointed to by the final argument. Note this is the actual size needed, not including any alignment packing space.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_GET_MEM_INFO_SIZE`
	
	`i_idx`
	Index of the memory
	
	`pv_value`
	`&size` – Pointer to the memory size

Restrictions	None

Example

```
int index;
unsigned int size;
res = (*api_func)(api_obj,
        XA_API_CMD_GET_MEM_INFO_SIZE,
        index,
        (pVOID) &size);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_API_FATAL_INVALID_CMD_TYPE

  `i_idx` is an invalid memory block number; valid block numbers obey the relation `0 <= i_idx < n_mems` (See XA_API_CMD_GET_N_MEMTABS)

2.6.9 XA_API_CMD_GET_MEM_INFO_ALIGNMENT

Table 2-22 XA_API_CMD_GET_MEM_INFO_ALIGNMENT command

Subcommand	Memory index
Description	This command gets the alignment information of the memory-type being referred to by the index. The alignment required in bytes is returned to the application.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_GET_MEM_INFO_ALIGNMENT`
	
	`i_idx`
	Index of the memory
	
	`pv_value`
	`&alignment` – Pointer to the alignment info variable

Restrictions	None

Example

```
int index;
unsigned int alignment;
res = (*api_func)(api_obj,
        XA_API_CMD_GET_MEM_INFO_ALIGNMENT,
        index,
        (pVOID) &alignment);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_API_FATAL_INVALID_CMD_TYPE

  `i_idx` is an invalid memory block number; valid block numbers obey the relation `0 <= i_idx < n_mems` (See XA_API_CMD_GET_N_MEMTABS)

2.6.10 XA_API_CMD_GET_MEM_INFO_TYPE

Table 2-23 XA_API_CMD_GET_MEM_INFO_TYPE command

Subcommand	Memory index
Description	This command gets the type of memory being referred to by the index.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_GET_MEM_INFO_TYPE`
	
	`i_idx`
	Index of the memory
	
	`pv_value`
	`&type` – Pointer to the memory type variable

Restrictions	None

Example

```
int index;
unsigned int type;
res = (*api_func)(api_obj,
      XA_API_CMD_GET_MEM_INFO_TYPE,
      index,
      (pVOID) &type);
```

Table 2-24 Memory Type Indices

Type	Description
XA_MEMTYPE_PERSIST	Persistent memory
XA_MEMTYPE_SCRATCH	Scratch memory
XA_MEMTYPE_INPUT	Input Buffer
XA_MEMTYPE_OUTPUT	Output Buffer

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_API_FATAL_INVALID_CMD_TYPE

  `i_idx` is an invalid memory block number; valid block numbers obey the relation `0 <= i_idx < n_mems` (See XA_API_CMD_GET_N_MEMTABS)

2.6.11 XA_API_CMD_GET_MEM_INFO_PRIORITY

Table 2-25 XA_API_CMD_GET_MEM_INFO_PRIORITY command

Subcommand	Memory index
Description	This command gets the allocation priority of memory being referred to by the index. (The meaning of the levels is defined on a codec-specific basis. This command returns a fixed dummy value unless the codec defines it otherwise.)

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_GET_MEM_INFO_PRIORITY`
	
	`i_idx`
	Index of the memory
	
	`pv_value`
	`&priority` – Pointer to the memory priority variable

Restrictions	None

Example

```
int index;
unsigned int priority;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_MEM_INFO_PRIORITY,
                  index,
                  (pVOID) &priority);
```

Table 2-26 Memory Priorities

Priority	Type
0	XA_MEMPRIORITY_ANYWHERE
1	XA_MEMPRIORITY_LOWEST
2	XA_MEMPRIORITY_LOW
3	XA_MEMPRIORITY_NORM
4	XA_MEMPRIORITY_ABOVE_NORM
5	XA_MEMPRIORITY_HIGH
6	XA_MEMPRIORITY_HIGHER
7	XA_MEMPRIORITY_CRITICAL

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_API_FATAL_INVALID_CMD_TYPE

  `i_idx` is an invalid memory block number; valid block numbers obey the relation `0 <= i_idx < n_mems` (See XA_API_CMD_GET_N_MEMTABS)

2.6.12 XA_API_CMD_SET_MEM_PTR

Table 2-27 XA_API_CMD_SET_MEM_PTR command

Subcommand	Memory index
Description	This command passes to the codec the pointer to the allocated memory. This is then stored in the memory tables structure allocated earlier. For the input and output buffers, it is legitimate to execute this command during the main codec loop.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_SET_MEM_PTR`
	
	`i_idx`
	Index of the memory
	
	`pv_value`
	`alloc` – Pointer to the memory buffer allocated

Restrictions	The pointer must be correctly aligned to the requirements.

Example

```
int index;
void * alloc; //alloc is a pointer to the aligned memory
res = (*api_func)(api_obj,
      XA_API_CMD_SET_MEM_PTR,
      index,
      (pVOID) alloc);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_API_FATAL_INVALID_CMD_TYPE

  `i_idx` is an invalid memory block number; valid block numbers obey the relation `0 <= i_idx < n_mems` (See XA_API_CMD_GET_N_MEMTABS)
- XA_API_FATAL_MEM_ALIGN

  `pv_value` is not of the required alignment for the requested memory block

2.6.13 XA_API_CMD_INPUT_OVER

Table 2-28 XA_API_CMD_INPUT_OVER command

Subcommand	None
Description	This command tells the codec that the end of the input data has been reached. This situation can arise both in the initialization loop and the execute loop.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_INPUT_OVER`
	
	`i_idx`
	`NULL`
	
	`pv_value`
	`NULL`

Restrictions	None

Example

```
res = (*api_func)(api_obj,
        XA_API_CMD_INPUT_OVER,
        0,
        NULL);
```

Errors

- Common API Errors

2.6.14 XA_API_CMD_SET_INPUT_BYTES

Table 2-29 XA_API_CMD_SET_INPUT_BYTES command

Subcommand	None
Description	This command sets the number of bytes available in the input buffer for the codec. It is used both in the initialization loop and execute loop. It is the number of valid bytes from the buffer pointer. It should be at least the minimum buffer size requested unless this is the end of the data.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_SET_INPUT_BYTES`
	
	`i_idx`
	`NULL`
	
	`pv_value`
	`&buff_size` – Pointer to the input byte variable

Restrictions	None

Example

```
int buff_size;
res = (*api_func)(api_obj,
       XA_API_CMD_SET_INPUT_BYTES,
       0,
       (pVOID) &buff_size);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.15 XA_API_CMD_GET_CURIDX_INPUT_BUF

Table 2-30 XA_API_CMD_GET_CURIDX_INPUT_BUF command

Subcommand	None
Description	This command gets the number of input buffer bytes consumed by the codec. It is used both in the initialization loop and execute loop.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_GET_CURIDX_INPUT_BUF`
	
	`i_idx`
	`NULL`
	
	`pv_value`
	`&bytes_consumed` – Pointer to the bytes consumed variable

Restrictions	None

Example

```
int bytes_consumed;
res = (*api_func)(api_obj,
        XA_API_CMD_GET_CURIDX_INPUT_BUF,
        0,
        (pVOID) &bytes_consumed);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.16 XA_API_CMD_EXECUTE

Table 2-31 XA_CMD_TYPE_DO_EXECUTE subcommand

Subcommand	`XA_CMD_TYPE_DO_EXECUTE`
Description	This command executes the codec.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_EXECUTE`
	
	`i_idx`
	`XA_CMD_TYPE_DO_EXECUTE`
	
	`pv_value`
	`NULL`

Restrictions	None

Example

```
res = (*api_func)(api_obj,
        XA_API_CMD_EXECUTE,
        XA_CMD_TYPE_DO_EXECUTE,
        NULL);
```

Errors

- Common API Errors
- See the codec-specific section for execution errors

Table 2-32 XA_CMD_TYPE_DONE_QUERY subcommand

Subcommand	`XA_CMD_TYPE_DONE_QUERY`
Description	This command checks to see if the end of processing has been reached. If it has, the flag value is set to 1; otherwise it is set to zero. The pointer to the flag is passed as an argument. Processing by the codec can continue for several invocations of the DO_EXECUTE command after the last input data has been passed to the codec, thus the application should not assume that the codec has finished generating all its output until so indicated by this command.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_EXECUTE`
	
	`i_idx`
	`XA_CMD_TYPE_DONE_QUERY`
	
	`pv_value`
	`&flag` – Pointer to the flag variable

Restrictions	None

Example

```
int flag;
res = (*api_func)(api_obj,
        XA_API_CMD_EXECUTE,
        XA_CMD_TYPE_DONE_QUERY,
        (pVOID) &flag);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.17 XA_API_CMD_GET_OUTPUT_BYTES

Table 2-33 XA_API_CMD_GET_OUTPUT_BYTES command

Subcommand	None
Description	This command obtains the number of bytes output by the codec during the last execution.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_GET_OUTPUT_BYTES`
	
	`i_idx`
	`NULL`
	
	`pv_value`
	`&out_bytes` – Pointer to the output bytes variable

Restrictions	None

Example

```
int out_bytes;
res = (*api_func)(api_obj,
        XA_API_CMD_GET_OUTPUT_BYTES,
        0,
        (pVOID) &out_bytes);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

3. HiFi DSP MP3 Encoder

The HiFi DSP MP3 Encoder conforms to the generic codec API. The flow chart of the command sequence used on the example test bench is shown in Figure 3.

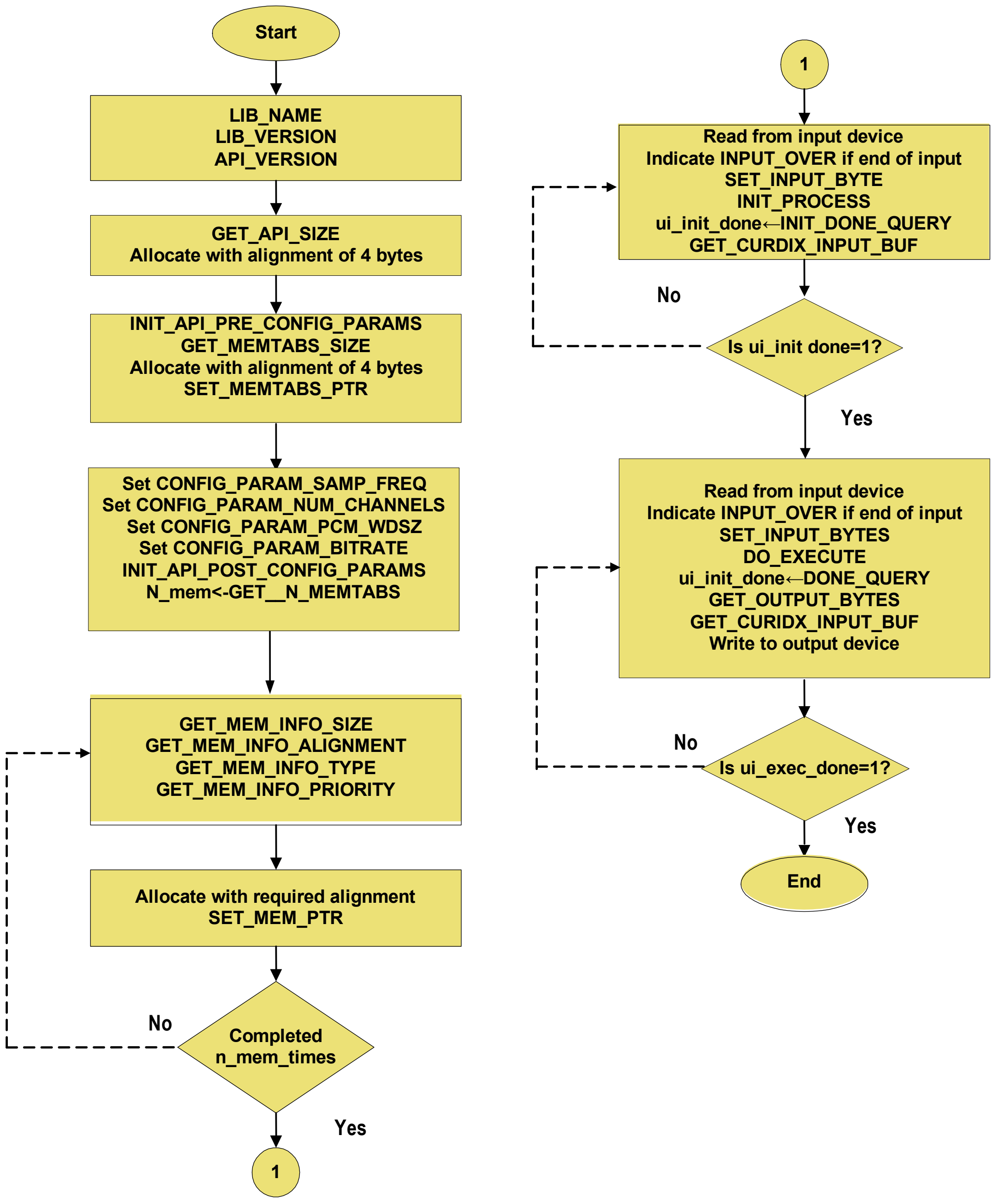

Figure 3 Flow Chart for MP3 Encoder Integration

3.1 Files Specific to the MP3 Encoder

The MP3 Encoder parameter header file (include/mp3_enc/):

- `xa_mp3_enc_api.h`

The MP3 Encoder library (lib):

- `xa_mp3_enc.a`

The MP3 Encoder API call is defined as:

```
XA_ERRORCODE xa_mp3_enc(xa_codec_handle_t p_xa_module_obj,
                         WORD32            i_cmd,
                         WORD32            i_idx,
                         pVOID             pv_value);
```

3.2 Configuration Parameters

The HiFi MP3 encode algorithm accepts the following parameters from the user.

These parameters must be supplied as there is no header information encoded in the PCM data provided through the input buffer. The application must determine these parameters from the source of the data. For example, the header data when using a '.wav' file.

- **`samp_freq`** – This is the sample frequency of the input samples. The units are Hz (samples per second). For example, a valid input is 44100.
- **`num_chan`** – This is the number of channels of data that are interleaved into the input buffer. There are only two valid values 1 (monaural) and 2 (stereo). In the case of stereo, the output data are interleaved in the output buffer with the first output sample in each pair being the left channel value and the second sample in each pair being the right channel value.
- **`pcm_wd_sz`** – This is the PCM word size of the data samples. The valid value is 16 only.

The following parameter is required for the encoding process:

- **`bit_rate`** – This should be set to a valid MP3 bit rate. If not the next higher valid rate is selected. Note the units are in kbps; 128 is a valid example.

The following parameter is provided by the encoder:

- **`bit_rate`** – This is the rounded up (if necessary) bit rate actually used during encoding.

3.3 MP3 Encoder-Specific Commands

These are the commands unique to the HiFi MP3 encoder. They are listed in sections based on their primary commands type (i_cmd). Each section contains a table for every subcommand. In the case of no subcommands, the one primary command is presented.

3.3.1 Initialization and Execution Errors

These errors can result from the initialization or execution API calls:

- XA_MP3ENC_CONFIG_NONFATAL_INVALID_BITRATE
  - During initialization the bit rate was adjusted upwards to the next valid value

3.3.2 XA_API_CMD_SET_CONFIG_PARAM

Table 3-1 XA_MP3ENC_CONFIG_PARAM_PCM_WDSZ subcommand

Subcommand	`XA_MP3ENC_CONFIG_PARAM_PCM_WDSZ`
Description	This command sets the PCM Word format size of the input samples in bits.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	
	`i_idx`
	`XA_MP3ENC_CONFIG_PARAM_PCM_WDSZ`
	
	`pv_value`
	`&pcm_wd_sz` – Pointer to sampling frequency variable

Restrictions	Valid only for 16.

Example

```
unsigned int pcm_wd_sz;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_MP3ENC_CONFIG_PARAM_PCM_WDSZ,
                  (void *) &pcm_wd_sz);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_MP3ENC_CONFIG_FATAL_PCM_WDSZ

  Value is not 16

Table 3-2 XA_MP3ENC_CONFIG_PARAM_SAMP_FREQ subcommand

Subcommand	`XA_MP3ENC_CONFIG_PARAM_SAMP_FREQ`
Description	Set the sampling frequency of the input PCM.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	
	`i_idx`
	`XA_MP3ENC_CONFIG_PARAM_SAMP_FREQ`
	
	`pv_value`
	`&samp_freq` – Pointer to sampling frequency variable

Restrictions	Valid for 16000, 22050, 24000, 32000, 44100 and 48000

Example

```
int samp_freq;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_MP3ENC_CONFIG_PARAM_SAMP_FREQ,
                  (void *) &samp_freq);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_MP3ENC_CONFIG_FATAL_SAMP_FREQ

  Value not valid

Table 3-3 XA_MP3ENC_CONFIG_PARAM_NUM_CHANNELS subcommand

Subcommand	`XA_MP3ENC_CONFIG_PARAM_NUM_CHANNELS`
Description	Set the number of interleaved channels in the PCM input buffer. When set to 2 channels, the Joint Stereo (MS) enabling decision is made internally depending upon the input data.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	
	`i_idx`
	`XA_MP3ENC_CONFIG_PARAM_NUM_CHANNELS`
	
	`pv_value`
	`&num_chan` – Pointer to number of channels variable

Restrictions	The value should be 1 or 2.

Example

```
int num_chan;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_MP3ENC_CONFIG_PARAM_NUM_CHANNELS,
                  (void *) &num_chan);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_MP3ENC_CONFIG_FATAL_NUM_CHANNELS

  Value not valid

Table 3-4 XA_MP3ENC_CONFIG_PARAM_BITRATE subcommand

Subcommand	`XA_MP3ENC_CONFIG_PARAM_BITRATE`
Description	This command sets the bit rate at which the current encoder operates.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	
	`i_idx`
	`XA_MP3ENC_CONFIG_PARAM_BITRATE`
	
	`pv_value`
	`&bit_rate` – Pointer to PCM word size

Restrictions	The value should be one of the values stated in references [1] and [2]. Otherwise, it is changed to the next higher bit rate and the caller is informed of the change through a non-fatal error. The changed value may be obtained after initialization. Note the units are in kHz e.g. 320.

Example

```
unsigned int bitrate;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_MP3ENC_CONFIG_PARAM_BITRATE,
                  (void *) &bitrate);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

3.3.3 XA_API_CMD_GET_CONFIG_PARAM

Table 3-5 XA_MP3ENC_CONFIG_PARAM_BITRATE subcommand

Subcommand	`XA_MP3ENC_CONFIG_PARAM_BITRATE`
Description	This command gets the bit rate at which the encoder is operating.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API structure
	
	`i_cmd`
	`XA_API_CMD_GET_CONFIG_PARAM`
	
	`i_idx`
	`XA_MP3ENC_CONFIG_PARAM_BITRATE`
	
	`pv_value`
	`&bitrate` – Pointer to bit rate variable

Restrictions	None

Example

```
unsigned int bitrate;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_MP3ENC_CONFIG_PARAM_BITRATE,
                  (void *) &bitrate);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

4. Introduction to the Example Test Bench

The supplied sample test bench consists of the following files:

- Test bench source file (found in `test/src`)
  - `xa_mp3_enc_error_handler.c`
  - `xa_mp3_enc_sample_testbench.c`
- Makefile to build the executable (`test/build`)
  - `makefile_testbench_sample`

4.1 Making the Executable

To build the application, follow these steps:

1. Go to `test/build`.
2. At the prompt, enter
   `xt-make -f makefile_testbench_sample clean all`

This will build the example application `xa_mp3_enc_test`.

Note: If you have source code distribution, you must build the MP3 Encoder library before building the test bench. To build the library, follow these steps:

1. Go to `build`.
2. At the prompt, enter
   `xt-make clean all install`

This will build the MP3 Encoder library `xa_mp3_enc.a` and copy it to the `lib` directory.

To build and execute the application from .xws based release package, please refer to the readme.html file available in the imported application project.

4.2 Usage

The sample application executable can be run with direct command line options or with a parameter file.

The command line usage is as follows:

```
xt-run xa_mp3_enc_test -ifile:<infile> -ofile:<outfile> \
            [-br:<bitrate>]
            [-help]
```

where,

- `<infile>` is the name of the mp3 input file.
- `<outfile>` is the name of the output file.
- `<bitrate>` is the bit rate for encoding `(default: 128)`.

Refer to the configuration parameter definitions in section 3.2 for a full description of their usage.

If no command line argument is given, the application reads the commands from the parameter file `paramfilesimple.txt` in same directory as the executable.

The syntax of `paramfilesimple.txt` file is

```
@Start
@Input_path <path to be pre-pended to all input files>
@Output_path <path to be pre-pended to all output files>
<command line 1>
<command line 2>
....
@Stop
```

The MP3 Encoder can be run for multiple test files using different command lines. The syntax for command lines in the parameter file is the same as the syntax for specifying options on the command line. All the @<*command*>s should be at the first column of a line except the `@New_line` command.

Note All the @<*command*>s should be at the first column of a line except the @New_line command.

Note All the @<*command*>s are case sensitive. If the command line in the parameter file has to be broken to two parts on two different lines, use the `@New_line` command.

Example:

```
<command line part 1> @New_line
<command line part 2>.
```

Note Blank lines will be ignored.

Note Individual lines can be commented out using "//" at the beginning of the line.

5. References

[1] *ISO/IEC 11172-3 Information technology -- Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s -- Part 3: Audio (MPEG-1)*

[2] *ISO/IEC 13818-3 Information technology -- Generic coding of moving pictures and associated audio information -- Part 3: Audio (MPEG-2)*