



Vorbis Decoder

Programmer's Guide

For HiFi DSPs and Fusion F1 DSP



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

© 2022 Cadence Design Systems, Inc. All rights reserved.
Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Trademarks: Trademarks and service marks of Cadence Design Systems, Inc. (Cadence) contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 1-800-862-4522.

All other trademarks are the property of their respective holders.

Patents: Licensed under U.S. Patent Nos. 7,526,739; 8,032,857; 8,209,649; 8,266,560; 8,650,516

Restricted Print Permission: This publication is protected by copyright and any unauthorized use of this publication may violate copyright, trademark, and other laws. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. This statement grants you permission to print one (1) hard copy of this publication subject to the following conditions:

- The publication may be used solely for personal, informational, and noncommercial purposes;
- The publication may not be modified in any way;
- Any copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement,
- The information contained in this document cannot be used in the development of like products or software, whether for internal or external use, and shall not be used for the benefit of any other party, whether or not for consideration; and
- Cadence reserves the right to revoke this authorization at any time, and any such use shall be discontinued immediately upon written notice from Cadence.

Disclaimer: Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. The information contained herein is the proprietary and confidential information of Cadence or its licensors, and is supplied subject to, and may be used only by Cadence's customer in accordance with, a written agreement between Cadence and its customer. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

Restricted Rights: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

For further assistance, contact Cadence Online Support at <https://support.cadence.com/>.
Copyright © 2022, Cadence Design Systems, Inc. All rights reserved.

Version: 1.8
Last Updated: October 2022

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

Contents

| | | |
|--------|--|----|
| 1. | Introduction to the HiFi Vorbis Decoder..... | 1 |
| 1.1 | Vorbis Description | 1 |
| 1.2 | Document Overview | 1 |
| 1.3 | HiFi Vorbis Decoder Specifications | 2 |
| 1.4 | HiFi Vorbis Decoder Performance | 2 |
| 1.4.1 | Memory | 2 |
| 1.4.2 | Timings..... | 3 |
| 2. | Generic HiFi Audio Codec API | 4 |
| 2.1 | Memory Management | 4 |
| 2.2 | C Language API | 5 |
| 2.3 | Generic API Errors | 7 |
| 2.4 | Commands | 8 |
| 2.4.1 | Start-up API Stage | 9 |
| 2.4.2 | Set Codec-Specific Parameters Stage | 10 |
| 2.4.3 | Memory Allocation Stage | 11 |
| 2.4.4 | Initialize Codec Stage | 12 |
| 2.4.5 | Get Codec-Specific Parameters Stage | 13 |
| 2.4.6 | Execute Codec Stage | 14 |
| 2.5 | Files Describing the API | 14 |
| 2.6 | HiFi API Command Reference | 15 |
| 2.6.1 | Common API Errors | 16 |
| 2.6.2 | XA_API_CMD_GET_LIB_ID_STRINGS | 17 |
| 2.6.3 | XA_API_CMD_GET_API_SIZE | 20 |
| 2.6.4 | XA_API_CMD_INIT | 21 |
| 2.6.5 | XA_API_CMD_GET_MEMTABS_SIZE | 25 |
| 2.6.6 | XA_API_CMD_SET_MEMTABS_PTR | 26 |
| 2.6.7 | XA_API_CMD_GET_N_MEMTABS..... | 27 |
| 2.6.8 | XA_API_CMD_GET_MEM_INFO_SIZE | 28 |
| 2.6.9 | XA_API_CMD_GET_MEM_INFO_ALIGNMENT | 29 |
| 2.6.10 | XA_API_CMD_GET_MEM_INFO_TYPE | 30 |
| 2.6.11 | XA_API_CMD_GET_MEM_INFO_PRIORITY | 32 |
| 2.6.12 | XA_API_CMD_SET_MEM_PTR | 34 |
| 2.6.13 | XA_API_CMD_INPUT_OVER | 35 |
| 2.6.14 | XA_API_CMD_SET_INPUT_BYTES..... | 36 |
| 2.6.15 | XA_API_CMD_GET_CURIDX_INPUT_BUF | 37 |
| 2.6.16 | XA_API_CMD_EXECUTE | 38 |

| | | |
|--------|--|----|
| 2.6.17 | XA_API_CMD_GET_CONFIG_PARAM | 42 |
| 2.6.18 | XA_API_CMD_SET_CONFIG_PARAM | 45 |
| 3. | HiFi DSP Vorbis Decoder | 47 |
| 3.1 | Files Specific to the Vorbis Decoder | 48 |
| 3.2 | Configuration Parameters | 48 |
| 3.3 | Comment Buffer Usage..... | 49 |
| 3.4 | API Usage Notes..... | 50 |
| 3.5 | Vorbis Decoder-Specific Commands | 51 |
| 3.5.1 | Configuration and Execution Errors | 51 |
| 3.5.2 | XA_API_CMD_SET_CONFIG_PARAM | 54 |
| 3.5.3 | XA_API_CMD_GET_CONFIG_PARAM | 60 |
| 4. | Introduction to Example Test Bench..... | 64 |
| 4.1 | Building the Executable..... | 64 |
| 4.2 | Usage | 65 |
| 5. | References | 66 |

Figures

| | |
|--|----|
| Figure 1 HiFi Audio Codec Interfaces | 4 |
| Figure 2 API Command Sequence Overview | 8 |
| Figure 3 Flow Chart for Vorbis Decoder Integration | 47 |

Tables

| | |
|--|----|
| Table 2-1 Vorbis Decoder API | 6 |
| Table 2-2 Commands for Initialization | 9 |
| Table 2-3 Commands for Setting Parameters | 10 |
| Table 2-4 Commands for Initial Table Allocation | 11 |
| Table 2-5 Commands for Memory Allocation..... | 11 |
| Table 2-6 Commands for Initialization | 12 |
| Table 2-7 Commands for Getting Parameters | 13 |
| Table 2-8 Commands for Codec Execution | 14 |
| Table 2-9 XA_CMD_TYPE_LIB_NAME subcommand..... | 17 |
| Table 2-10 XA_CMD_TYPE_LIB_VERSION subcommand..... | 18 |
| Table 2-11 XA_CMD_TYPE_API_VERSION subcommand..... | 19 |
| Table 2-12 XA_API_CMD_GET_API_SIZE command..... | 20 |
| Table 2-13 XA_CMD_TYPE_INIT_API_PRE_CONFIG_PARAMS subcommand | 21 |
| Table 2-14 XA_CMD_TYPE_INIT_API_POST_CONFIG_PARAMS subcommand..... | 22 |
| Table 2-15 XA_CMD_TYPE_INIT_PROCESS subcommand | 23 |
| Table 2-16 XA_CMD_TYPE_INIT_DONE_QUERY subcommand..... | 24 |
| Table 2-17 XA_API_CMD_GET_MEMTABS_SIZE command..... | 25 |
| Table 2-18 XA_API_CMD_SET_MEMTABS_PTR command..... | 26 |
| Table 2-19 XA_API_CMD_GET_N_MEMTABS command | 27 |
| Table 2-20 XA_API_CMD_GET_MEM_INFO_SIZE command..... | 28 |
| Table 2-21 XA_API_CMD_GET_MEM_INFO_ALIGNMENT command | 29 |
| Table 2-22 XA_API_CMD_GET_MEM_INFO_TYPE command | 30 |
| Table 2-23 Memory Type Indices | 30 |
| Table 2-24 XA_API_CMD_GET_MEM_INFO_PRIORITY command | 32 |
| Table 2-25 Memory Priorities | 32 |
| Table 2-26 XA_API_CMD_SET_MEM_PTR command | 34 |

| | | |
|------------|---|----|
| Table 2-27 | XA_API_CMD_INPUT_OVER command | 35 |
| Table 2-28 | XA_API_CMD_SET_INPUT_BYTES command | 36 |
| Table 2-29 | XA_API_CMD_GET_CURIDX_INPUT_BUF command | 37 |
| Table 2-30 | XA_CMD_TYPE_DO_EXECUTE subcommand | 38 |
| Table 2-31 | XA_CMD_TYPE_DONE_QUERY subcommand | 39 |
| Table 2-32 | XA_CMD_TYPE_DO_RUNTIME_INIT subcommand | 40 |
| Table 2-33 | XA_API_CMD_GET_OUTPUT_BYTES command | 41 |
| Table 2-34 | XA_CONFIG_PARAM_CUR_INPUT_STREAM_POS subcommand | 42 |
| Table 2-35 | XA_CONFIG_PARAM_GEN_INPUT_STREAM_POS subcommand | 43 |
| Table 2-36 | XA_CONFIG_PARAM_GEN_INPUT_STREAM_POS subcommand | 44 |
| Table 2-37 | XA_CONFIG_PARAM_CUR_INPUT_STREAM_POS subcommand | 45 |
| Table 2-38 | XA_CONFIG_PARAM_CUR_INPUT_STREAM_POS subcommand | 46 |
| Table 3-1 | XA_VORBISDEC_CONFIG_PARAM_COMMENT_MEM_PTR subcommand | 54 |
| Table 3-2 | XA_VORBISDEC_CONFIG_PARAM_COMMENT_MEM_SIZE subcommand | 55 |
| Table 3-3 | XA_VORBISDEC_CONFIG_PARAM_RAW_VORBIS_FILE_MODE subcommand | 56 |
| Table 3-4 | XA_VORBISDEC_CONFIG_PARAM_RAW_VORBIS_LAST_PKT_GRANULE_POS subcommand | 57 |
| Table 3-5 | XA_VORBISDEC_CONFIG_PARAM_OGG_MAX_PAGE_SIZE subcommand | 58 |
| Table 3-6 | XA_VORBISDEC_CONFIG_PARAM_RUNTIME_MEM subcommand | 59 |
| Table 3-7 | XA_VORBISDEC_CONFIG_PARAM_SAMP_FREQ subcommand | 60 |
| Table 3-8 | XA_VORBISDEC_CONFIG_PARAM_NUM_CHANNELS subcommand | 61 |
| Table 3-9 | XA_VORBISDEC_CONFIG_PARAM_PCM_WDSZ subcommand | 62 |
| Table 3-10 | XA_VORBISDEC_CONFIG_PARAM_GET_CUR_BITRATE subcommand | 63 |

Document Change History

| Version | Changes |
|---------|--|
| 1.3 | <ul style="list-style-type: none"> ■ Added 16-bit PCM data output feature to Section 1.3 ■ Updated Text Library Kbytes in Section 1.4.1 ■ Updated Section 1.4.2 ■ Added restriction information to XA_API_CMD_SET_CONFIG_PARAM in Section 2.6.19 ■ Added information for granule_pos and file_type in Section 3.2. ■ Added new bullet items in Section 3.4. ■ Added XA_VORBISDEC_CONFIG_NONFATAL_GROUPED_STREAM, XA_VORBISDEC_CONFIG_FATAL_INVALID_PARAM, and XA_VORBISDEC_EXECUTE_FATAL_CORRUPT_STREAM in Section 3.5.1. |
| 1.4 | <ul style="list-style-type: none"> ■ Changed generic references of HiFi 2 to HiFi ■ Deleted references to Diamond 330HiFi. ■ Added memory and timings data for HiFi Mini and HiFi 3. |
| 1.5 | <ul style="list-style-type: none"> ■ Added Performance data for HiFi 4 in Section 1.4. |
| 1.6 | <ul style="list-style-type: none"> ■ Added Performance data for HiFi 3z in Section 1.4. ■ Added new API for managing runtime memory use in Section 3. |
| 1.7 | <ul style="list-style-type: none"> ■ Added Performance data for Fusion F1 in Section 1.4. |
| 1.8 | <ul style="list-style-type: none"> ■ Optimization done for HiFi 1 |

1. Introduction to the HiFi Vorbis Decoder

The HiFi Vorbis Decoder implements the Vorbis I audio codec standard specified by the Xiph.Org foundation. It is capable of decoding audio streams encoded by the reference implementation of the codec (*libvorbis* versions 1.0 and later).^{[1] [2]}

The vendor source code version is Xiph.org Vorbis 1.0.2 with some modifications (including raw Vorbis decoding support).

For this document, HiFi DSPs include Fusion F1 DSP.

1.1 Vorbis Description

A quote from the Xiph.Org foundation:

“Ogg Vorbis is a fully open, non-proprietary, patent-and-royalty-free, general-purpose compressed audio format for mid to high quality (8kHz-48.0kHz, 16+ bit, polyphonic) audio and music at fixed and variable bitrates from 16 to 128 kbps/channel. This places Vorbis in the same competitive class as audio representations such as MPEG-4 (AAC), and similar to, but higher performance than MPEG-1/2 audio layer 3, MPEG-4 audio (TwinVQ), WMA and PAC.”

Ogg Vorbis files and streams contain audio payloads encoded using the Vorbis audio codec standard and carried in Ogg – Xiph.Org’s container format for audio, video, and metadata. However, the Vorbis streams are not required to be carried inside an Ogg container; they can also be decoded in the raw format. The rest of this document refers to the HiFi Vorbis decoder as the *HiFi Vorbis decoder* or simply as the *Vorbis decoder*.

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 and these are described after the introduction. Section 3 covers all the features and information particular to the HiFi Vorbis Decoder. Section 4 describes the example test bench. Section 5 provides references.

1.3 HiFi Vorbis Decoder Specifications

The HiFi DSP Vorbis Decoder from Cadence Tensilica implements the following features in conformance to the Ogg Vorbis specifications:

- Cadence Audio Codec API is used
- Sampling rates: 8, 11.025, 16, 22.05, 32, 44.1, and 48 kHz
- Mono and stereo channels
- Quality levels: 0 through 10. Vorbis is variable bit rate by design and quality levels are proportional to bit rates
- Support for Floor 1 encoded streams
- Support for bit-rate peeled streams
- Support for decoding of Ogg Vorbis streams
- Support for decoding of raw Vorbis streams (Vorbis streams without an Ogg container)
- Output: 16 bit PCM data, interleaved

1.4 HiFi Vorbis Decoder Performance

The HiFi DSP Vorbis 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 Memory

| Text (Kbytes) | | | | | | Data |
|---------------|--------|--------|---------|--------|--------|--------|
| Fusion F1 | HiFi 1 | HiFi 3 | HiFi 3z | HiFi 4 | HiFi 5 | Kbytes |
| 33.9 | 34.1 | 33.0 | 34.7 | 36.5 | 45.1 | 18.9 |

| Stream Type | Runtime Memory | Runtime Memory (Kbytes) | | | | |
|-------------|----------------|-------------------------|---------|-------|---------------|--------|
| | | Persistent | Scratch | Stack | Input | Output |
| Ogg Stream | 0 | 84.0 | 33.1 | 0.7 | <ogg_maxpage> | 4.0 |
| | 1 | 116.0 | 65.1 | 0.7 | <ogg_maxpage> | 4.0 |
| Raw Stream | 0 | 86.0 | 33.1 | 0.7 | 12.0 | 4.0 |
| | 1 | 118.0 | 65.1 | 0.7 | 12.0 | 4.0 |

Note Default value of <ogg_maxpage> is 12, minimum is 12, and maximum is 128. Pre config parameter XA_VORBISDEC_CONFIG_PARAM_OGG_MAX_PAGE_SIZE is provided to modify this, for more details refer to Table 3-5. Pre-config parameter XA_VORBISDEC_CONFIG_PARAM_RUNTIME_MEM is provided to choose between Runtime Memory 0 or 1 configuration (default is 0), for more details refer to Table 3-6.

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 | 10.3 | 9.8 | 10.9 | 8.4 | 8.9 | 8.4 |
| 44.1 | 2 | 320 | 13.4 | 13.0 | 14.7 | 11.3 | 11.8 | 11.2 |
| 48 | 2 | 500 | 15.1 | 14.7 | 16.7 | 12.8 | 13.3 | 12.7 |

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. The MCPS numbers for HiFi 3/HiFi 3z/HiFi 4/HiFi 5/Fusion F1 are obtained by running the test with binaries that are based on 24-bit optimized source code. Optimization is done for HiFi 1. No specific optimization is done for HiFi 3/HiFi 3 z /HiFi 4/HiFi 5/Fusion F1.

Note In the Ogg Vorbis format, codebooks are encoded in the Ogg Vorbis stream. Due to codebook processing, decoder initialization may take up to 16 million cycles per audio stream.

Note All the memory and MCPS numbers are captured with RI-2022.9 tools and XT-CLANG compiler.

2. Generic HiFi Audio Codec API

This section 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.

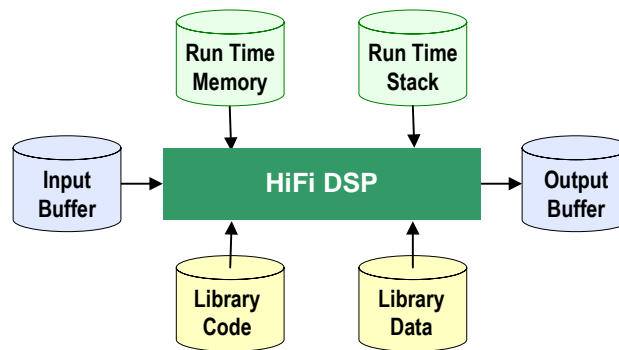


Figure 1 HiFi Audio Codec Interfaces

Section 2.1 discusses all the types of runtime 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 runtime.

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.

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 must 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 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 the same and are specified in Table 2-1.

Table 2-1 Vorbis Decoder API

| xa_<codec>_dec | |
|-----------------------------|---|
| Description | This C API is the only access function to the audio codec. |
| Syntax | <pre>XA_ERRORCODE xa_<codec>(xa_codec_handle_t p_xa_module_obj, WORD32 i_cmd, WORD32 i_idx, pVOID pv_value);</pre> |
| Parameters | <p>p_xa_module_obj Pointer to opaque API structure.</p> <p>i_cmd Command.</p> <p>i_idx Command subtype or index.</p> <p>pv_value Pointer to the variable used to pass in, or get out properties, from state structure</p> |
| 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 is 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 subdivided into those that are fatal, which require the restarting of the entire codec, and those that are non-fatal 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 shown in Figure 2. For each stage of the flow chart, there is a section that lists the required commands in the order they should occur. For individual command 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.

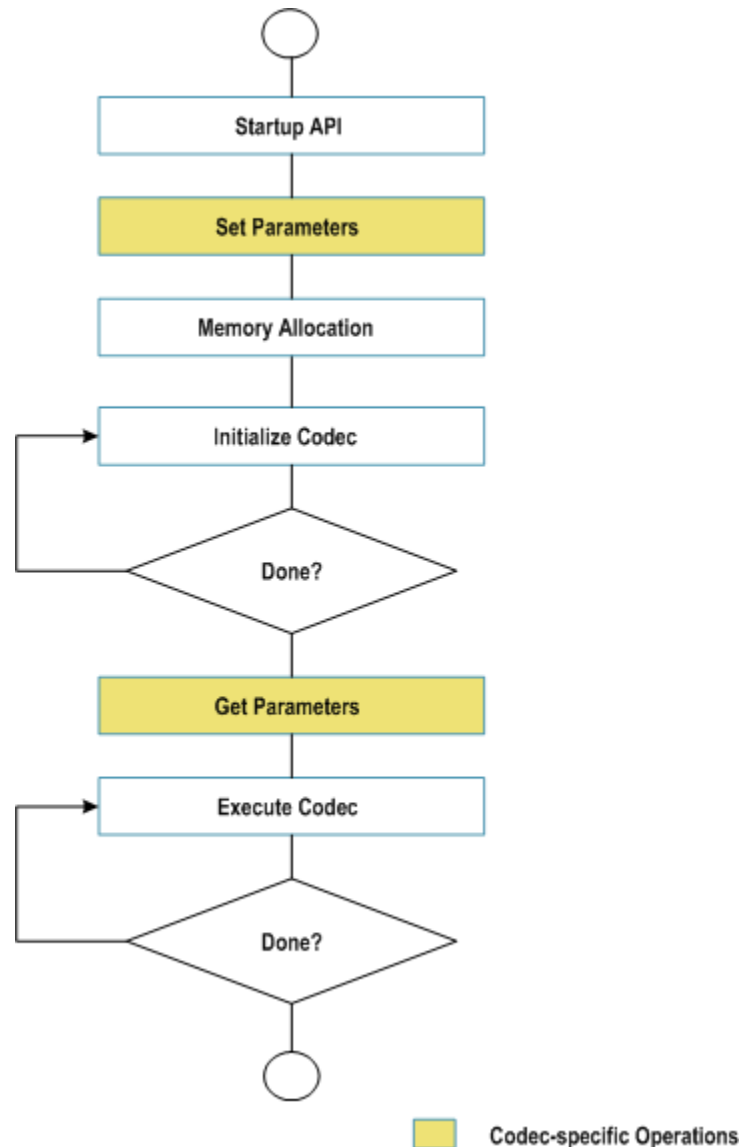


Figure 2 API Command Sequence Overview

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 codec specific 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 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. 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 |
|--|---|
| <code>XA_API_CMD_SET_INPUT_BYTES</code> | Set the number of bytes available in the input buffer for initialization. |
| <code>XA_API_CMD_INPUT_OVER</code> | Signals to the codec the end of the bitstream |
| <code>XA_API_CMD_INIT</code> <code>XA_CMD_TYPE_INIT_PROCESS</code> | Search for the valid header, does header decoding to get the parameters and initializes state and configuration structures. |
| <code>XA_API_CMD_INIT</code> <code>XA_CMD_TYPE_INIT_DONE_QUERY</code> | Check if the initialization process has completed. |
| <code>XA_API_CMD_GET_CURIDX_INPUT_BUF</code> | 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 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 |
|--|--|
| <code>XA_API_CMD_INPUT_OVER</code> | Signal the end of bitstream to the library. |
| <code>XA_API_CMD_SET_INPUT_BYTES</code> | Set the number of bytes available in the input buffer for the execution. |
| <code>XA_API_CMD_EXECUTE</code> <code>XA_CMD_TYPE_DO_EXECUTE</code> | Execute the codec thread. |
| <code>XA_API_CMD_EXECUTE</code> <code>XA_CMD_TYPE_DONE_QUERY</code> | Check if the end of stream has been reached. |
| <code>XA_API_CMD_GET_OUTPUT_BYTES</code> | Get the number of bytes output by the codec in the last frame. |
| <code>XA_API_CMD_GET_CURIDX_INPUT_BUF</code> | Get the number of input buffer bytes consumed by the last call to the codec. |

2.5 Files Describing the API

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
- `xa_error_handler.h`
Error related function prototypes and definitions

2.6 HiFi API Command Reference

In this section, the different commands are described along with their associated subcommands. The only missing commands 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 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>p_xa_module_obj NULL</p> <p>i_cmd XA_API_CMD_GET_LIB_ID_STRINGS</p> <p>i_idx XA_CMD_TYPE_LIB_NAME</p> <p>pv_value process_name - Pointer to a character buffer in which the name of the library is returned.</p> |
| 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>p_xa_module_obj NULL</p> <p>i_cmd XA_API_CMD_GET_LIB_ID_STRINGS</p> <p>i_idx XA_CMD_TYPE_LIB_VERSION</p> <p>pv_value lib_version – Pointer to a character buffer in which the version of the library is returned</p> |
| 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>p_xa_module_obj NULL</p> <p>i_cmd XA_API_CMD_GET_LIB_ID_STRINGS</p> <p>i_idx XA_CMD_TYPE_API_VERSION</p> <p>pv_value api_version - Pointer to a character buffer in which the version of the API is returned.</p> |
| 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>p_xa_module_obj NULL</p> <p>i_cmd XA_API_CMD_GET_API_SIZE</p> <p>i_idx NULL</p> <p>pv_value &api_size – Pointer to API size variable</p> |
| 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>p_xa_module_obj api_obj - Pointer to API structure</p> <p>i_cmd XA_API_CMD_INIT</p> <p>i_idx XA_CMD_TYPE_INIT_API_PRE_CONFIG_PARAMS</p> <p>pv_value NULL</p> |
| 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>p_xa_module_obj api_obj – Pointer to API structure</p> <p>i_cmd XA_API_CMD_INIT</p> <p>i_idx XA_CMD_TYPE_INIT_API_POST_CONFIG_PARAMS</p> <p>pv_value NULL</p> |
| 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>p_xa_module_obj api_obj - Pointer to API structure</p> <p>i_cmd XA_API_CMD_INIT</p> <p>i_idx XA_CMD_TYPE_INIT_PROCESS</p> <p>pv_value NULL</p> |
| 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>p_xa_module_obj api_obj - Pointer to API structure</p> <p>i_cmd XA_API_CMD_INIT</p> <p>i_idx XA_CMD_TYPE_INIT_DONE_QUERY</p> <p>pv_value &init_done - Pointer to flag that indicates the completion of initialization process.</p> |
| 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>p_xa_module_obj api_obj – Pointer to API structure</p> <p>i_cmd XA_API_CMD_GET_MEMTABS_SIZE</p> <p>i_idx NULL</p> <p>pv_value &proc_mem_tabs_size – Pointer to memory size variable</p> |
| 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>p_xa_module_obj api_obj – Pointer to API structure</p> <p>i_cmd XA_API_CMD_SET_MEMTABS_PTR</p> <p>i_idx NULL</p> <p>pv_value alloc – Allocated pointer</p> |
| 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>p_xa_module_obj api_obj – Pointer to API structure</p> <p>i_cmd XA_API_CMD_GET_N_MEMTABS</p> <p>i_idx NULL</p> <p>pv_value &n_mems – Number of memory blocks required to be allocated</p> |
| 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>p_xa_module_obj api_obj – Pointer to API structure</p> <p>i_cmd XA_API_CMD_GET_MEM_INFO_SIZE</p> <p>i_idx Index of the memory</p> <p>pv_value &size – Pointer to memory size</p> |
| 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 $0 \leq i_idx < n_mems$ (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>p_xa_module_obj api_obj – Pointer to API structure</p> <p>i_cmd XA_API_CMD_GET_MEM_INFO_ALIGNMENT</p> <p>i_idx Index of the memory</p> <p>pv_value &alignment – Pointer to the alignment info variable</p> |
| 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 $0 \leq i_idx < n_mems$ (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>p_xa_module_obj api_obj – Pointer to API structure</p> <p>i_cmd XA_API_CMD_GET_MEM_INFO_TYPE</p> <p>i_idx Index of the memory</p> <p>pv_value &type – Pointer to the memory type variable</p> |
| 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 $0 \leq i_idx < n_mems$ (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>p_xa_module_obj api_obj – Pointer to API structure</p> <p>i_cmd XA_API_CMD_GET_MEM_INFO_PRIORITY</p> <p>i_idx Index of the memory</p> <p>pv_value &priority – Pointer to the memory priority variable</p> |
| 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 $0 \leq i_idx < n_mems$ (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>p_xa_module_obj api_obj – Pointer to API structure</p> <p>i_cmd XA_API_CMD_SET_MEM_PTR</p> <p>i_idx Index of the memory</p> <p>pv_value alloc – Pointer to memory buffer allocated</p> |
| 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
pc_valve is null
- XA_API_FATAL_INVALID_CMD_TYPE
i_idx is an invalid memory block number; valid block numbers obey the relation $0 \leq i_idx < n_mems$ (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>p_xa_module_obj api_obj – Pointer to API structure</p> <p>i_cmd XA_API_CMD_INPUT_OVER</p> <p>i_idx NULL</p> <p>pv_value NULL</p> |
| 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>p_xa_module_obj api_obj – Pointer to API structure</p> <p>i_cmd XA_API_CMD_SET_INPUT_BYTES</p> <p>i_idx NULL</p> <p>pv_value &buff_size – Pointer to the input byte variable</p> |
| 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>p_xa_module_obj api_obj – Pointer to API Structure</p> <p>i_cmd XA_API_CMD_GET_CURIDX_INPUT_BUF</p> <p>i_idx NULL</p> <p>pv_value &bytes_consumed – Pointer to the bytes consumed variable</p> |
| 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>p_xa_module_obj api_obj – Pointer to API structure</p> <p>i_cmd XA_API_CMD_EXECUTE</p> <p>i_idx XA_CMD_TYPE_DO_EXECUTE</p> <p>pv_value NULL</p> |
| 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; 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>p_xa_module_obj api_obj – Pointer to API Structure</p> <p>i_cmd XA_API_CMD_EXECUTE</p> <p>i_idx XA_CMD_TYPE_DONE_QUERY</p> <p>pv_value &flag – Pointer to the flag variable</p> |
| 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 | <p>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. The command should be issued before the application starts feeding the decoder with new data from a random place in the input stream.</p> <p>Note: This command is available in API version 1.14 or later.</p> |
| Actual Parameters | <p>p_xa_module_obj api_obj – Pointer to API structure</p> <p>i_cmd XA_API_CMD_EXECUTE</p> <p>i_idx XA_CMD_TYPE_DO_RUNTIME_INIT</p> <p>pv_value NULL</p> |
| Restrictions | None |

Example

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

Errors

- Common API Errors

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>p_xa_module_obj api_obj – Pointer to API structure</p> <p>i_cmd XA_API_CMD_GET_OUTPUT_BYTES</p> <p>i_idx NULL</p> <p>pv_value &out_bytes – Pointer to the output bytes variable</p> |
| 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

2.6.17 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 | <p>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 <code>size</code> at byte offset <code>offset</code> in the input buffer, then the input stream position range for this unit may be calculated as follows:</p> <pre>start_pos = CUR_INPUT_STREAM_POS + offset end_pos = CUR_INPUT_STREAM_POS + offset + size</pre> |
| Actual Parameters | <p><code>p_xa_module_obj</code> <code>api_obj</code> – Pointer to API structure</p> <p><code>i_cmd</code> XA_API_CMD_GET_CONFIG_PARAM</p> <p><code>i_idx</code> XA_CONFIG_PARAM_CUR_INPUT_STREAM_POS</p> <p><code>pv_value</code> <code>&ui_cur_input_stream_pos</code> – Pointer to the current input stream position variable</p> |
| Restrictions | <p>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.</p> <p>This command is available in API version 1.15 or later.</p> |

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 <code>start_pos</code> (see Error! Reference source not found.), when the library generates the decoded or encoded data corresponding to this unit, it sets <code>GEN_INPUT_STREAM_POS</code> to <code>start_pos</code> . |
| Actual Parameters | <p><code>p_xa_module_obj</code> <code>api_obj</code> – Pointer to API structure</p> <p><code>i_cmd</code> <code>XA_API_CMD_GET_CONFIG_PARAM</code></p> <p><code>i_idx</code> <code>XA_CONFIG_PARAM_GEN_INPUT_STREAM_POS</code></p> <p><code>pv_value</code> <code>&ui_gen_input_stream_pos</code> – Pointer to the input stream position of the generated data variable</p> |
| Restrictions | <p>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.</p> <p>This command is available in API version 1.15 or later.</p> |

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 Error

Table 2-36 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 <code>start_pos</code> (see Error! Reference source not found.), when the library generates the decoded or encoded data corresponding to this unit, it sets <code>GEN_INPUT_STREAM_POS</code> to <code>start_pos</code> . |
| Actual Parameters | <p><code>p_xa_module_obj</code> <code>api_obj</code> – Pointer to API structure</p> <p><code>i_cmd</code> <code>XA_API_CMD_GET_CONFIG_PARAM</code></p> <p><code>i_idx</code> <code>XA_CONFIG_PARAM_GEN_INPUT_STREAM_POS</code></p> <p><code>pv_value</code> <code>&ui_gen_input_stream_pos</code> – Pointer to the input stream position of the generated data variable</p> |
| Restrictions | <p>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.</p> <p>This command is available in API version 1.15 or later.</p> |

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

2.6.18 XA_API_CMD_SET_CONFIG_PARAM

Table 2-37 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 Error! Reference source not found. for details. |
| Actual Parameters | <p>p_xa_module_obj api_obj – Pointer to API structure</p> <p>i_cmd XA_API_CMD_SET_CONFIG_PARAM</p> <p>i_idx XA_CONFIG_PARAM_CUR_INPUT_STREAM_POS</p> <p>pv_value &ui_cur_input_stream_pos – Pointer to the current input stream position variable</p> |
| 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

Table 2-38 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>p_xa_module_obj api_obj – Pointer to API Structure</p> <p>i_cmd XA_API_CMD_SET_CONFIG_PARAM</p> <p>i_idx XA_CONFIG_PARAM_CUR_INPUT_STREAM_POS</p> <p>pv_value &ui_cur_input_stream_pos – Pointer to the current input stream position variable</p> |
| Restrictions | This command is available in API version 1.15 or later. This command is supported for only Ogg Vorbis streams. This command is not supported for raw Vorbis streams as the sync words /packet boundaries are not available. |

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 Vorbis Decoder

The HiFi DSP Vorbis Decoder conforms to the generic codec API. Figure 3 shows the flow chart of the command sequence used in the example test bench.

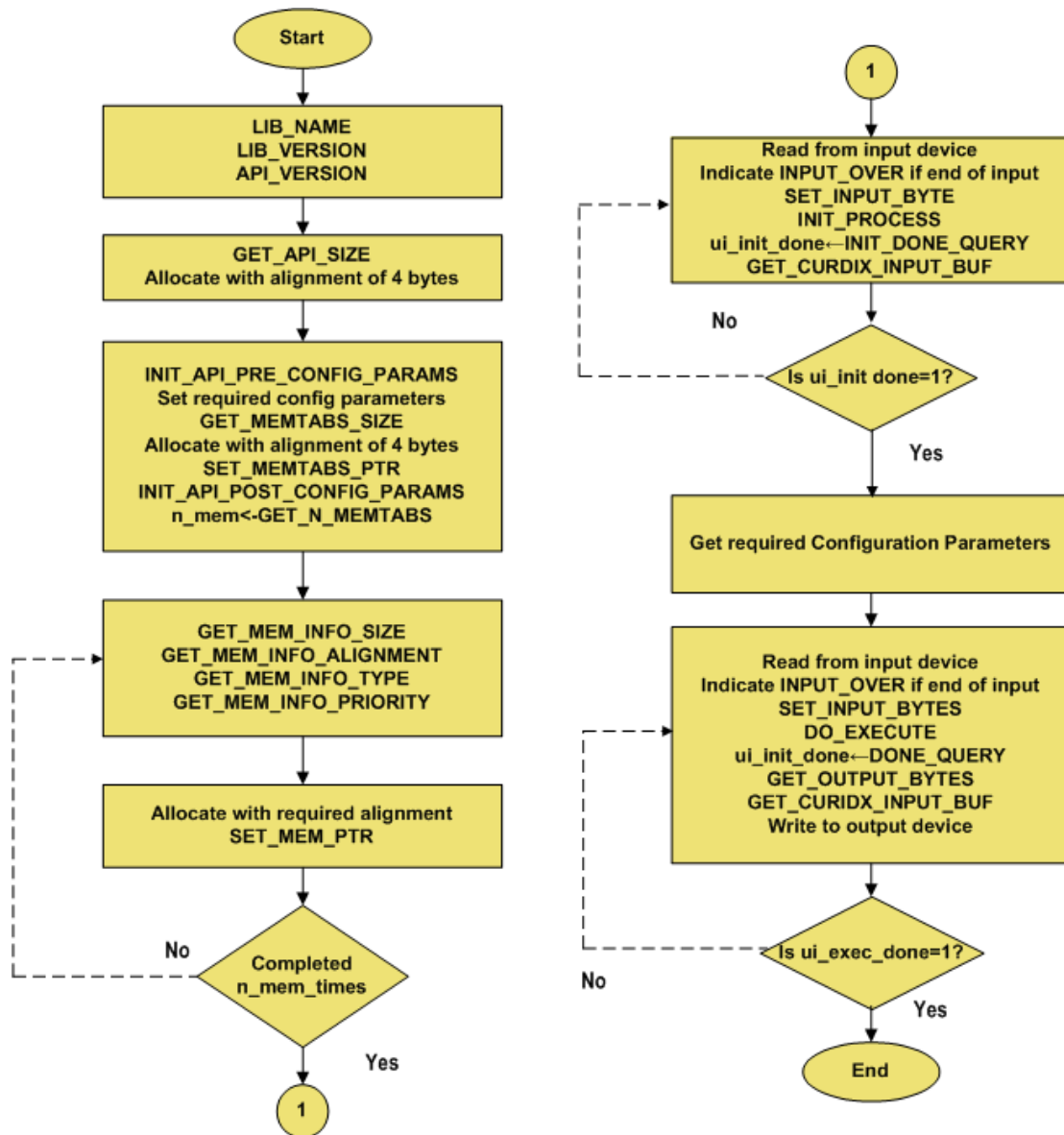


Figure 3 Flow Chart for Vorbis Decoder Integration

3.1 Files Specific to the Vorbis Decoder

- Vorbis decoder parameter header file (include/vorbis_dec): `xa_vorbis_dec_api.h`
- Vorbis decoder library (lib): `xa_vorbis_dec.a`
- The Vorbis decoder API call is defined as:

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

3.2 Configuration Parameters

The HiFi Vorbis decoder library accepts the following parameters from the user. Details of the Set config API commands for these parameters are described in Section 3.5.2.

- **comment_mem_ptr** – A pointer to an optional memory block to hold the vendor string and the user comments extracted from the Ogg Vorbis stream. The format of the comment block is described in Section 3.3. The size of the memory block may be arbitrary and is provided by the application through the `comment_mem_size` parameter. The default value of `comment_mem_ptr` is NULL.
- **comment_mem_size** – The size of the optional comment memory block pointed by the `comment_mem_ptr` parameter. The default value of `comment_mem_size` is 0.
- **granule_pos** – The granule position for the last packet in the stream. This parameter is optional when decoding raw Vorbis streams. If the raw Vorbis stream is extracted from an Ogg Vorbis file, this value can be extracted from the same file. When not available, the value of this parameter is set to -1 by default. The presence of this parameter only affects the decoding of the last packet; the decoder returns some extra PCM output samples when this parameter is not present. In most audio applications, the effects of this behavior is negligible.
- **file_type** – `file_type` specifies whether the input file is a raw Vorbis file or Ogg Vorbis file. The default value is 0, which means the input file is an Ogg Vorbis file. For Ogg Vorbis files, the Vorbis decoder parses and decodes the Ogg container inside the library. The valid Ogg page is first captured and the Vorbis packets are extracted using information provided in the Ogg container. These extracted Vorbis packets are further processed by the core decoder. For raw Vorbis files, the library expects the complete Vorbis packet to be passed in the input buffer to the decoder library.
- **ogg_max_page_size** – `ogg_max_page_size` specifies the maximum ogg page size for ogg vorbis input stream. This parameter decides the input buffer size required to successfully decode the ogg vorbis input stream and has no effect if file mode is raw.
- **runtime_mem** – `runtime_mem` provides option to increase runtime memories persistent and scratch. If the decoder fails to decode a stream with the default persistent and scratch memory, this parameter can be used to increase them.

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. Details of the Get config API commands for these parameters are described in Section 3.5.3.

- **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. There are only two valid values: 1 (mono) and 2 (stereo). In the case of stereo, the output data is 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 in the output buffer. The only valid value is 16.

3.3 Comment Buffer Usage

The format of the comment buffer specified through the `comment_mem_ptr` and the `comment_mem_size` configuration parameters is equivalent to the Ogg Vorbis comment header format ^[3]. It is decoded as follows:

- `[vendor_length]` = unsigned 32-bit integer (little-endian format)
- `[vendor_string]` = UTF-8 vector of `[vendor_length]` octets
- `[user_comment_list_length]` = unsigned 32-bit integer (little-endian format)
- **iterate** `[user_comment_list_length]` times:
 - `[length]` = unsigned 32-bit integer (little-endian format)
 - `[user_comment]` = UTF-8 vector of `[length]` octets

The comment block contents become available after the decoder has been initialized. If the comment header is corrupted or the comment memory buffer provided by the application is not big enough to hold all comment fields, the `[vendor_length]` field is set to `0xFFFFFFFF`. If the application does not point the decoder to a comment memory block, the decoder skips over the comment header. As the decoder does not write to the output buffer until the Execute phase, the application may use the output buffer to extract the vendor string and the comment fields from the stream.

3.4 API Usage Notes

Although the HiFi Vorbis Decoder conforms to the generic codec API described in Section 2, to achieve optimal performance during the Execution phase, consider the following:

- The decoder does not consume bytes from the input buffer every `DO_EXECUTE` call and, therefore, the `GET_CURIDX_INPUT_BUF` command may sometimes return 0. The application may use this information to avoid unnecessary shifting of the input buffer data.
- Because of the varying Vorbis frame sizes, the decoder does not always fill the output buffer completely. Also, some `DO_EXECUTE` calls perform CRC checks for the Ogg page that will increase the peak CPU usage. To improve performance and smooth the CPU load, the application may need to buffer up output samples.
- Support for comment headerless streams: As per Vorbis specifications, the Vorbis stream starts with three header packets (Info, Comment, Setup) in that order. The Vorbis library now supports decoding of the streams without the comment header packet because (1) the comment header packet is not useful for initialization of the decoder, (2) proper decoding of the Vorbis stream can be done without it, and (3) there are known use cases where raw Vorbis streams do not contain the comment header. If the stream has valid info and setup header packets in the above mentioned order, the library can decode it.
- Maximum page/package size limitations: The Vorbis decoder library assumes that the maximum Ogg page or maximum Vorbis packet size for any Ogg Vorbis stream is no greater than 12k bytes. If the packets/page size is beyond this, the decoder will not be able to process such a stream. Packet processing for raw streams: As the packet sizes for raw Vorbis streams are not available/known in advance, the Vorbis decoder initialization and execution for raw streams is designed in such a way that once the library detects /decodes a valid packet, the library will consume a single packet at a time, and then pass the control to the test application. The test application then needs to discard the consumed bytes of the processed packet, load the input buffer with new data, and then pass the new data to the library.
- For raw stream decoding, the application must ensure that header packets are correct. For example, if the bitstream is in an Ogg container, the application must validate the checksum, if the bitstream is an RTP payload, the application must follow the RFC5215. If the wrong header packet is passed to the decoder, the decoder behavior is unknown, and may cause exceptions.
- Use stream position APIs after initialization is complete and at least one packet is successfully decoded. For stream-repositioning, do not set a new stream position before the start of the first packet; as the packet start for a raw stream is not known, stream position APIs will not work well for raw streams.

3.5 Vorbis Decoder-Specific Commands

This section describes the HiFi Vorbis Decoder specific commands. They are listed in sections based on their primary command type (`i_cmd`). Each section contains a table for every subcommand. In the case of no subcommands, the one primary command is presented.

3.5.1 Configuration and Execution Errors

These errors can result from initialization or execution API calls:

- **XA_VORBISDEC_EXECUTE_NONFATAL_OV_HOLE**
The Ogg Vorbis bitstream is missing one or more packets. The application may ignore the error and continue with the decoding process.
- **XA_VORBISDEC_EXECUTE_NONFATAL_OV_NOTAUDIO**
The Ogg Vorbis bitstream contains a non-audio packet. The application may ignore the error and continue with the decoding process, in which case the decoder will skip the non-audio packet.
- **XA_VORBISDEC_EXECUTE_NONFATAL_OV_BADPACKET**
The Ogg Vorbis bitstream contains a packet with invalid mode or window size parameters. The application may ignore the error and continue the decoding process, in which case the decoder will skip the corrupted packet.
- **XA_VORBISDEC_EXECUTE_NONFATAL_OV_RUNTIME_DECODE_FLUSH_IN_PROGRESS**
This error indicates that because of the `RUNTIME_INIT` command, decode data flushing is in progress.
- **XA_VORBISDEC_EXECUTE_NONFATAL_OV_INVALID_STRM_POS**
This error indicates that the decoded frame stream position may be invalid.
- **XA_VORBISDEC_EXECUTE_NONFATAL_OV_INSUFFICIENT_DATA**
This error indicates that the decoder needs more bits for the processing.
- **XA_VORBISDEC_EXECUTE_NONFATAL_OV_UNEXPECTED_IDENT_PKT_RECEIVED**
This error indicates that the decoder has received an identification packet when decoding is in progress. Hence, re-init may be required.
- **XA_VORBISDEC_EXECUTE_NONFATAL_OV_UNEXPECTED_HEADER_PKT_RECEIVED**
This error indicates that the decoder has received a comment/setup packet when decoding is in progress. Hence, re-init may be required.
- **XA_VORBISDEC_CONFIG_NONFATAL_BAD_PARAM**
The config parameter is out of range.
- **XA_VORBISDEC_CONFIG_NONFATAL_GROUPED_STREAM**
This error indicates that the decoder has received an Ogg page corresponding to a different Ogg Vorbis stream. Grouped Ogg Vorbis streams are not supported.

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

- **XA_VORBISDEC_CONFIG_FATAL_BADHDR**
The stream contains invalid headers or a header checksum verification fails.
- **XA_VORBISDEC_CONFIG_FATAL_NOTVORBIS**
A packet header without the Vorbis identification string ("vorbis") is encountered at configuration time.
- **XA_VORBISDEC_CONFIG_FATAL_BADINFO**
The Vorbis identification header that provides the information about the audio stream is invalid.
- **XA_VORBISDEC_CONFIG_FATAL_BADVERSION**
The Vorbis version specified in the headers is incorrect, i.e., the `vorbis_version` field is different than 0 as required by the Ogg Vorbis I format specification.
- **XA_VORBISDEC_CONFIG_FATAL_INVALID_PARAM**
The config parameter is invalid.
- **XA_VORBISDEC_CONFIG_FATAL_BADBOOKS**
The codebook header is corrupted.
- **XA_VORBISDEC_CONFIG_FATAL_CODEBOOK_DECODE**
Codebook decode fails because of an invalid Huffman entry value.
- **XA_VORBISDEC_EXECUTE_FATAL_PERSIST_ALLOC**
The decoder runs out of persistent memory. The decoder has been tested on an extensive suite of Ogg Vorbis streams and this error is not expected to occur. In case it occurs, pre-config parameter `XA_VORBISDEC_CONFIG_PARAM_RUNTIME_MEM` can be used to increase persistent and scratch memory sizes (Refer to Table 3-6).
- **XA_VORBISDEC_EXECUTE_FATAL_CORRUPT_STREAM**
The stream is either unsupported or corrupt and cannot be decoded further.

- `XA_VORBISDEC_EXECUTE_FATAL_SCRATCH_ALLOC`

The decoder runs out of scratch memory. The decoder has been tested on an extensive suite of Ogg Vorbis streams and this error is not expected to occur. In case it occurs, pre-config parameter `XA_VORBISDEC_CONFIG_PARAM_RUNTIME_MEM` can be used to increase persistent and scratch memory sizes (Refer to Table 3-6).

- `XA_VORBISDEC_EXECUTE_FATAL_INSUFFICIENT_INP_BUF_SIZE`

The decoder input buffer is not sufficient to hold one full ogg page (or the last packet from previous page if that is a continued packet and next full page) from input stream. In case of this error pre-config parameter `XA_VORBISDEC_CONFIG_PARAM_OGG_MAX_PAGE_SIZE` can be used to increase input buffer size (Refer to Table 3-5).

- `XA_VORBISDEC_EXECUTE_FATAL_OUT_OF_SEQUENCE_API`

The decoder API command called out of sequence.

3.5.2XA_API_CMD_SET_CONFIG_PARAM

Table 3-1 XA_VORBISDEC_CONFIG_PARAM_COMMENT_MEM_PTR subcommand

| | |
|--------------------------|--|
| Subcommand | XA_VORBISDEC_CONFIG_PARAM_COMMENT_MEM_PTR |
| Description | <p>This command points the decoder to a user-allocated memory block. The pointer has no specific alignment requirements and the size of the block is set through the <code>XA_VORBISDEC_CONFIG_PARAM_COMMENT_MEM_SIZE</code> parameter (see Table 3-2). The decoder initializes this block with the vendor string and the user comment fields from the Ogg Vorbis stream. The format of the comment block is described in Section 3.3. Default value: <code>NULL</code>. The use of the comment memory block is optional: by default, the application skips over the comment header.</p> |
| Actual Parameters | <p><code>p_xa_module_obj</code> <code>api_obj</code> – Pointer to API Structure</p> <p><code>i_cmd</code> <code>XA_API_CMD_SET_CONFIG_PARAM</code></p> <p><code>i_idx</code> <code>XA_VORBISDEC_CONFIG_PARAM_COMMENT_MEM_PTR</code></p> <p><code>pv_value</code> <code>comment_mem_ptr</code> – Pointer to the comment memory block</p> |
| Restrictions | None. |

Example

```
char *comment = malloc(4096);
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_VORBISDEC_CONFIG_PARAM_COMMENT_MEM_PTR,
                  (pVOID) comment);
```

Errors

- Common API Errors

Table 3-2 XA_VORBISDEC_CONFIG_PARAM_COMMENT_MEM_SIZE subcommand

| | |
|--------------------------|---|
| Subcommand | XA_VORBISDEC_CONFIG_PARAM_COMMENT_MEM_SIZE |
| Description | <p>This command initializes the size of the user-allocated comment memory block specified through the <code>XA_VORBISDEC_CONFIG_PARAM_COMMENT_MEM_PTR</code> parameter (see Table 3-1). The format of the comment block is described in Section 3.3.</p> <p>Default value: 0. The use of the comment memory block is optional: by default, the application skips over the comment header.</p> |
| Actual Parameters | <p><code>p_xa_module_obj</code> <code>api_obj</code> – Pointer to API Structure</p> <p><code>i_cmd</code> <code>XA_API_CMD_SET_CONFIG_PARAM</code></p> <p><code>i_idx</code> <code>XA_VORBISDEC_CONFIG_PARAM_COMMENT_MEM_SIZE</code></p> <p><code>pv_value</code> <code>&comment_mem_size</code> – Pointer to comment memory size variable</p> |
| Restrictions | None |

Example

```
int comment_size = 4096;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_VORBISDEC_CONFIG_PARAM_COMMENT_MEM_SIZE,
                  (pVOID) &comment_size);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC
 - `pv_value` is NULL

Table 3-3 XA_VORBISDEC_CONFIG_PARAM_RAW_VORBIS_FILE_MODE subcommand

| | |
|--------------------------|--|
| Subcommand | XA_VORBISDEC_CONFIG_PARAM_RAW_VORBIS_FILE_MODE |
| Description | This command provides the file type of the currently decoded file. The 0 value indicates an Ogg Vorbis file. The 1 value indicates a raw Vorbis file without an Ogg container. The default value is 0. By default, the library assumes the file as an Ogg Vorbis file. |
| Actual Parameters | <p>p_xa_module_obj api_obj – Pointer to API Structure</p> <p>i_cmd XA_API_CMD_SET_CONFIG_PARAM</p> <p>i_idx XA_VORBISDEC_CONFIG_PARAM_RAW_VORBIS_FILE_MODE pv_value &file_type – Pointer to file type variable</p> |
| Restrictions | This command must be called before calling XA_CMD_TYPE_INIT_API_POST_CONFIG_PARAMS command. |

Example

```
int file_type = 1;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_VORBISDEC_CONFIG_PARAM_RAW_VORBIS_FILE_MODE,
                  (pVOID)&file_type);
```

Errors

■ Common API Errors

■ XA_API_FATAL_INVALID_CMD_TYPE

Configuration of Vorbis file mode is not allowed after calling XA_CMD_TYPE_INIT_API_POST_CONFIG_PARAMS, since memory size has been calculated by then.

■ XA_API_FATAL_MEM_ALLOC

pv_value is NULL

Table 3-4 XA_VORBISDEC_CONFIG_PARAM_RAW_VORBIS_LAST_PKT_GRANULE_POS subcommand

| | |
|--------------------------|--|
| Subcommand | XA_VORBISDEC_CONFIG_PARAM_RAW_VORBIS_LAST_PKT_GRANULE_POS |
| Description | This command provides the granule position of last packet of the currently decoded file. The default value is -1. |
| Actual Parameters | <p>p_xa_module_obj api_obj – Pointer to API Structure</p> <p>i_cmd XA_API_CMD_SET_CONFIG_PARAM</p> <p>i_idx XA_VORBISDEC_CONFIG_PARAM_RAW_VORBIS_LAST_PKT_GRANULE_POS pv_value &granule_pos – Pointer to granule position variable</p> |
| Restrictions | None |

Example

```

long long granule_pos = 1000000;
res = (*api_func) (api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_VORBISDEC_CONFIG_PARAM_RAW_VORBIS_LAST_PKT_GRANULE
                  _POS,
                  (pVOID) &granule_pos);

```

Errors

- Common API Errors
- XA_VORBISDEC_CONFIG_NONFATAL_BAD_PARAM
The parameter value is not within valid range
- XA_API_FATAL_MEM_ALLOC
pv_value is NULL

Table 3-5 XA_VORBISDEC_CONFIG_PARAM_OGG_MAX_PAGE_SIZE subcommand

| | |
|--------------------------|--|
| Subcommand | XA_VORBISDEC_CONFIG_PARAM_OGG_MAX_PAGE_SIZE |
| Description | <p>This command specifies the input buffer size (kB) required for ogg vorbis mode. Minimum value is 12, maximum is 128. The default value is 12.</p> <p>Note: The maximum page size specified in Ogg standard is 64 kB, however for decoding continued packets decoder needs the continued packet from last page and full next page in input buffer and hence input buffer size requirement can go beyond 64 kB. To handle those scenarios maximum is kept to 128. Suggested value for this parameter is max. packet size + max. ogg page size (in kB).</p> |
| Actual Parameters | <p>p_xa_module_obj api_obj - Pointer to API Structure</p> <p>i_cmd XA_API_CMD_SET_CONFIG_PARAM</p> <p>i_idx XA_VORBISDEC_CONFIG_PARAM_OGG_MAX_PAGE_SIZE</p> <p>pv_value &ogg_maxpage – Pointer to ogg max page size variable</p> |
| Restrictions | This command must be called before calling XA_CMD_TYPE_INIT_API_POST_CONFIG_PARAMS command. |

Example

```
int ogg_maxpage = 14;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_VORBISDEC_CONFIG_PARAM_OGG_MAX_PAGE_SIZE,
                  (pVOID) &ogg_maxpage);
```

Errors

- Common API Errors
- XA_API_FATAL_INVALID_CMD_TYPE

Maximum ogg page size is not allowed to be modified after calling XA_CMD_TYPE_INIT_API_POST_CONFIG_PARAMS, since memory sizes have been calculated by then.
- XA_VORBISDEC_CONFIG_NONFATAL_BAD_PARAM

The parameter value is not within valid range
- XA_API_FATAL_MEM_ALLOC

pv_value is NULL

Table 3-6 XA_VORBISDEC_CONFIG_PARAM_RUNTIME_MEM subcommand

| | |
|--------------------------|--|
| Subcommand | XA_VORBISDEC_CONFIG_PARAM_RUNTIME_MEM |
| Description | This command provides whether additional runtime memory required for stream with 8 kB blocksize is to be allocated or not. Minimum value is 0, maximum is 1. The default value is 0. 0 – Runtime memory allocation will be assuming 4 kB blocksize. 1 – Runtime memory allocation will be assuming 8 kB blocksize. |
| Actual Parameters | <p>p_xa_module_obj api_obj – Pointer to API Structure</p> <p>i_cmd XA_API_CMD_SET_CONFIG_PARAM</p> <p>i_idx XA_VORBISDEC_CONFIG_PARAM_RUNTIME_MEM</p> <p>pv_value &rtmem – Pointer to runtime memory variable</p> |
| Restrictions | This command must be called before calling XA_CMD_TYPE_INIT_API_POST_CONFIG_PARAMS command. |

Example

```
int rtmem = 1;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_VORBISDEC_CONFIG_PARAM_RUNTIME_MEM,
                  (pVOID) &rtmem);
```

Errors

- Common API Errors
- XA_API_FATAL_INVALID_CMD_TYPE

Runtime memory sizes are not allowed to be modified after calling XA_CMD_TYPE_INIT_API_POST_CONFIG_PARAMS, since memory sizes have been calculated by then.
- XA_VORBISDEC_CONFIG_NONFATAL_BAD_PARAM

The parameter value is not within valid range
- XA_API_FATAL_MEM_ALLOC

pv_value is NULL

3.5.3 XA_API_CMD_GET_CONFIG_PARAM

Table 3-7 XA_VORBISDEC_CONFIG_PARAM_SAMP_FREQ subcommand

| | |
|--------------------------|--|
| Subcommand | XA_VORBISDEC_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. The parameter becomes available after the codec has been initialized. |
| Actual Parameters | <p>p_xa_module_obj api_obj – Pointer to API Structure</p> <p>i_cmd XA_API_CMD_GET_CONFIG_PARAM</p> <p>i_idx XA_VORBISDEC_CONFIG_PARAM_SAMP_FREQ</p> <p>pv_value &samp_freq – Pointer to sampling frequency variable</p> |
| Restrictions | None |

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC
pv_value is NULL

Table 3-8 XA_VORBISDEC_CONFIG_PARAM_NUM_CHANNELS subcommand

| | |
|--------------------------|---|
| Subcommand | XA_VORBISDEC_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. This parameter becomes available after the codec has been initialized. |
| Actual Parameters | <p>p_xa_module_obj api_obj – Pointer to API Structure</p> <p>i_cmd XA_API_CMD_GET_CONFIG_PARAM</p> <p>i_idx XA_VORBISDEC_CONFIG_PARAM_NUM_CHANNELS</p> <p>pv_value &num_chan – Pointer to the channel count variable</p> |
| Restrictions | None |

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC
pv_value is NULL

Table 3-9 XA_VORBISDEC_CONFIG_PARAM_PCM_WDSZ subcommand

| | |
|--------------------------|---|
| Subcommand | XA_VORBISDEC_CONFIG_PARAM_PCM_WDSZ |
| Description | This command gets the output PCM word size in bits. The current version of the library always returns 16. |
| Actual Parameters | <p>p_xa_module_obj api_obj – Pointer to API Structure</p> <p>i_cmd XA_API_CMD_GET_CONFIG_PARAM</p> <p>i_idx XA_VORBISDEC_CONFIG_PARAM_PCM_WDSZ</p> <p>pv_value &pcm_wd_sz – Pointer to PCM word size variable</p> |
| Restrictions | None |

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC
pv_value is NULL

Table 3-10 XA_VORBISDEC_CONFIG_PARAM_GET_CUR_BITRATE subcommand

| | |
|--------------------------|--|
| Subcommand | XA_VORBISDEC_CONFIG_PARAM_GET_CUR_BITRATE |
| Description | This command gets the current bit rate. |
| Actual Parameters | <p>p_xa_module_obj api_obj – Pointer to API Structure</p> <p>i_cmd XA_API_CMD_GET_CONFIG_PARAM i_idx XA_VORBISDEC_CONFIG_PARAM_GET_CUR_BITRATE</p> <p>pv_value &bitrate – Pointer to the bit rate variable</p> |
| Restrictions | None |

Example

```
int bitrate;

res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_VORBISDEC_CONFIG_PARAM_GET_CUR_BITRATE,
                  (void *) &bitrate);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC
 - pv_value is NULL

4. Introduction to Example Test Bench

The supplied test bench contains the following files:

- Test bench source files (`test/src`)
 - `xa_vorbis_dec_error_handler.c`
 - `xa_vorbis_dec_test.c`
- Makefile to build the executable (`test/build`)
 - `makefile_testbench_sample`
- Sample parameter file to run the test bench (`test/build`)
 - `parameter.txt`

4.1 Building the Executable

To build the application, follow these steps:

1. Go to `test/build`.
2. At the command prompt, enter

```
xt-make -f makefile_testbench_sample clean all
```

This will build the decoder application `xa_vorbis_dec_test`.

Note: If you have source code distribution, you must build the `xa_vorbis_dec.a` library before you can build the test-bench. To build the library, follow these steps:

1. Go to the build directory.
2. Enter:

```
$xt-make clean all install
```

This will build the `xa_vorbis_dec.a` library and copy it to the `lib` directory.

4.2 Usage

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

The command line usage is as follows:

```
xt-run xa_vorbis_dec_test -ifile:<infile> -ofile:<outfile> [-r] [-g<granule_pos>] [-ogg_maxpage:<omp>] [-rtmem:<rtm>]
```

where,

<infile> is the name of the input Vorbis file.
<outfile> is the name of the output raw PCM file.
<granule_pos> specifies the granule position for last audio packet.
-r flag specifies that the current input file is a raw Vorbis file, without this flag, the current input file is an Ogg Vorbis file
<omp> specifies maximum ogg page size (in kB) which can fit in input buffer. Min- 12, max- 128. Default – 12.
<rtm> specifies whether to allocate extra runtime memory or not. Min- 0, max – 1. Default – 0 (Extra runtime mem not allocated).

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

The syntax for writing into the `parameter.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 Vorbis Decoder can be run for multiple test files using different command lines. The syntax for the 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 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.

5. References

- [1] The Xiph.Org Foundation: <http://www.xiph.org/>
- [2] Vorbis I Specification: http://www.xiph.org/vorbis/doc/Vorbis_I_spec.html
- [3] Ogg Vorbis I format specification: comment field and header specification:
<http://www.xiph.org/vorbis/doc/v-comment.html>