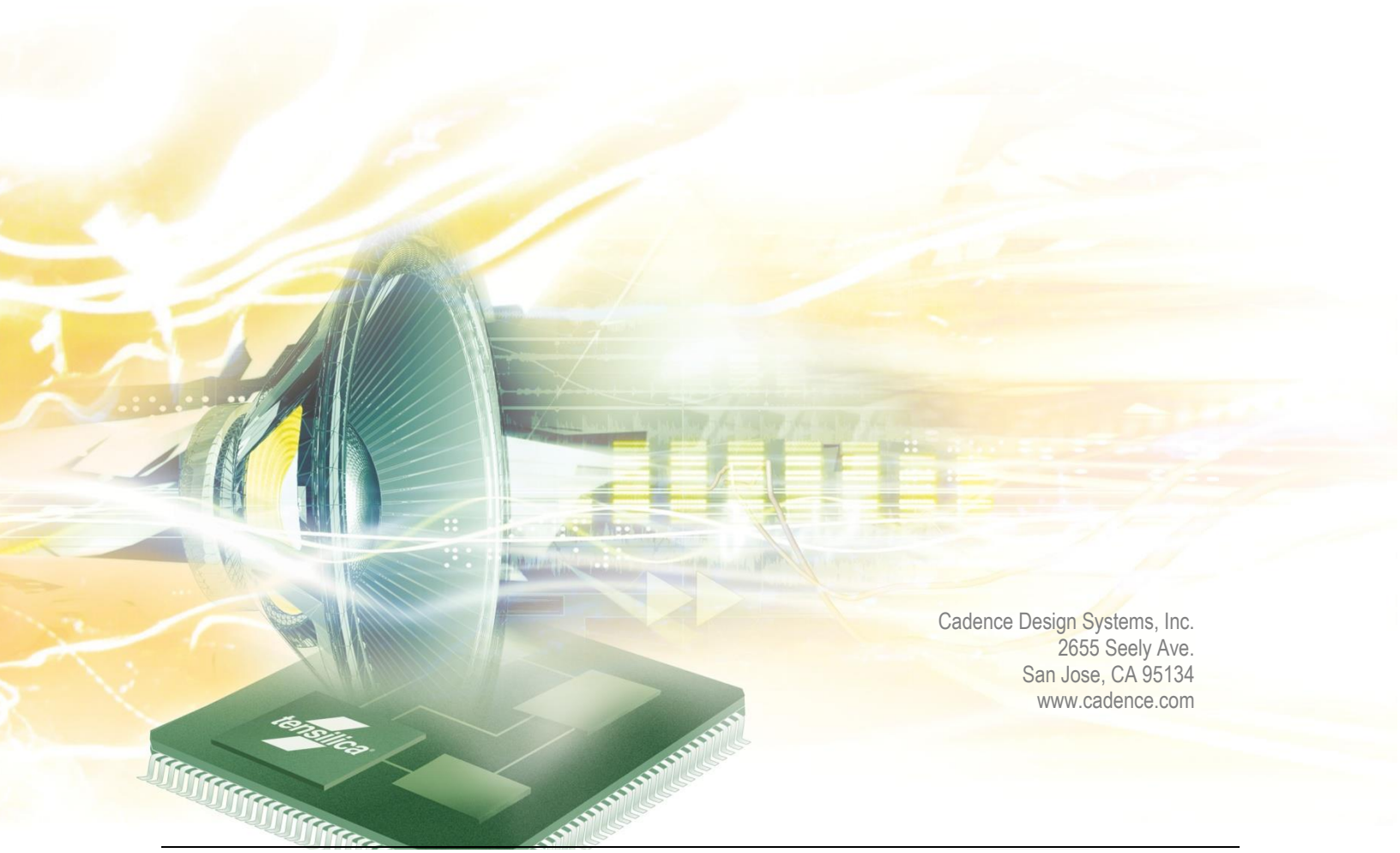
cadence®

MP3 Decoder

Programmer's Guide

For HiFi DSPs



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

Version 3.10
September 2017

Contents

Figures

Tables

Document Change History

Version	Changes
3.7	■ History section added. ■ Added memory and timing performance data for HiFi Mini and HiFi 3.
3.8	■ Updated memory and timing data.
3.9	■ Added memory and timing data for HiFi 4.
3.10	■ Added performance data for HiFi 3z in Section 1.4.

1. Introduction to the HiFi MP3 Decoder

The HiFi MP3 Decoder implements the audio codec standard specified as part of the MPEG-1, MPEG-2 and MPEG-2.5 specifications. MPEG-1 and MPEG-2 standards originate from the Moving Picture Experts Group (MPEG), a working group of ISO/IEC [1] [2]. MPEG-2.5 is a proprietary extension to MPEG-1/2 Layer III by the Fraunhofer Institute for Integrated Circuits (FhG IIS), Germany.

The HiFi MP3 Decoder is a general term for three different levels of MPEG-1/2/2.5 audio technology supplied by Tensilica as separate libraries:

- The MP2 Decoder library (`mp2_dec`) implements MPEG-1/2 Layers I and II 2-channel decoding.
- The MP3 Decoder library (`mp3_dec`) implements MPEG-1/2/2.5 Layers I, II and III 2-channel decoding.
- The MP3 5.1-Channel Decoder library (`mp3mch_dec`) implements MPEG-1/2/2.5 Layers I, II and III 2-channel decoding as well as MPEG-2 Layer II 5.1-channel decoding.

1.1 MP3 Description

Quote from the Moving Picture Experts Group:

”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.”

MP3 was extended by MPEG-2 to support further sampling rates and encoded bit rates. The first of the MP3 players arrived in 1998. The success was due to the compression achieved with respect to the CD format.

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

The codec handles audio signals sampled in the range of 8 kHz to 48 kHz. It operates on a frame of 1152 samples (Layers II and III) or 384 samples (Layer I).

1.2 Document Overview

This document covers 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 are common to all the HiFi codecs; these are described after the introduction. The next section covers all the features and information particular to the HiFi MP3 Decoder. Finally, the example test benches are described.

1.3 HiFi MP3 Decoder Specifications

The HiFi DSP MP3 Decoder from Cadence Tensilica is a full-precision MP3 decoder that supports MPEG-1 and MPEG-2 Layer I, II and III and MPEG-2.5 Layer III audio streams. It implements the following features in conformance to the MPEG-1, MPEG-2, and MPEG 2.5 specifications. Cadence Audio Codec API is used in all 3 libraries.

MPEG-1

- Compliant with ISO/IEC 11172-3
  - MPEG-1 Layers I and II decoding
  - MPEG-1 Layer III decoding (`mp3_dec` and `mp3mch_dec` only)
- Sample rates: 32, 44.1 and 48 kHz
- Bit rates 32-320 Kbps for Layer III
- Bit rates 32-384 Kbps for Layer II
- Bit rates 32-448 Kbps for Layer I
- Supports free format bit-rate decoding for Layer III
- Mono and stereo channels
- Joint stereo (MS and Intensity Stereo)
- Support for VBR
- CRC check support for Layers I and II
- Full precision decoder

MPEG-2

In addition to MPEG-1,

- Compliant with ISO/IEC 13818-3
  - Main audio channels (Lo/Ro) decoding of MPEG-2 Layers I and II
  - Main audio channels (Lo/Ro) decoding of MPEG-2 Layer III (`mp3_dec` and `mp3mch_dec` only)
  - 5.1-channel decoding of MPEG-2 Layer II (`mp3mch_dec` only)
- Additional sample rates: 16, 22.05 and 24 kHz
- Additional bit rates 8 -160 Kbps for Layer III applicable to lower sample rates
- Additional bit rates 8 -160 Kbps for Layer II applicable to lower sample rates
- Additional bit rates 32-256 Kbps for Layer I applicable to lower sample rates
- Bit rates up to 1067 Kbps for Layer II 5.1-Channel decoding
- Support for VBR

MPEG-2.5

In addition to MPEG-2,

- Decoding of MPEG-2.5 Layer III (`mp3_dec` and `mp3mch_dec` only)
- Sample rates: 8, 11.025 and 12 kHz
- Bit rates 8-160 kbps for Layer III
- Support for VBR

1.4 HiFi MP3 Decoder Performance

The HiFi DSP MP3 Decoder from Cadence was characterized on the HiFi 5-stage DSP. The memory usage and performance figures are provided for design reference.

1.4.1 MP2 Decoder Memory

Text (Kbytes)					Data (Kbytes)	
HiFi Mini	HiFi 2	HiFi 3	HiFi 3z	HiFi 4	HiFi Mini/2	HiFi 3/3z/4
10.7	12.4	11.7	12.5	12.7	3.9	3.9

Run Time Memory (Kbytes)				
Persistent	Scratch	Stack	Input	Output
7.0	1.9	0.4	2.0	4.5

1.4.2 MP2 Decoder Timings

Rate	Channels	Bit Rate	Average CPU Load (MHz)				
kHz		kbps	HiFi Mini	HiFi 2	HiFi 3	HiFi 3z	HiFi 4
48	2	192	4.8	4.9	3.8	3.3	2.9

1.4.3 MP3 Decoder Memory

Text (Kbytes)					Data (Kbytes)	
HiFi Mini	HiFi 2	HiFi 3	HiFi 3z	HiFi 4	HiFi Mini/2	HiFi 3/3z/4
22.4	26.3	24.7	26.3	27.1	16.0	14.8

Run Time Memory (Kbytes)				
Persistent	Scratch	Stack	Input	Output
12.2	7.0	0.8	2.0	4.5

1.4.4 MP3 Decoder Timings

Rate	Channels	Bit Rate	Average CPU Load (MHz)				
kHz		kbps	HiFi Mini	HiFi 2	HiFi 3	HiFi 3z	HiFi 4
44.1	2	128	4.6	4.8	3.5	3.1	2.8
44.1	2	320	5.6	5.9	4.2	3.8	3.5
48	2	128	4.8	5.0	3.6	3.2	2.9
48	2	320	6.1	6.4	4.6	4.1	3.7

1.4.5 MP3 5.1-Channel Decoder Memory

Text (Kbytes)					Data (Kbytes)	
HiFi Mini	HiFi 2	HiFi 3	HiFi 3z	HiFi 4	HiFi Mini/2	HiFi 3/3z/4
37.6	42.3	40.8	43.3	44.3	16.3	15.1

Run Time Memory (Kbytes)				
Persistent	Scratch	Stack	Input	Output
21.3	36.5	1.2	4.0	13.5

1.4.6 MP3 5.1-Channel Decoder Timings

Rate	Channels	Bit Rate	Average CPU Load (MHz)				
kHz		kbps	HiFi Mini	HiFi 2	HiFi 3	HiFi 3z	HiFi 4
48	6	1066	27.0	27.6	24.6	20.4	19.9

Note Performance specification measurements are carried on a cycle-accurate simulator assuming an ideal memory system, *i.e.*, 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-warmed cache model. The MCPS numbers for HiFi Mini are obtained by running the test that is recompiled from the HiFi 2 source code in the HiFi Mini configuration. The MCPS numbers for HiFi 3z/HiFi 4 are obtained by running the test that is recompiled from the HiFi 3 source code in the HiFi 3z/HiFi 4 configuration. No specific optimization is done for HiFi Mini/HiFi 3z/HiFi 4. However, specific optimization is done for HiFi 3, hence its MCPS numbers are fairly lower. Note that the output of the HiFi 3/HiFi 3z/HiFi 4 code is not bit-exact with that of the HiFi 2/HiFi Mini code.

Note Output buffer requirements are specified for 16-bit output, which is typically used. For 24-bit output, the output buffer size is doubled.

2. Generic HiFi Audio Codec API

This chapter describes the API which 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.

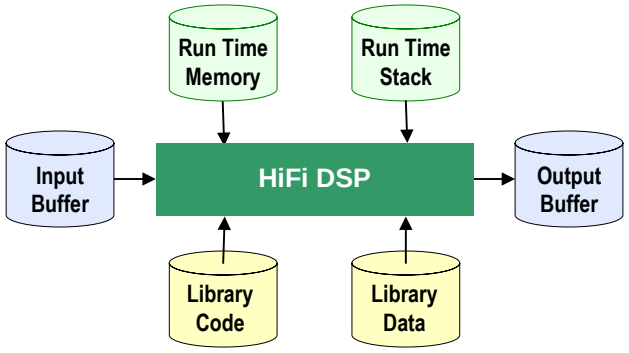


Figure 1 **HiFi Audio Codec Interfaces**

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 be 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 run time 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.

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.

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.

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.

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, *i.e.*, 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.

Input Buffer

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

From API Version 1.16 or later, the input buffer can be partially filled before the call to the codec. The codec returns a non-fatal error indicating insufficient data if data in the input buffer is not enough to decode PCM samples.

Output Buffer

This is the buffer in which the algorithm writes the output. This buffer needs to 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 the same and are specified in the 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 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 state structure

Returns	Error Code based on the success or failure of 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 are defined in the following table.

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 sub divided into those that are fatal, which require the restarting of the whole 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 command sequence overview flow chart below. 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.

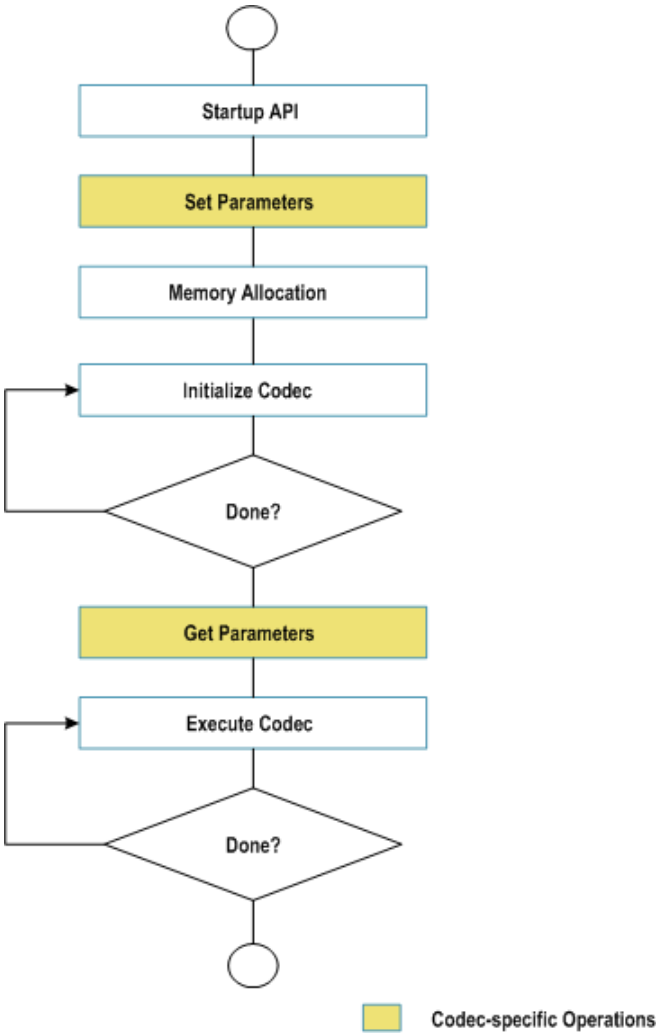


Figure 2 **API Command Sequence Overview**

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 2-2 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 2-3 Commands for Setting Parameters

Command / Subcommand	Description
XA_API_CMD_SET_CONFIG_PARAM XA_<*codec*>_CONFIG_PARAM_<*param_name*>	Set 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. Once 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 2-4 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 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 2-5 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 encode 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 2-6 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`	Signals 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 2-7 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 2-8 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 4th 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 and they 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 will also cover the “Execution” errors that occur during the initialization or execution calls to the API.

2.6.1 Common API Errors

All 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 2-9 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 2-10 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 2-11 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 2-12 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 API size variable

Restrictions	The application shall 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 2-13 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 parameter 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 2-14 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 2-15 XA_CMD_TYPE_INIT_PROCESS subcommand

Subcommand	XA_CMD_TYPE_INIT_PROCESS
Description	This command initializes the codec. In the case of a decoder, it searches for the valid header and performs the header decoding to get the encoded stream parameters. This command is part of the initialization loop. It must be repeatedly called until the codec signals it has finished. In the case of an encoder, the initialization of codec is performed. 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 2-16 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 else 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 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 2-17 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 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 2-18 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 shall 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 2-19 XA_API_CMD_GET_N_MEMTABS command

Subcommand	None
Description	This command is used to obtain 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 2-20 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 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 2-21 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 2-22 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 2-23 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 2-24 XA_API_CMD_GET_MEM_INFO_PRIORITY command

Subcommand	Memory index
Description	This command gets 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 2-25 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 2-26 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 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 2-27 XA_API_CMD_INPUT_OVER command

Subcommand	None
Description	This command is used to tell 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 2-28 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 2-29 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 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 2-30 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 2-31 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 is, the flag value is set to 1 else 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, so 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

Table 2-32 XA_CMD_TYPE_DO_RUNTIME_INIT subcommand

Subcommand	XA_CMD_TYPE_DO_RUNTIME_INIT
Description	This command resets the decoder's history buffers. It can be used to avoid distortions and clicks by facilitating playback ramping up and down during trick-play. Note: This command is available in API version 1.14 or later.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_EXECUTE i_idx XA_CMD_TYPE_DO_RUNTIME_INIT pv_value **NULL**
Restrictions	None

Example

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

Errors

- Common API Errors

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`

2.6.18 XA_API_CMD_GET_CONFIG_PARAM

Table 2-34 XA_CONFIG_PARAM_CUR_INPUT_STREAM_POS subcommand

Subcommand	`XA_CONFIG_PARAM_CUR_INPUT_STREAM_POS`

Description	This command reads the current input stream position, which is equal to the total number of consumed input bytes until the start of the input buffer. This running counter is set to zero at library initialization time and incremented every time the codec library consumes any bytes from the input buffer. If the application layer places a unit of input data with a byte size equal to `size` at byte offset `offset` in the input buffer, then the input stream position range for this unit may be calculated as follows:
	`start_pos = CUR_INPUT_STREAM_POS + offset`
	`end_pos = CUR_INPUT_STREAM_POS + offset + size`

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_GET_CONFIG_PARAM`
	
	`i_idx`
	`XA_CONFIG_PARAM_CUR_INPUT_STREAM_POS`
	
	`pv_value`
	`&ui_cur_input_stream_pos` – Pointer to the current input stream position variable

Restrictions	The current input stream position counter is 32-bits and, therefore, will overflow and wrap-around if the input stream length is more than 2^{32}-1 bytes. This command is available in API version 1.15 or later.

Example

```
unsigned int ui_cur_input_stream_pos;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_CONFIG_PARAM_CUR_INPUT_STREAM_POS,
                  (void *) &ui_cur_input_stream_pos);
```

Errors

- Common API Errors

Table 2-35 XA_CONFIG_PARAM_GEN_INPUT_STREAM_POS subcommand

Subcommand	XA_CONFIG_PARAM_GEN_INPUT_STREAM_POS
Description	This command reads the input stream position of the unit (e.g., frame) corresponding to the generated (decoded or encoded) output data block. That is, if the main processing (DO_EXECUTE) call into the library generates any data in the output buffer, then this command reads the total number of input bytes consumed until the start of the unit that has been processed and placed into the output buffer. For example, if the application layer places a unit in the input buffer at input stream position start_pos (see Table 2-34), when the library generates the decoded or encoded data corresponding to this unit, it sets GEN_INPUT_STREAM_POS to start_pos.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_GET_CONFIG_PARAM i_idx XA_CONFIG_PARAM_GEN_INPUT_STREAM_POS pv_value &ui_gen_input_stream_pos – Pointer to the input stream position of the generated data variable
Restrictions	The input stream position of the generated data counter is 32-bits and, therefore, will overflow and wrap-around if the input stream length is more than 2^{32}-1 bytes. This command is available in API version 1.15 or later.

Example

```
unsigned int ui_gen_input_stream_pos;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_CONFIG_PARAM_GEN_INPUT_STREAM_POS,
                  (void *) &ui_gen_input_stream_pos);
```

Errors

- Common API Errors

2.6.19 XA_API_CMD_SET_CONFIG_PARAM

Table 2-36 XA_CONFIG_PARAM_CUR_INPUT_STREAM_POS subcommand

Subcommand	XA_CONFIG_PARAM_CUR_INPUT_STREAM_POS
Description	This command resets the current input stream position. See Table 2-34 for details.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_SET_CONFIG_PARAM i_idx XA_CONFIG_PARAM_CUR_INPUT_STREAM_POS pv_value &ui_cur_input_stream_pos – Pointer to the current input stream position variable
Restrictions	This command is available in API version 1.15 or later.

Example

```
unsigned int ui_cur_input_stream_pos = 0;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_CONFIG_PARAM_CUR_INPUT_STREAM_POS,
                  (void *) &ui_cur_input_stream_pos);
```

Errors

- Common API Errors

3. HiFi DSP MP3 Decoder

The HiFi DSP MP3 Decoder conforms to the generic codec API. The following flow chart shows the command sequence used in the example test bench.

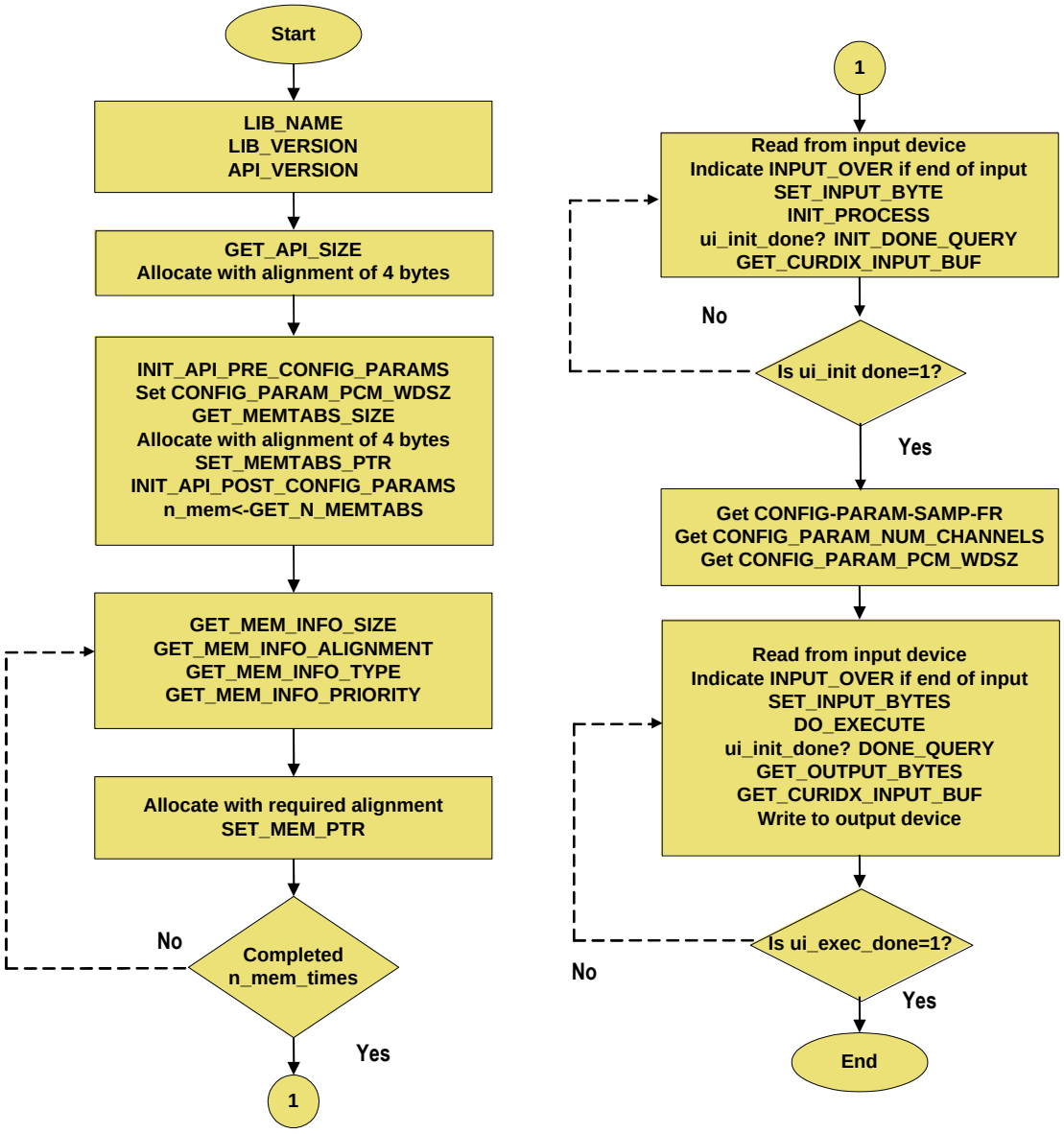


Figure 3 Flow Chart for MP3 Decoder Integration

3.1 Files Specific to the MP3 Decoder

The MP3 decoder parameter header file (`include/mp3_dec`)

- `xa_mp3_dec_api.h`

The MP3 decoder libraries (`lib`):

- `xa_mp2_dec.a` (MP2 Decoder)
- `xa_mp3_dec.a` (MP3 Decoder)
- `xa_mp3mch_dec.a` (MP3 5.1-Channel Decoder)

The MP3 decoder API call is defined as:

```
XA_ERRORCODE xa_mp3_dec(xa_codec_handle_t p_xa_module_obj,
                        WORD32            i_cmd,
                        WORD32            i_idx,
                        pVOID             pv_value);
```

3.2 Configuration Parameters

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

- **`pcm_wd_sz`** – The PCM word size requested.
  - Smaller or equal to 16 – Value used is 16
  - Greater than 16 – Value used is 24
- **`crc_check`** – Enable (non-zero) or disable (0) CRC validation for error-protected Layer I and II streams. If CRC validation is enabled and fails for an error-protected stream, a non-fatal error code is returned along with the decoded output. By default, CRC checking is disabled.
- **`mch_enable`** – Enable (non-zero) or disable (0) Layer II 5.1-channel decoding. This flag is supported by the MP3 5.1-Channel Decoder only and is enabled by default.

`dabmp2` – Enable (non-zero) or disable (0) DAB specific Scale factor CRC (see *ETSI EN 300 401 V1.4.1 Radio Broadcasting Systems* [3]).

The parameters required for storage and audio output through a DAC are obtained from the bitstream during the initialization stage. After the initialization the following parameters are available:

- **samp_freq** – This is the sample frequency of the output buffer samples. The units are Hz (samples per second). For example, a valid output is 44100.
- **num_chan** – This is the number of channels of data put into the output buffer.
  - In the stereo-only libraries (`mp2_dec` and `mp3_dec`), there are only two valid values 1 (mono) and 2 (stereo) for this parameter. 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.
  - In the MP3 5.1-Channel Decoder, the value of this parameter is between 1 and 6. The PCM samples for the separate channels are interleaved in the output buffer – the exact channel mapping can be retrieved through the `chan_map` parameter. Note that the `num_chan` parameter indicates the actual number of channels present in the output buffer, i.e., there will be no unused PCM sample offsets in the output buffer.
- **chmode_info** – This is the channel mode information:
  - bits 1:0 – 0 (stereo), 1 (joint stereo), 2 (dual mono), 3 (mono)
  - bit 4: intensity stereo, if set to 1
  - bit 5: mid-side stereo, if set to 1
- **bitrate** – This is the data rate (bit rate) of the input bitstream. The units are kbps (kbits per second). For example, a valid output is 320.
- **pcm_wd_sz** – This is the PCM word size of the data in the output buffer. The only values are 16 or 24. This is only useful if neither 16 nor 24 were given to the decoder.
- **mch_status** – This is the multi-channel status of the library and the input stream. Status codes:
  - `XA_MP3DEC_MCH_STATUS_UNSUPPORTED` (0) – the library does not support multi-channel decoding (i.e., the library is either `mp2_dec` or `mp3_dec`).
  - `XA_MP3DEC_MCH_STATUS_DISABLED` (1) – multi-channel decoding is disabled through the `mch_enable` configuration parameter (`mp3mch_dec` only).
  - `XA_MP3DEC_MCH_STATUS_NOT_PRESENT` (2) – multi-channel decoding is supported but the input stream is mono or stereo (`mp3mch_dec` only).
  - `XA_MP3DEC_MCH_STATUS_PRESENT` (3) – the library is decoding multi-channel content (`mp3mch_dec` only).
- **extn_present** – This indicates if an extension substream is present in the input stream (`mp3mch_dec` only).
- **lfe_present** – This indicates if the LFE (low-frequency effects) channel is present in the input multi-channel stream (`mp3mch_dec` only).
- **num_xchan** – This is the number of extra full-bandwidth channels in addition to the main audio channels. This number does not include the LFE channel.
  - Valid values: 0 through 3

  - For example, a 5.1-Channel stream will have `num_xchan` set to 3 and `lfe_present` set to 1.

- **`chan_config`** – This specifies the channel configuration of the input stream (`mp3mch_dec` only). Valid values and channel configurations:
  - 1 – L(Left) / C (Center) / R (Right) / Ls (Left Surround) / Rs (Right Surround)
  - 2 – L / C / R/ Cs (Center Surround)
  - 3 – L / C / R
  - 4 – L / C / R / L2 (Second Left) / R2 (Second Right)
  - 5 – L / R / Ls / Rs
  - 6 – L / R / Cs
  - 7 – L / R
  - 8 – L / R / L2 / R2
  - 9 – C
  - 10 – C / L2 / R2

- **`chan_map`** – This parameter specifies how the channels are arranged in the output buffer. Note that the values of the `num_chan`, `lfe_present`, `num_xchan` and `chan_config` parameters can be derived from this parameter.
  - Nibbles 0 to 5 of the `chan_map` parameter variable are set to channel index values based on their sample offsets in the interleaved output PCM buffer. If a channel (with channel index C) appears at sample offset N in the interleaved output PCM buffer, then N^{th} nibble of the `chan_map` parameter is set to C.
  - Valid channel indices: 0 – L, 1 – C, 2 – R, 3 – Ls/Cs, 4 – Rs, 5 – L2, 6 – R2, 7 – LFE, 0xF – invalid PCM sample offset.
  - For example, `chan_map = 0xFF743120` indicates that the left channel (channel index 0) is present at sample offset 0, the right channel (channel index 2) is present at sample offset 1, the center channel (channel index 1) is present at sample offset 2, the left surround (channel index 3) is present at sample offset 3, the right surround (channel index 4) is present at sample offset 4, and the LFE channel (channel index 7) is present at sample offset 5.

3.3 Usage Notes

Although the HiFi MP3 Decoder conforms to the generic codec API described in Section 2, the following notes must be taken into account to ensure correct decoder operation.

- The MP3 5.1-Channel Decoder can process both base and extension input streams. If an extension stream is present, the input to the decoder must be a frame-by-frame concatenation of the base and extension streams:
  - [base frame 0] [extension frame 0] [base frame 1] [extension frame 1] …

- The PCM samples for the separate channels are interleaved in the output buffer. The number of channels present in the output buffer depends on the number of channels present in the input stream. There are no unused PCM offsets in the output buffer.
- For each DO_EXECUTE call, the MP3 Decoder expects enough data in the input buffer to parse and decode one complete frame. If there is insufficient data in the input buffer, the MP3 Decoder does not consume any bytes and returns a non-fatal error that allows the application to fill in additional data before calling the decoder again. Note that the MP3 5.1-Channel Decoder consumes both the base frame and the extension frame (if present) and produces the decoded audio data in a single DO_EXECUTE call.
- During initialization, the MP3 5.1-Channel Decoder detects availability of multi-channel content in the input stream by doing a CRC check on the base and extension frames (if present). If the CRC check fails, the MP3 5.1-Channel Decoder indicates through the `mch_status` parameter that multi-channel content is not present and switches to 2-channel decoding only.

3.4 MP3 Decoder-Specific Commands

These are the commands unique to the HiFi MP3 Decoder. 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.

3.4.1 Initialization and Execution Errors

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

- XA_MP3DEC_CONFIG_NONFATAL_MCH_NOT_SUPPORTED

  The application tried to enable multi-channel support in the MP2 Decoder or the MP3 Decoder libraries. Layer II 5.1-Channel decoding is supported only by the MP3 5.1-Channel Decoder library.

- XA_MP3DEC_CONFIG_NONFATAL_INVALID_GEN_STRM_POS

  The input stream position of the decoded data is invalid. This may occur if the decoder has not generated any output data, or a runtime-init ramp-down frame is active, or the current input stream position is reset by the application.

- XA_MP3DEC_EXECUTE_NONFATAL_NEED_MORE

  The codec requires more bits to continue the processing.

- XA_MP3DEC_EXECUTE_NONFATAL_CANNOT_REWIND

  The MP3 bitstream syntax states that some bits corresponding to the current frame can be contained in the space designated for the previous frame. This error situation occurs when the bitstream needs to be rewound beyond the buffered bits.

- XA_MP3DEC_EXECUTE_NONFATAL_CHANGED_CHANNELS

  The number of channels changed between successive frames.

- XA_MP3DEC_EXECUTE_NONFATAL_CHANGED_SAMP_FREQ

  The sample frequency changed between successive frames.

- XA_MP3DEC_EXECUTE_NONFATAL_CHANGED_LAYER

  The MPEG layer of the bitstream changed between successive frames

- XA_MP3DEC_EXECUTE_NONFATAL_NEXT_SYNC_NOT_FOUND

  The bytes required for the current frame exceeded the buffer limit.

- XA_MP3DEC_EXECUTE_NONFATAL_CRC_FAILED

  CRC validation failed for an error-protected stream.

- XA_MP3DEC_EXECUTE_NONFATAL_NO_MAIN_AUDIO_INPUT

  Insufficient data present to decode main audio data. This error may occur at broken frame boundaries or during trick play where input frames are not time continuous.

- XA_MP3DEC_EXECUTE_NONFATAL_MCH_CRC_ERROR

  The CRC check of the multi-channel frame failed during normal decoding. Only the stereo channels will be decoded and the rest will be set to 0. This error can be returned only by the MP3 5.1-Channel Decoder.

- XA_MP3DEC_EXECUTE_NONFATAL_MCH_EXT_NOTFOUND

  The CRC check of the extension multi-channel frame failed during normal decoding. Only the stereo channels will be decoded and the rest will be set to 0. This error can be returned only by the MP3 5.1-Channel Decoder.

- XA_MP3DEC_EXECUTE_NONFATAL_SCF_CRC_FAILED1

  The DAB-specific scale factor CRC check failed. 1 CRC is in error.

- XA_MP3DEC_EXECUTE_NONFATAL_SCF_CRC_FAILED2

  The DAB-specific scale factor CRC check failed. 2 CRC's are in error.

- XA_MP3DEC_EXECUTE_NONFATAL_SCF_CRC_FAILED3

  The DAB-specific scale factor CRC check failed. 3 CRC's are in error.

- XA_MP3DEC_EXECUTE_NONFATAL_SCF_CRC_FAILED4

  The DAB-specific scale factor CRC check failed. 4 CRC are in error.

- XA_MP3DEC_EXECUTE_NONFATAL_INVALID_BITRATE_MODE_COMB

  The MP3 bitstream syntax states that for layer II, not all combinations of total bit rate and mode are allowed.

Fatal errors require the codec to be completely reinitialized and presented with an alternative bitstream.

- XA_MP3DEC_EXECUTE_FATAL_UNSUPPORTED_LAYER

  A bitstream is presented with an unsupported MPEG layer

- XA_MP3DEC_EXECUTE_FATAL_OVERLOADED_IN_BUF

  The decoder attempted to decoding the bitstream beyond the available data

- XA_MP3DEC_EXECUTE_FATAL_STREAM_ERROR

  Errors were encountered in the bitstream

3.4.2 XA_API_CMD_SET_CONFIG_PARAM

Table 3-1 XA_MP3DEC_CONFIG_PARAM_PCM_WDSZ subcommand

Subcommand	XA_MP3DEC_CONFIG_PARAM_PCM_WDSZ
Description	This command sets the PCM Word format size of the output samples in bits. The word format is set to 16 bits or 24 bits.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_SET_CONFIG_PARAM i_idx XA_MP3DEC_CONFIG_PARAM_PCM_WDSZ pv_value &pcm_wd_sz – Pointer to word size variable
Restrictions	Only 16 and 24 are valid internal word format word sizes. 16, if pcm_wd_sz <= 16 24, if pcm_wd_sz > 16 See non-fatal errors below.

Note The 24 bit format is returned in 32 bit aligned words with the data left justified in the word. This is the reason that the buffer size requirement for 24-bit samples is double the requirement for 16-bit samples.

Example

```
int pcm_wd_sz = 24;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_MP3DEC_CONFIG_PARAM_PCM_WDSZ,
                  (pVOID) &pcm_wd_sz);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_MP3DEC_CONFIG_NONFATAL_MP3_PCM_ADJUST_16

  The value given has been adjusted to 16
- XA_MP3DEC_CONFIG_NONFATAL_MP3_PCM_ADJUST_24

  The value given has been adjusted to 24

Table 3-2 XA_MP3DEC_CONFIG_PARAM_CRC_CHECK subcommand

Subcommand	`XA_MP3DEC_CONFIG_PARAM_CRC_CHECK`
Description	This command enables or disables CRC validation for error-protected Layer I and II streams. During the execution phase, CRC failures are signaled to the application through a non-fatal error code (see Section 3.4.1).

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	
	`i_idx`
	`XA_MP3DEC_CONFIG_PARAM_CRC_CHECK`
	
	`pv_value`
	`&crc_check` – Pointer to CRC check variable

Restrictions	0 – disable CRC validation (default); non-zero – enable CRC validation.

Example

```
int crc_check = 1;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_MP3DEC_CONFIG_PARAM_CRC_CHECK,
                  (pVOID) &crc_check);
```

Errors

- Common API Errors

Table 3-3 XA_MP3DEC_CONFIG_PARAM_MCH_ENABLE subcommand

Subcommand	XA_MP3DEC_CONFIG_PARAM_MCH_ENABLE
Description	This command enables or disables multi-channel decoding in the MP3 5.1-Channel Decoder library.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_SET_CONFIG_PARAM i_idx XA_MP3DEC_CONFIG_PARAM_MCH_ENABLE pv_value &mch_enable – Pointer to multi-channel enable flag.
Restrictions	This flag is supported by the MP3 5.1-Channel Decoder library only. 0 – disable multi-channel decoding; non-zero – enable multi-channel decoding (default).

Example

```
int mch_enable = 0;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_MP3DEC_CONFIG_PARAM_MCH_ENABLE,
                  (pVOID) &mch_enable);
```

Errors

- Common API Errors

Table 3-4 XA_MP3DEC_CONFIG_PARAM_DAB_MP2 subcommand

Subcommand	XA_MP3DEC_CONFIG_PARAM_DAB_MP2
Description	This command enables or disables DAB-specific Scale Factor CRC checking.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_SET_CONFIG_PARAM i_idx XA_MP3DEC_CONFIG_PARAM_DAB_MP2 pv_value & dabmp2 – Pointer to dabmp2 enable flag
Restrictions	0 – disable DAB-specific Scale Factor CRC validation (default); non-zero – enable DAB-specific Scale Factor CRC validation

Example

```
int dabmp2 = 0;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_MP3DEC_CONFIG_PARAM_MCH_ENABLE,
                  (pVOID) &dabmp2);
```

Errors

- Common API Errors

Table 3-5 `XA_MP3DEC_CONFIG_PARAM_ACTIVATE_VLC_REWIND` subcommand

Subcommand	`XA_MP3DEC_CONFIG_PARAM_ACTIVATE_VLC_REWIND`
Description	This command enables or disables VLC rewind. This is a method to recover from bitstream overrun caused by a bitstream error during the VLC decoding process. If this feature is disabled, the decoder will drop the frame.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	
	`i_idx`
	`XA_MP3DEC_CONFIG_PARAM_ACTIVATE_VLC_REWIND`
	
	`pv_value`
	`& activate_vlc_rewind` – Pointer to `activate_vlc_rewind` flag.

Restrictions	0 – disable (default) 1 – enable.

Example

```
Int activate_vlc_rewind = 0;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_MP3DEC_CONFIG_PARAM_ACTIVATE_VLC_REWIND,
                  (pVOID) & activate_vlc_rewind);
```

Errors

- Common API Errors

3.4.3 XA_API_CMD_GET_CONFIG_PARAM

Table 3-6 XA_MP3DEC_CONFIG_PARAM_SAMP_FREQ subcommand

Subcommand	XA_MP3DEC_CONFIG_PARAM_SAMP_FREQ
Description	This command gets the sampling frequency from the codec. The sampling frequency can be used for example to write a '.wav' file header or to initialize a DAC or sample rate converter for real-time playout. The sampling frequency in Hz is returned.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_GET_CONFIG_PARAM i_idx XA_MP3DEC_CONFIG_PARAM_SAMP_FREQ pv_value &samp_freq – Pointer to sampling frequency variable
Restrictions	None

Example

```
int samp_freq;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_MP3DEC_CONFIG_PARAM_SAMP_FREQ,
                  (pVOID) &samp_freq);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  pv_value is NULL

Table 3-7 XA_MP3DEC_CONFIG_PARAM_NUM_CHANNELS subcommand

Subcommand	XA_MP3DEC_CONFIG_PARAM_NUM_CHANNELS
Description	This command gets the number of channels in the decoded bitstream. This information can be used to write a '.wav' file header or to initialize a DAC or sample rate converter for real-time playout.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_GET_CONFIG_PARAM i_idx XA_MP3DEC_CONFIG_PARAM_NUM_CHANNELS pv_value &num_chan – Pointer to number of channels
Restrictions	None

Example

```
int num_chan;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_MP3DEC_CONFIG_PARAM_NUM_CHANNELS,
                  (pVOID) &num_chan);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  pv_value is NULL

Table 3-8 XA_MP3DEC_CONFIG_PARAM_CHMODE_INFO subcommand

Subcommand	XA_MP3DEC_CONFIG_PARAM_CHMODE_INFO
Description	This command gets the channel mode information present in the frame header. The information can be used to derive the channel mode (nibble 0) and the mode extension values (nibble 1). The channel mode information is decoded as specified: Bits 1:0 – 0 (stereo), 1 (joint stereo), 2 (dual mono), 3 (mono) Bit 4 – intensity stereo, if set to 1 Bit 5 – mid-side stereo, if set to 1
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_GET_CONFIG_PARAM i_idx XA_MP3DEC_CONFIG_PARAM_CHMODE_INFO pv_value &chmode_info – Pointer to the channel mode variable
Restrictions	None

Example

```
int chmode_info;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_MP3DEC_CONFIG_PARAM_CHMODE_INFO,
                  (pVOID) &chmode_info);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  pv_value is NULL

Table 3-9 XA_MP3DEC_CONFIG_PARAM_BITRATE subcommand

Subcommand	XA_MP3DEC_CONFIG_PARAM_BITRATE
Description	This command gets the data rate (bit rate) of the input stream in kbps.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_GET_CONFIG_PARAM i_idx XA_MP3DEC_CONFIG_PARAM_BITRATE pv_value &bitrate – Pointer to bit rate variable
Restrictions	None

Example

```
int bitrate;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_MP3DEC_CONFIG_PARAM_BITRATE,
                  (pVOID) &bitrate);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  pv_value is NULL

Table 3-10 XA_MP3DEC_CONFIG_PARAM_PCM_WDSZ subcommand

Subcommand	XA_MP3DEC_CONFIG_PARAM_PCM_WDSZ
Description	This command gets the output PCM word size in bits, which may be different from the one given by the user if the user specified a value other than 16 or 24.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_GET_CONFIG_PARAM i_idx XA_MP3DEC_CONFIG_PARAM_PCM_WDSZ pv_value &pcm_wd_sz – Pointer to PCM word size
Restrictions	None

Example

```
int pcm_wd_sz;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_MP3DEC_CONFIG_PARAM_PCM_WDSZ,
                  (pVOID) &pcm_wd_sz);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  pv_value is NULL

Table 3-11 XA_MP3DEC_CONFIG_PARAM_MCH_STATUS subcommand

Subcommand	XA_MP3DEC_CONFIG_PARAM_MCH_STATUS
Description	This command gets the multi-channel status of the library and the input stream. The status may be one of the following: 0 – multi-channel decoding not supported, 1 – multi-channel decoding disabled, 2 – multi-channel content not available, 3 – multi-channel content decoding in progress. See Section 3.2.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_GET_CONFIG_PARAM i_idx XA_MP3DEC_CONFIG_PARAM_MCH_STATUS pv_value &mch_status – Pointer to the multi-channel status variable
Restrictions	None

Example

```
int mch_status;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_MP3DEC_CONFIG_PARAM_MCH_STATUS,
                  (pVOID) &mch_status);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  pv_value is NULL

Table 3-12 XA_MP3DEC_CONFIG_PARAM_EXTN_PRESENT subcommand

Subcommand	XA_MP3DEC_CONFIG_PARAM_EXTN_PRESENT
Description	This command indicates if an extension substream is present in the input stream. This command is supported only by the MP3 5.1-Channel Decoder.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_GET_CONFIG_PARAM i_idx XA_MP3DEC_CONFIG_PARAM_EXTN_PRESENT pv_value &extn_present – Pointer to the extension substream present flag
Restrictions	None

Example

```
int extn_present;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_MP3DEC_CONFIG_PARAM_EXTN_PRESENT,
                  (pVOID) &extn_present);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  pv_value is NULL

Table 3-13 XA_MP3DEC_CONFIG_PARAM_LFE_PRESENT subcommand

Subcommand	XA_MP3DEC_CONFIG_PARAM_LFE_PRESENT
Description	This command indicates if the LFE channel is present in the input multi-channel stream. This command is supported only by the MP3 5.1-Channel Decoder.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_GET_CONFIG_PARAM i_idx XA_MP3DEC_CONFIG_PARAM_LFE_PRESENT pv_value &lfe_present – Pointer to the LFE present flag
Restrictions	None

Example

```
int lfe_present;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_MP3DEC_CONFIG_PARAM_LFE_PRESENT,
                  (pVOID) &lfe_present);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  pv_value is NULL

Table 3-14 XA_MP3DEC_CONFIG_PARAM_NUM_XCHAN subcommand

Subcommand	XA_MP3DEC_CONFIG_PARAM_NUM_XCHAN
Description	This command gets the number of extra channels in addition to the main audio channels and potentially the LFE channel. This command is supported only by the MP3 5.1-Channel Decoder.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_GET_CONFIG_PARAM i_idx XA_MP3DEC_CONFIG_PARAM_NUM_XCHAN pv_value &num_xchan – Pointer to the number of extra channels variable
Restrictions	None

Example

```
int num_xchan;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_MP3DEC_CONFIG_PARAM_NUM_XCHAN,
                  (pVOID) &num_xchan);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  pv_value is NULL

Table 3-15 XA_MP3DEC_CONFIG_PARAM_CHAN_CONFIG subcommand

Subcommand	XA_MP3DEC_CONFIG_PARAM_CHAN_CONFIG
Description	This command gets the channel configuration of the input stream. The valid channel configuration codes are listed in Section 3.2. This command is supported only by the MP3 5.1-Channel Decoder.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_GET_CONFIG_PARAM i_idx XA_MP3DEC_CONFIG_PARAM_CHAN_CONFIG pv_value &chan_config – Pointer to the channel configuration variable
Restrictions	None

Example

```
int chan_config;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_MP3DEC_CONFIG_PARAM_CHAN_CONFIG,
                  (pVOID) &chan_config);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  pv_value is NULL

Table 3-16 XA_MP3DEC_CONFIG_PARAM_CHAN_MAP subcommand

Subcommand	XA_MP3DEC_CONFIG_PARAM_CHAN_MAP
Description	This command gets the channel map specifying how the channels are arranged in the output buffer. See Section 3.2.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_GET_CONFIG_PARAM i_idx XA_MP3DEC_CONFIG_PARAM_CHAN_MAP pv_value &chan_map – Pointer to the channel map variable
Restrictions	None

Example

```
int chan_map;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_MP3DEC_CONFIG_PARAM_CHAN_MAP,
                  (pVOID) &chan_map);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  pv_value is NULL

Table 3-17 XA_MP3DEC_CONFIG_PARAM_DAB_MP2 subcommand

Subcommand	XA_MP3DEC_CONFIG_PARAM_DAB_MP2
Description	This command gets the dabmp2 flag. See Section 3.2.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_GET_CONFIG_PARAM i_idx XA_MP3DEC_CONFIG_PARAM_DAB_MP2 pv_value &dabmp2 – Pointer to the channel map variable
Restrictions	None

Example

```
int dabmp2;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_MP3DEC_CONFIG_PARAM_DAB_MP2,
                  (pVOID) &dabmp2);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  pv_value is NULL

Table 3-18 XA_MP3DEC_CONFIG_PARAM_ORIGINAL_OR_COPY subcommand

Subcommand	XA_MP3DEC_CONFIG_PARAM_ORIGINAL_OR_COPY
Description	This command gets the original/copy flag encoded in frame header. 1 – The bitstream is original 0 – The bitstream is copy
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_GET_CONFIG_PARAM i_idx XA_MP3DEC_CONFIG_PARAM_ORIGINAL_OR_COPY pv_value &IsOriginal – Pointer to the flag variable
Restrictions	This API returns the value read from the `recently decoded frame header`. Hence this API should be called only after successful decoding of a frame. The return value may not be valid if there is any frame decoding error.

Example

```
int IsOriginal;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  MP3DEC_CONFIG_PARAM_ORIGINAL_OR_COPY,
                  (pVOID) &IsOriginal);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  pv_value is NULL

Table 3-19 XA_MP3DEC_CONFIG_PARAM_COPYRIGHT_FLAG subcommand

Subcommand	`XA_MP3DEC_CONFIG_PARAM_COPYRIGHT_FLAG`
Description	This command gets the copyright flag encoded in the bitstream 1 – The bitstream is copy right protected 0 – There is no copy right in the stream

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_GET_CONFIG_PARAM`
	
	`i_idx`
	`XA_MP3DEC_CONFIG_PARAM_COPYRIGHT_FLAG`
	
	`pv_value`
	`&IsCopyRigthProtected` – Pointer to the flag variable

Restrictions	This API returns the value read from the `recently decoded frame header`. Hence this API should be called only after successful decoding of a frame. The return value may not be valid if there is any frame decoding error.

Example

```
int IsCopyRigthProtected;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_MP3DEC_CONFIG_PARAM_COPYRIGHT_FLAG,
                  (pVOID) &IsCopyRigthProtected);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

Table 3-20 XA_MP3DEC_CONFIG_PARAM_MCH_EXT_HDR_INFO subcommand

Subcommand	XA_MP3DEC_CONFIG_PARAM_MCH_HDR_INFO
Description	This command returns the header info read from the multi-channel extension header. This Info contains downmixing procedures, multi-channel audio playback info, info related to multi lingual audio, etc. Interpretation of this information is given in ISO/IEC 13818 part 3.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_GET_CONFIG_PARAM i_idx XA_MP3DEC_CONFIG_PARAM_MCH_HDR_INFO pv_value &mchExtHdrInfo info – Pointer to the structure of type *xa_mch_ext_hdr_info_t* into which multi-channel extension header info is stored. The definition of the structure is present in xa_mp3_dec_api.h
Restrictions	This API returns the value read from the `recently decoded frame header`. Hence this API should be called only after successful decoding of a frame. The return value may not be valid if there is any frame decoding error. This API is available only in mp3mch_dec.

Example

```
xa_mch_ext_hdr_info_t mchExtHdrInfo;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_MP3DEC_CONFIG_PARAM_MCH_HDR_INFO,
                  (pVOID) &mchExtHdrInfo);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  pv_value is NULL

Table 3-21 XA_MP3DEC_CONFIG_PARAM_MCH_COPYRIGHT_ID_PTR subcommand

Subcommand	`XA_MP3DEC_CONFIG_PARAM_MCH_COPYRIGHT_ID_PTR`
Description	This command returns a pointer to an array of nine unsigned characters (representing 72 bits of copyright identification information received from the bitstream). The first bit is in bit 7 of `p_copyright_info[0]` and the last bit is in bit 0 of `p_copyright_info[8]`.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_GET_CONFIG_PARAM`
	
	`i_idx`
	`XA_MP3DEC_CONFIG_PARAM_MCH_COPYRIGHT_ID_PTR`
	
	`pv_value`
	& `p_copyright_info` – Pointer to copyright_info array

Restrictions	The application should treat the array containing the copyright identification as read-only. As the information is available only after accumulating 72 bits, one from each frame, the application will get a valid pointer only after successful accumulation is done. When there is no copyright identification in the stream this API returns a NULL pointer Also when the accumulation is in progress and no prior valid identifier is available, this API returns a NULL pointer. The return value may not be valid if there is any frame decoding error. This API is available only in `mp3mch_dec`.

Example

```
char * p_copyright_info;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_MP3DEC_CONFIG_PARAM_MCH_COPYRIGHT_ID_PTR,
                  (pVOID) &p_copyright_info);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

4. Introduction to the Example Test Bench

The supplied test bench is made of the following files:

- Test bench source files in `test/src`
  - `xa_mp3_dec_error_handler.c`
  - `xa_mp3_dec_sample_testbench.c`
  - `id3_tag_decode.c`
- `Optional post processing test bench source files`
  - `See readme.txt in test/build`
- Test bench include files in `test/include`
  - `id3_tag_decode.h`
- Makefile to build the executable in `test/build`
  - `makefile_testbench_sample`
- Sample parameter file to use during run in `test/build`
  - `paramfilesimple.txt`

4.1 Making the Executable

To build the application, follow these steps:

1. Go to `test/build`.
2. Type:
   `xt-make –f` makefile_testbench_sample `clean <lib>`

where `<lib>` is `mp2`, `mp3` or `mp3mch`. This will build the example decoder application `xa_<lib>_dec_test`.

Optional post processing can be built from the test bench source files. See readme.txt in test/build

Note: If you have source code distribution, you must build the <lib> library before you can build the test-bench. You can build the library by following these steps:

1. Go to the build directory
2. Type:
   `$xt-make clean <lib> install`

This will build the <lib> library and copy it to the `lib` directory.

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_<lib>_dec_test \
            -ifile:<infile> -ofile:<outfile> \
            [-pcmsz:<pcmbitwidth>] \
            [-crc:<crc_check>]
            [-mch:<mch_enable>]
            [-dabmp2:<dabmp2_enable>]
```

Where

<infile> is the name of the input MPEG audio file.

<outfile> is the name of the output file.

<pcmbitwidth> is the PCM bit-width which can be either 16 or 24 bits.

<crc_check> is 0 to disable or 1 to enable CRC validation.

<mch_enable> is 0 to disable or 1 to enable multi-channel decoding (`mp3mch_dec` only).

<dabmp2_enable> is 0 to disable or 1 to enable DAB-specific Scale factor CRC check.

The following command-line options are specific to error concealment.

```
-pcmsz:<pcmbitwidth>
-numch:<numch>
-numsamps_perch:<numsamps_perch>

-num_mute_frames:<num_mute_frames>
-num_rampdown_samps:<num_rampdown_samps>
-num_rampup_samps:<num_rampup_samps>

-alpha0x:<alpha>
-beta0x:<beta>
-force_mute:<force_mute>
```

Where

<*pcmbitwidth*> is the PCM bit-width which can be either 16 or 24 bits.
<*numch*> can be 1 or 2.
<*numsamps_perch*> can be <= 1152

<*num_mute_frames*> Number of frames to continue mute state during concealment phase
<*num_rampdown_samps*> Number of samples to apply ramp-down while entering concealment phase
<*num_rampup_samps*> Number of samples to apply ramp-up while exiting concealment phase

<*alpha*> Ramp down scale factor in 1.31 format
<*beta*> Ramp up scale factor in 1.31 format
<*force_mute*> 0 to disable, 1 to enable soft muting.
When set to 1, error concealment will forcefully activate and continue till this flag is set to zero

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 for writing into the `paramfilesimple.txt` file is as follows:

```
@Start
@Input_path <path to be prepended to all input filenames>
@Output_path <path to be prepended to all output filenames>
<command line 1>
<command line 2>
....
@Stop
```

The MP3 Decoder 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 to the test bench program.

Note Generally, 16-bit output is used for applications.

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. For 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.

4.3 ID3 Decoding Example

Part of the MP3 decoder example test bench is a decoder for the ID3 container format. This container is used to carry information about the content of the encoded audio, which is added to the bitstream. There are two popular versions: ID3v1 and ID3v2. This section gives an introduction to the formats.

ID3v1

ID3v1 is placed towards the end of the MP3 file. To make it easy to detect, a fixed size of 128 bytes was chosen. The tag has the structure as shown in Table 4-1.

Table 4-1 ID3v1 Structure

Field	Size (in Bytes)
“TAG”	3
Song title	30
Artist	30
Album	30
Year	4
Comment	30
Genre	1

The first three bytes are always "TAG" and is the identification that this is an ID3 tag. The easiest way to decode an ID3v1 tag is to look for the word "TAG" 128 bytes from the end of a file.

ID3v2

ID3v2 is a general tagging format for audio, which makes it possible to store meta-data about the audio inside the audio file itself. The ID3 tag described in this document is mainly targeted at files encoded with MPEG-1/2 layer I, MPEG-1/2 layer II, MPEG-1/2 layer III and MPEG-2.5, but may work with other types of encoded audio or as a stand-alone format for audio meta-data.

ID3v2 is designed to be as flexible and expandable as possible to meet any new metadata information needs that might arise. To achieve this flexibility, ID3v2 is constructed as a container for several information blocks, called frames, whose format need not be known to the software that encounters them. At the start of every frame is a unique and predefined identifier, a size descriptor that allows software to skip unknown frames, and a flags field.

The overall tag structure for ID3v2 is as shown in Table 4-2.

Table 4-2 ID3v2 Structure

Header (10 bytes)
Extended Header
Frames (variable length)
Padding
(variable length, OPTIONAL)
Footer (10 bytes, OPTIONAL)

5. Reference

[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)*

[3] *ETSI EN 300 401 V1.4.1 Radio Broadcasting Systems; Digital Audio Broadcasting (DAB) to mobile, portable and fixed receivers*