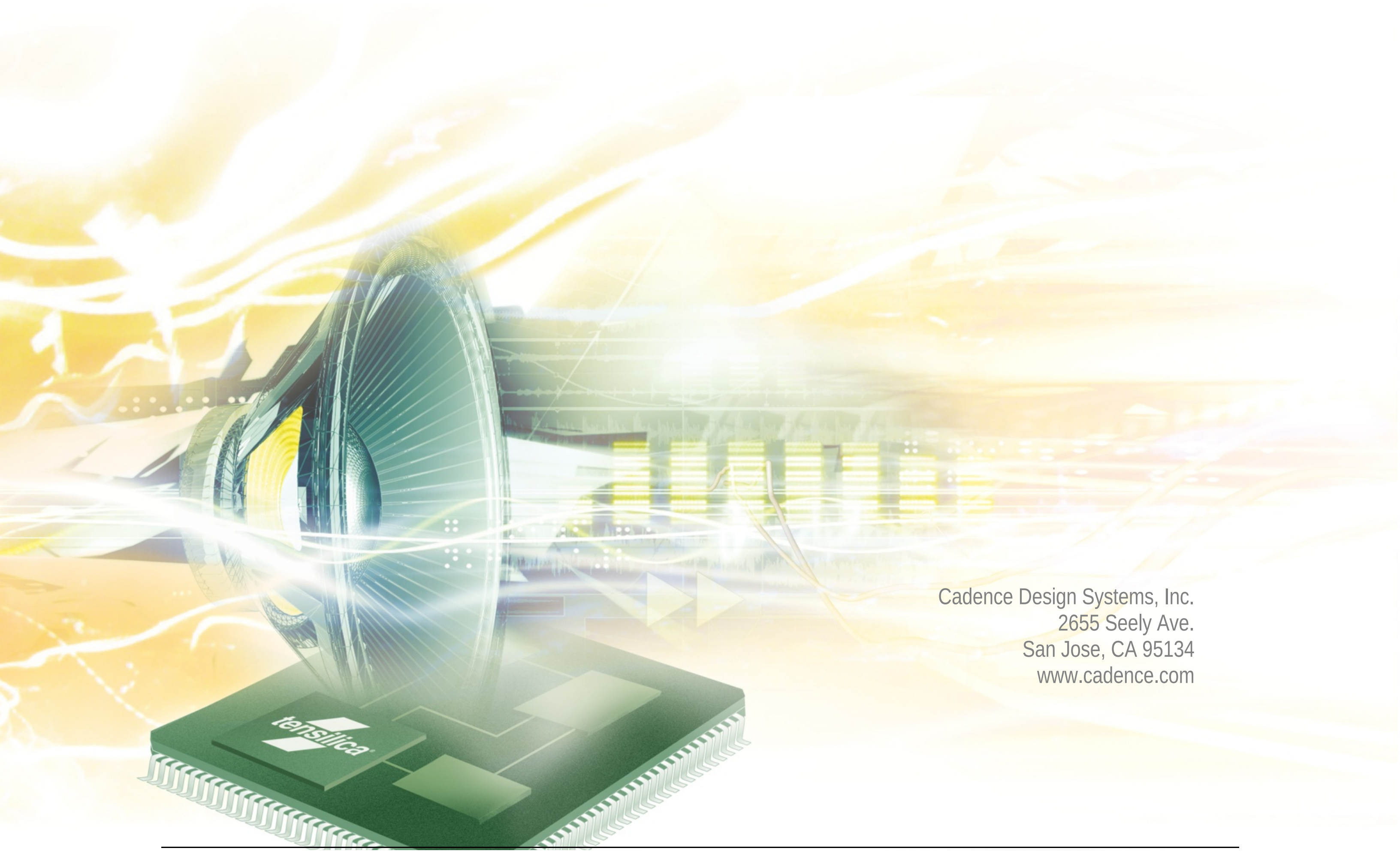
cādence®

BSAC Decoder

Programmer's Guide

For HiFi DSPs



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

Version 2.3

September 2017

Contents

Figures

Tables

Document Change History

Version	Changes
2.2	▪ Changed all generic references for HiFi 2 to HiFi. ▪ Added performance data for HiFi Mini/HiFi 3/HiFi 4. ▪ Added stream position APIs in Section 2.6. ▪ Reorganized error codes in Section 3.
2.3	▪ Added performance data for HiFi 3z in Section 1.4

1. Introduction to the HiFi BSAC Decoder

The HiFi DSP BSAC Decoder implements the MPEG-4 standard (ISO/IEC 14496-3 subpart 4) for scalable audio coding in conjunction with the AAC-LC coding tools. The rest of this document refers to the HiFi DSP BSAC Decoder simply as the BSAC Decoder.

1.1 BSAC Description

BSAC (Bit Sliced Arithmetic Coding) replaces the Huffman coding portion of the conventional AAC standard used for noiseless coding of scale factors and spectral data. The rest of the processing is identical to AAC.

BSAC offers fine grain audio scalability in the range from 16kbps to 64kbps in steps of 1kbps per audio channel. To obtain fine grain scalability, frequency lines of spectral data are grouped into layers with the base layer offering a data rate of 16kbps and each enhancement layer providing an additional data rate of 1kbps per audio channel. The audio quality is refined by adding higher layers and is gracefully degraded by removing layers. This process enables real-time adjustment to the quality of service (QoS) without changing the core coding algorithm running on the server.

In addition to fine grain scalability, BSAC provides error resilience through its Segmented Binary Arithmetic coding (SBA) mode. In this mode, the decoder greatly improves the audio quality of a signal transmitted over an error-prone channel such as a wireless network by starting and terminating arithmetic decoding at sub-frame boundaries.

1.2 Document Overview

This document covers all the information required to integrate the HiFi BASC Decoder 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 in Section 2 after the introduction. Section 3 covers all the features and information particular to the HiFi CA Decoder. Section 4 describes an example test bench application.

1.3 HiFi BSAC Decoder Specifications

The HiFi BASC Decoder supports the following features:

- Cadence Audio Codec API is used
- An MPEG-4 ISO/IEC 14496-3 subpart 4 compliant decoder supporting BSAC decoding in combination with AAC-LC coding tools [1].
- Sampling rates: 8 to 48 kHz
- Data rates: 16 to 64 kbps per audio channel scalable in steps of 1 kbps
- Mono and stereo channels
- Output PCM sample size: 16 bits or 24 bits (aligned in the 24 MSBs of a 32-bit word)
- Support for IS/MS, TNS and PNS
- Support for raw BSAC bitstreams only

This decoder implementation conforms to the test criteria specified by the standard.

1.4 HiFi BASC Decoder Performance

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

- The API structure sizes returned by XA_API_CMD_GET_API_SIZE is approximately 150 bytes.
- The memory table structure size returned by XA_API_CMD_GET_MEMTABS_SIZE is approximately 150 bytes.

1.4.1 Memory

Text (Kbytes)					Data	Run Time Memory (Kbytes)				
HiFi Mini	HiFi 2	HiFi 3	HiFi 3z	HiFi 4	(Kbytes)	Persistent	Scratch	Stack	Input	Output
32.3	35.9	37.0	38.7	40.7	22.9	4.7	28.5	1.0	2.1	8.0

Note The output buffer requirement is halved if the output PCM size is 16 bits.

1.4.2 Timings

Rate	Channels	Bit Rate	Average CPU Load (MHz)				
kHz		kbps	HiFi Mini	HiFi 2	HiFi 3	HiFi 3z	HiFi 4
44.1	2	64	27.4	27.4	24.9	21.2	21.2
48	2	128	46.3	46.4	42.8	35.9	36.3

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-filled cache model. The MCPS numbers for HiFi Mini/HiFi 3/HiFi 3z/HiFi 4 are obtained by running the test that is recompiled from the HiFi 2 source code in the HiFi Mini/HiFi 3/HiFi 3z/HiFi 4 configuration. No specific optimization is done for HiFi Mini/HiFi 3/HiFi 3z/HiFi 4.

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.

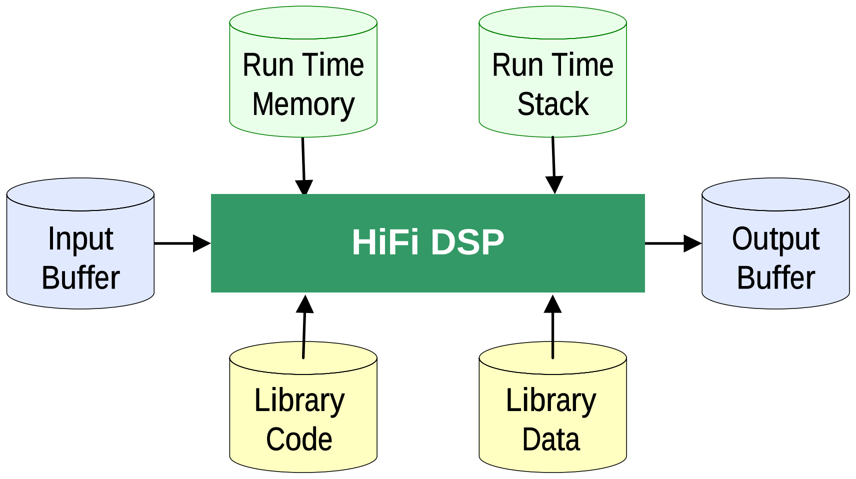


Figure 1: HiFi Audio Codec Interfaces

Section 2.1 discusses all the types of 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

This is the buffer used by the algorithm for accepting input data. Before the call to the codec, the input buffer needs to be completely filled with one frame worth of input data.

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, defined per codec library, 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>*_dec	
Description	This C API is the only access function to the audio codec.

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

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

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

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

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

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

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

When a value must be passed to a particular API command or an API command returns a value, the value expected or returned is passed through a pointer that 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 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 following command sequence overview flow chart. For each stage of the flow chart there is a section that lists the required commands in the order they should occur. For individual commands, definitions and examples refer to Section 2.6. The codecs have a common set of generic API commands that are represented by the white stages. The yellow stages are specific to each codec.

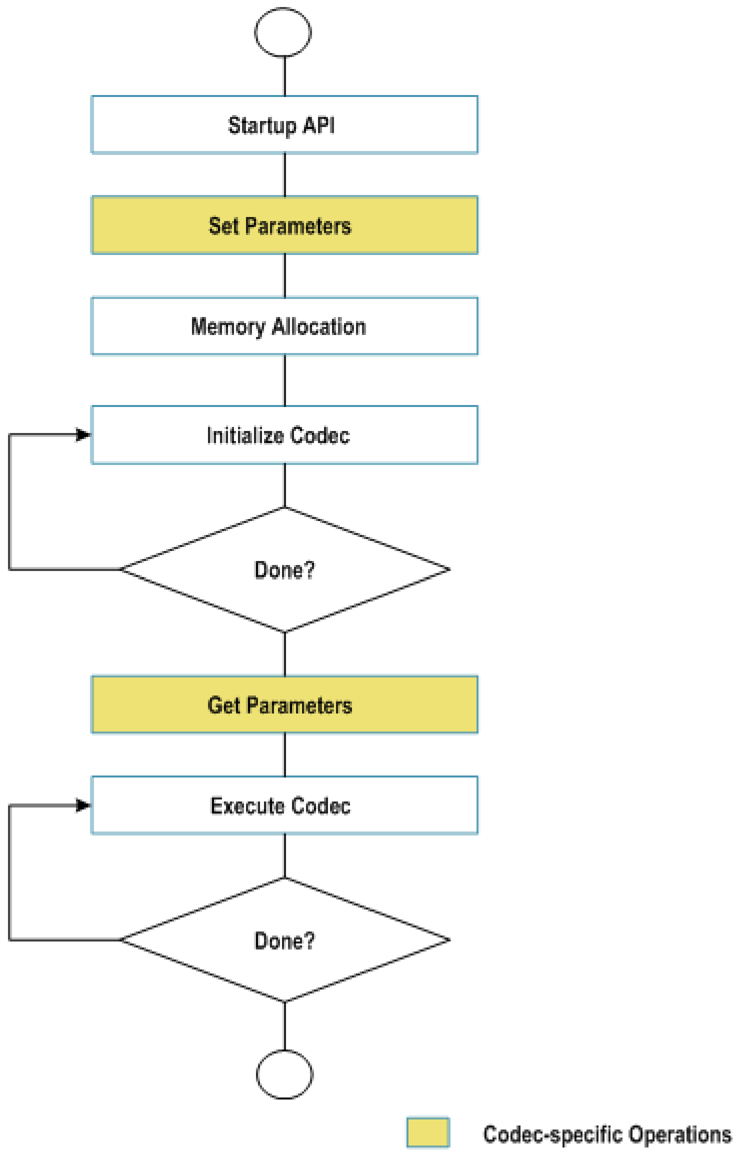


Figure 2: API Command Sequence Overview

2.4.1 Start-up API Stage

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

Table 2-2 Commands for Initialization

Command / Subcommand	Description
`XA_API_CMD_GET_LIB_ID_STRINGS` `XA_CMD_TYPE_LIB_NAME`	Gets the name of the library.
`XA_API_CMD_GET_LIB_ID_STRINGS` `XA_CMD_TYPE_LIB_VERSION`	Gets the version of the library.
`XA_API_CMD_GET_LIB_ID_STRINGS` `XA_CMD_TYPE_API_VERSION`	Gets the version of the API.
`XA_API_CMD_GET_API_SIZE`	Gets the size of the API structure.
`XA_API_CMD_INIT` `XA_CMD_TYPE_INIT_API_PRE_CONFIG_PARAMS`	Sets 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>*	Sets 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`	Gets the size of the memory structures to be allocated for the codec tables.
`XA_API_CMD_SET_MEMTABS_PTR`	Passes the memory structure pointer allocated for the tables.
`XA_API_CMD_INIT` `XA_CMD_TYPE_INIT_API_POST_CONFIG_PARAMS`	Calculates the required sizes for all the memory blocks based on the codec-specific parameters.
`XA_API_CMD_GET_N_MEMTABS`	Obtains the number of memory blocks required by the codec.

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

Table 2-5 Commands for Memory Allocation

Command / Subcommand	Description
`XA_API_CMD_GET_MEM_INFO_SIZE`	Gets the size of the memory type being referred to by the index.
`XA_API_CMD_GET_MEM_INFO_ALIGNMENT`	Gets the alignment information of the memory-type being referred to by the index.
`XA_API_CMD_GET_MEM_INFO_TYPE`	Gets the type of memory being referred to by the index.
`XA_API_CMD_GET_MEM_INFO_PRIORITY`	Gets the allocation priority of memory being referred to by the index.
`XA_API_CMD_SET_MEM_PTR`	Sets 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 PCM data and decoding the stream data. During the initialization of a decoder, the initialization task reads the input stream to discover the parameters of the encoding. However, for an encoder there is no header information in PCM data. Even so, the encoder application is still required to perform the initialization described in this stage. However, encoders will not consume data during initialization. Furthermore, this has an implication in that some encoders provide parameters that can be used to modify the input buffer data requirements after the initialization stage. These modifications will always be a reduction in the size. The application only needs to provide the reduced amount per execution of the main codec process.

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

Table 2-6 Commands for Codec Initialization

Command / Subcommand	Description
`XA_API_CMD_SET_INPUT_BYTES`	Sets 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`	Searches 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`	Checks if the initialization process has completed.
`XA_API_CMD_GET_CURIDX_INPUT_BUF`	Gets 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>`
	Gets 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`	Signals the end of bitstream to the library.
`XA_API_CMD_SET_INPUT_BYTES`	Sets the number of bytes available in the input buffer for the execution.
`XA_API_CMD_EXECUTE` `XA_CMD_TYPE_DO_EXECUTE`	Executes the codec thread.
`XA_API_CMD_EXECUTE` `XA_CMD_TYPE_DONE_QUERY`	Checks if the end of stream has been reached.
`XA_API_CMD_GET_OUTPUT_BYTES`	Gets the number of bytes output by the codec in the last frame.
`XA_API_CMD_GET_CURIDX_INPUT_BUF`	Gets 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 the block allocation

- `xa_type_def.h`

  All the types required for the API calls

2.6 HiFi API Command Reference

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

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

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

The errors returned by the API are detailed after each of the command definitions. However, there are a few errors that are common to all the API commands, which are listed in Section 0. 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 the 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, otherwise, it is set to zero. A pointer to the flag variable is passed as an argument.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_INIT`
	
	`i_idx`
	`XA_CMD_TYPE_INIT_DONE_QUERY`
	
	`pv_value`
	`&init_done` – Pointer to the flag that indicates the completion of the 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 the memory size variable

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

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is NULL

2.6.6 XA_API_CMD_SET_MEMTABS_PTR

Table 2-18 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 the memory size

Restrictions	None

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is NULL
- XA_API_FATAL_INVALID_CMD_TYPE

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

2.6.9 XA_API_CMD_GET_MEM_INFO_ALIGNMENT

Table 2-21 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 the allocation priority of memory being referred to by the index. (The meaning of the levels is defined on a codec-specific basis. This command returns a fixed dummy value unless the codec defines it otherwise.)

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

Restrictions	None

Example

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

Table 2-25 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 the memory buffer allocated

Restrictions	The pointer must be correctly aligned to the requirements.

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_API_FATAL_INVALID_CMD_TYPE

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

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

2.6.13 XA_API_CMD_INPUT_OVER

Table 2-27 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 the bytes consumed variable

Restrictions	None

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.16 XA_API_CMD_EXECUTE

Table 2-30 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, 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, 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. The command should be issued before the application starts feeding the decoder with new data from a random place in the input stream. 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 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 BSAC Decoder

The HiFi DSP BSAC decoder conforms to the generic codec API. The flow chart of the command sequence used in the example test bench is provided below.

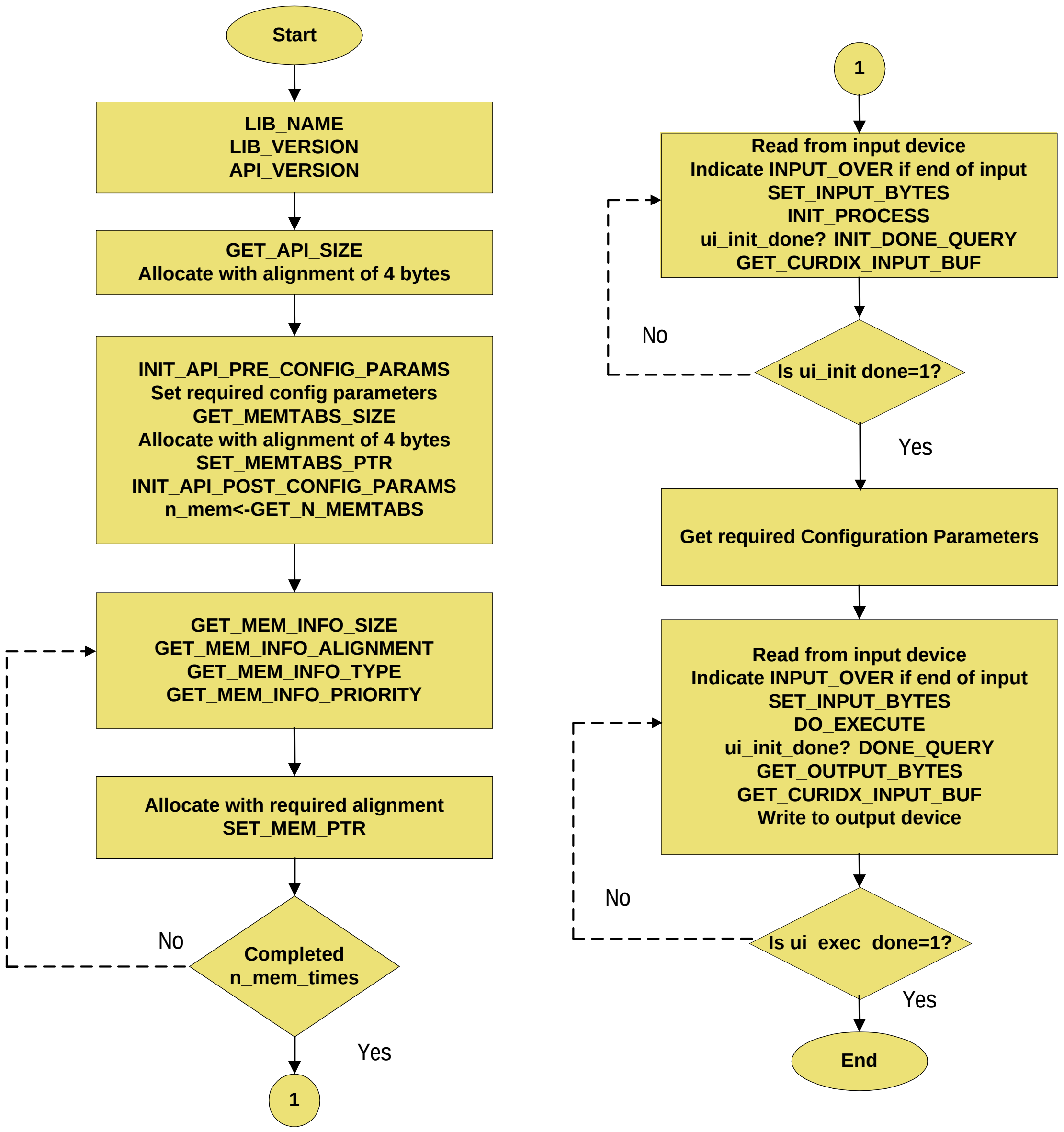


Figure 3: Flow Chart for BSAC Decoder Integration

3.1 Files Specific to the BSAC Decoder

The BSAC eDecoder Parameter Header File (`include/acc_dec`) is:

- `xa_aac _dec_api.h`

The BSAC Decoder Library File is:

- `xa_bsac_dec.a`

The BSAC Decoder API call is defined as:

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

3.2 BSAC Specific Error Codes

Other than the common error codes explained in Section 2, BSAC decoder APIs may also report error codes specific to BSAC decoder library. These errors are classified into three classes:

- API Errors
- Configuration Errors
- Execute Errors

To simplify the text, the following terminologies are used in this section:

- INIT API or INIT process:

  Calling the decoder API for XA_API_CMD_INIT command with subcommand XA_CMD_TYPE_INIT_PROCES.

- EXEC API or EXEC process:

  Calling the decoder API for XA_API_CMD_EXECUTE command with subcommand XA_CMD_TYPE_DO_EXECUTE.

- Config API:

  Calling the decoder API for XA_API_CMD_SET_CONFIG_PARAM or XA_API_CMD_GET_CONFIG_PARAM with any subcommand.

3.2.1 API Errors

API Errors are errors reported by the Decoder when the application tries to call the API command/subcommand when it is not supposed to be called. The API error specific to BSAC decoder library is explained below.

- XA_AACDEC_API_FATAL_INVALID_API_SEQ

  Description: This error is reported in case:

  - INIT API is called before memory is set.
  - EXEC API is called before INIT is successfully done.

  Required or suggested actions: Application code should be modified. Refer to Figure 3 for the correct API sequence.

- XA_AACDEC_API_NONFATAL_CMD_TYPE_NOT_SUPPORTED

  Description: This error is reported when a specific config API is not supported for the bitstream format under decoding.

  Note: This error is only reported by config APIs.

  Required or suggested actions: The application should not use the returned parameters. Application programmers may also consider modifying the code to avoid this non-fatal error.

3.2.2 Configuration Errors

Configuration errors are reported when a configuration subcommand fails. The failure may be due to an invalid config parameter value provided by the application or the config parameter queried is not yet read from the stream. Config APIs may also return the common errors described in Section 3.5.1 and API errors described in Section 3.5.2.

The following is the common error reported by configuration subcommands.

- XA_AACDEC_CONFIG_NONFATAL_PARAMS_NOT_SET

  Description: This error is reported when a specific parameter is not yet read from a field in the encoded stream.

  Required or suggested actions: The application should not use the returned parameter.

Configuration errors that are unique to specific configuration subcommands are explained in Sections 3.5.1 and 3.5.2 along with the configuration subcommands.

3.2.3 Execute Errors

Execute errors are errors occurred during the initialization or execution process. Typically, these errors are caused by but not limited to the following reasons:

- Invalid or missing configuration parameters
- Stream parsing errors

The following execute errors are specific to BSAC decoder:

- XA_AACDEC_EXECUTE_NONFATAL_INSUFFICIENT_FRAME_DATA

  Description: The input buffer has insufficient data for initialization or execution.

  Required or suggested actions: The application should feed more data into the input buffer.

- XA_AACDEC_EXECUTE_NONFATAL_RUNTIME_INIT_RAMP_DOWN

  Description: This non-fatal status code may be returned after a DO_EXECUTE call following a RUNTIME_INIT command. The output ramp down happens over one frame and this status code indicates that the ramp down is in progress.

  Required or suggested actions: This error is for information only and no explicit action is required from the application.

- XA_AACDEC_EXECUTE_FATAL_INIT_FAILED

  Description: This error is reported when an error is encountered in an INIT API call.

  Required or suggested actions: The application should handle the error and initialize the decoder with another stream to continue decoding.

- XA_AACDEC_EXECUTE_NONFATAL_RAW_FRAME_PARSE_ERROR
  XA_AACDEC_EXECUTE_FATAL_RAW_FRAME_PARSE_ERROR

  Description: These errors are reported by the decoder when it encounters errors while parsing a RAW frame during EXEC call.

  Required or suggested actions:

  - Case 1: FATAL error: The application should stop the decoding process. Call INIT API and feed another stream.
  - Case 2: NONFATAL error: The application can continue decoding without any action. The current frame data will be discarded. The application must feed the data from the next frame start. It is assumed that the application has the knowledge about frame boundary.

- XA_AACDEC_EXECUTE_FATAL_ERROR_IN_CHANROUTING

  Description: This error is reported when there is a conflict between the channel routing configured from the application and the number of channels present in the stream under decoding.

  Required or suggested actions: Application should reconfigure the channel routing specification and call the INIT_API to start the decoding again.

- XA_AACDEC_EXECUTE_FATAL_ZERO_FRAME_LENGTH

  Description: This error is reported when the length of a given frame read from the 'length field' of the frame header is zero.

  Required or suggested actions: The application should discard the current frame and start feeding the decoder from the next frame start. It is assumed that the application has the knowledge about the frame boundary in this case.

- XA_AACDEC_EXECUTE_FATAL_EMPTY_INPUT_BUFFER

  Description: This error is reported when INIT is called without any input data and input_over is set. This error indicates unsuccessful initialization.

  Required or suggested actions: The application should handle the error and initialize the decoder with another stream to continue decoding.

3.3 Configuration Parameters

The HiFi BSAC Decoder library accepts the following parameters from the user:

- `pcm_wdsz` – The PCM sample size in bits.
  - 16 – 16-bit PCM samples
  - 24 (default) – 24-bit PCM samples. The samples are stored in the 24 MSBs of each output 32-bit word; the 8 LSBs are set to 0.
- `externalsr` – The external sampling rate between 8 kHz and 48 kHz. The default value is 44100. For proper decoding of raw BSAC bitstreams, this parameter must be initialized by the application layer after parsing the bitstream container format headers.
- `externalchcfg` – The external channel configuration is either set to 1 for mono or 2 for stereo audio channel configuration. The default value is 2. For proper decoding of raw BSAC bitstreams, this parameter must be initialized by the application layer after parsing the bitstream container format headers.
- `layers` – Number of scalable (enhancement) layers to be decoded. This parameter may be specified for each frame. The BSAC decoder will process the minimum of the layers available in each frame and the number of layers specified through this parameter. The default value is 48. Each scalable layer enhances the data rate by 1 kbps per audio channel. For example, a `layers` value of 8 for a stereo stream corresponds to a data rate of 48 kbps:

  2 channels × (16 kbps base data rate + 8 layers × 1 kbps) = 48 kbps

- `to_stereo` – Duplication of mono signals to stereo
  - 0 means mono streams are presented as a single channel
  - 1 means mono stream are presented as two identical channels (default)

After the initialization, the configuration parameters related to the input stream and decoded audio can be obtained by getting the following parameter values:

- `samp_freq` – The output sample rate given in Hz. For the raw BSAC decoder, this value is always equal to the value of the `externalsr` parameter.
- `num_channels` – Number of decoded channels present in the output buffer (1 or 2).
- `pcm_wdsz` – The output PCM bit width (16 or 24). Note that the 24-bit samples are stored in the 24 MSBs of each output 32-bit word; the 8 LSBs are set to 0.
- `data_rate` – This command gets the data rate of the encoded stream. This value is available only after decoding a few (typically 10 to 15) frames of data.
- `chanmap` – This parameter specifies the order of the channels in the output buffer. The value of the *N*th nibble is set to the channel index available at offset *N* in the output PCM buffer. The nibbles corresponding to unused sample offsets are set to `0xF`. The output channel order can be controlled through the `ochanrouting` parameter.
  - Example: `chanmap = 0xFFFFFF20` 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. There are no decoded samples present at the remaining sample offsets.

3.4 API Usage Notes

Although the HiFi BSAC Decoder conforms to the generic codec API described in Section 2, take the following into account to ensure correct decoder operation:

- For correct decoding, the application must present a full BSAC frame in the input buffer. The frame length (number of bytes) is specified in the first 11 bits of a raw BSAC data block. Typically, presenting a full BSAC frame is not an issue when all layers are part of a single elementary stream. However, if the scalable layers are split into separate elementary streams, the application layer needs to reassemble the layers of a single frame before presenting them to the decoder. If the data for some layers is not available (*e.g.*, due to low bit rate conditions), the application layer must fill the available data and pad the remaining frame bytes with zeroes. It should also set the correct number of layers to be decoded.
- The application layer must ensure that the following parameters are set correctly: `externalsr`, `externalchcfg` and `layers`. See Section 3.3.

3.5 HiFi BSAC Specific Commands

This section contains the commands unique to the HiFi BSAC Decoder. They are listed in sections based on their primary command's 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 XA_API_CMD_SET_CONFIG_PARAM

Table 3-1 XA_AACDEC_CONFIG_PARAM_PCM_WDSZ subcommand

Subcommand	`XA_AACDEC_CONFIG_PARAM_PCM_WDSZ`
Description	This command sets the output PCM sample bit width to 16 or 24. Each 24-bit sample is aligned in the 24 MSBs of a 32-bit word in the output buffer.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	
	`i_idx`
	`XA_AACDEC_CONFIG_PARAM_PCM_WDSZ`
	
	`pv_value`
	`&pcm_wdsz` – Pointer to the PCM sample width variable

Restrictions	Valid values: 16 or 24 (default)

Example

```
int pcm_wdsz = 16;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_AACDEC_CONFIG_PARAM_PCM_WDSZ,
                  (void *) &pcm_wdsz);
```

Errors

- Common API Errors
- XA_AACDEC_CONFIG_FATAL_INVALID_PCM_WDSZ

  The specified PCM bit width is invalid

Table 3-2 XA_AACDEC_CONFIG_PARAM_EXTERNALSAMPLINGRATE subcommand

Subcommand	`XA_AACDEC_CONFIG_PARAM_EXTERNALSAMPLINGRATE`
Description	This command sets the sampling rate of the raw BSAC stream.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	
	`i_idx`
	`XA_AACDEC_CONFIG_PARAM_EXTERNALSAMPLINGRATE`
	
	`pv_value`
	`&externalsr` – Pointer to the external sample rate variable

Restrictions	Valid values are: 8000, 11025, 12000, 16000, 22050, 24000, 32000, 44100, 48000 Hz

Example

```
int externalsr;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_AACDEC_CONFIG_PARAM_EXTERNALSAMPLINGRATE,
                  (void *) &externalsr);
```

Errors

- Common API Errors
- XA_AACDEC_CONFIG_FATAL_INVALID_EXTERNALSAMPLINGRATE

  Value not valid

Table 3-3 XA_AACDEC_CONFIG_PARAM_EXTERNALCHCONFIG subcommand

Subcommand	`XA_AACDEC_CONFIG_PARAM_EXTERNALCHCONFIG`
Description	This command sets the audio channel configuration of the raw BSAC stream.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	
	`i_idx`
	`XA_AACDEC_CONFIG_PARAM_EXTERNALCHCONFIG`
	
	`pv_value`
	`&externalchcfg` – Pointer to the external channel configuration variable

Restrictions	Valid values: 1 (mono) or 2 (stereo)

Example

```
int externalchcfg;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_AACDEC_CONFIG_PARAM_EXTERNALCHCONFIG,
                  (void *) &externalchcfg);
```

Errors

- Common API Errors
- XA_AACDEC_CONFIG_FATAL_INVALID_EXTERNALCHCONFIG

  Value not valid

Table 3-4 XA_AACDEC_CONFIG_PARAM_EXTERNALBSFORMAT subcommand

Subcommand	`XA_AACDEC_CONFIG_PARAM_EXTERNALBSFORMAT`
Description	This command sets the bitstream format for the given stream.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	
	`i_idx`
	`XA_AACDEC_CONFIG_PARAM_EXTERNALCHCONFIG`
	
	`pv_value`
	`&externalbsformat` – Pointer to the external bitstream format variable

Restrictions	Valid values: 7 (XA_AACDEC_EBITSTREAM_TYPE_BSAC_RAW)

Example

```
int externalbsformat;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_AACDEC_CONFIG_PARAM_EXTERNALBSFORMAT,
                  (void *) &externalbsformat);
```

Errors

- Common API Errors
- XA_AACDEC_CONFIG_FATAL_INVALID_EXTERNALBSFORMAT

  Value not valid

Table 3-5 XA_AACDEC_CONFIG_PARAM_DECODELAYERS subcommand

Subcommand	`XA_AACDEC_CONFIG_PARAM_DECODELAYERS`
Description	This command sets the number of enhancements layers of the BSAC stream to be decoded. Each enhancement layer scales up the data rate by 1 kbps per audio channel.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	
	`i_idx`
	`XA_AACDEC_CONFIG_PARAM_DECODELAYERS`
	
	`pv_value`
	`&layers` – Pointer to the layer count variable

Restrictions	Valid values: 0 to 48

Example

```
int layers;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_AACDEC_CONFIG_PARAM_DECODELAYERS,
                  (void *) &layers);
```

Errors

- Common API Errors
- XA_AACDEC_CONFIG_FATAL_INVALID_DECODELAYERS

  Value not valid

Table 3-6 XA_AACDEC_CONFIG_PARAM_TO_STEREO subcommand

Subcommand	`XA_AACDEC_CONFIG_PARAM_TO_STEREO`
Description	This command enables or disables interleaving of mono to stereo in the output buffer. If enabled, the mono signal is replicated in two (stereo) output channels.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	
	`i_idx`
	`XA_AACDEC_CONFIG_PARAM_TO_STEREO`
	
	`pv_value`
	`&to_stereo` – Pointer to the stereo conversion flag variable

Restrictions	0 to disable; 1 to enable (default).

Example

```
int to_stereo;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_AACDEC_CONFIG_PARAM_TO_STEREO,
                  (void *) &to_stereo);
```

Errors

- Common API Errors
- XA_AACDEC_CONFIG_FATAL_INVALID_TO_STEREO

  Value not valid

Table 3-7 XA_AACDEC_CONFIG_PARAM_ZERO_UNUSED_CHANS subcommand

Subcommand	`XA_AACDEC_CONFIG_PARAM_ZERO_UNUSED_CHANS`
Description	This command enables (1) or disables (0) zeroing of unused output channels.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	
	`i_idx`
	`XA_AACDEC_CONFIG_PARAM_ZERO_UNUSED_CHANS`
	
	`pv_value`
	`&zero_unused_chans` – Pointer to the zero-unused-channels flag variable

Restrictions	Valid values: 0 (default) or 1

Example

```
int zero_unused_chans = 1;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_AACDEC_CONFIG_PARAM_ZERO_UNUSED_CHANS,
                  (void *) &zero_unused_chans);
```

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_AACDEC_CONFIG_FATAL_INVALID_ZERO_UNUSED_CHANS

  The zero-unused-channels flag must be 0 or 1

3.5.2 XA_API_CMD_GET_CONFIG_PARAM

Table 3-8 XA_AACDEC_CONFIG_PARAM_SAMP_FREQ subcommand

Subcommand	`XA_AACDEC_CONFIG_PARAM_SAMP_FREQ`
Description	This command gets the output sample rate.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_GET_CONFIG_PARAM`
	
	`i_idx`
	`XA_AACDEC_CONFIG_PARAM_SAMP_FREQ`
	
	`pv_value`
	`&samp_freq` – Pointer to the output sample rate variable

Restrictions	None

Example

```
int samp_freq;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_AACDEC_CONFIG_PARAM_SAMP_FREQ,
                  (void *) &samp_freq);
```

Errors

- Common API Errors

Table 3-9 XA_AACDEC_CONFIG_PARAM_NUM_CHANNELS subcommand

Subcommand	`XA_AACDEC_CONFIG_PARAM_NUM_CHANNELS`
Description	This command gets the number of output channels.

Actual Parameters	`P_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_GET_CONFIG_PARAM`
	
	`i_idx`
	`XA_AACDEC_CONFIG_PARAM_NUM_CHANNELS`
	
	`pv_value`
	`&num_channels` – Pointer to the output channel count variable

Restrictions	None

Example

```
int num_channels;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_AACDEC_CONFIG_PARAM_NUM_CHANNELS,
                  (void *) &num_channels);
```

Errors

- Common API Errors

Table 3-10 XA_AACDEC_CONFIG_PARAM_PCM_WDSZ subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_PCM_WDSZ
Description	This command gets the output sample width.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_GET_CONFIG_PARAM i_idx XA_AACDEC_CONFIG_PARAM_PCM_WDSZ pv_value &pcm_wdsz – Pointer to the PCM sample width variable
Restrictions	None

Example

```
int pcm_wdsz;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_AACDEC_CONFIG_PARAM_PCM_WDSZ,
                  (void *) &pcm_wdsz);
```

Errors

- Common API Errors

Table 3-11 XA_AACDEC_CONFIG_PARAM_DATA_RATE subcommand

Subcommand	`XA_AACDEC_CONFIG_PARAM_DATA_RATE`
Description	This command gets the data rate of the raw audio stream in bps.

Actual Parameters	`P_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_GET_CONFIG_PARAM`
	
	`i_idx`
	`XA_AACDEC_CONFIG_PARAM_DATA_RATE`
	
	`pv_value`
	`&data_rate` – Pointer to the data rate variable

Restrictions	None

Example

```
int data_rate;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_AACDEC_CONFIG_PARAM_DATA_RATE,
                  (void *) &data_rate);
```

Errors

- Common API Errors

Table 3-12 XA_AACDEC_CONFIG_PARAM_OUTNCHANS subcommand

Subcommand	`XA_AACDEC_CONFIG_PARAM_OUTNCHANS`
Description	This command gets the maximum number of decoded channels present in the output buffer. If a channel is not present in the encoded input stream, the corresponding sample value is set to zero in the output buffer.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_GET_CONFIG_PARAM`
	
	`i_idx`
	`XA_AACDEC_CONFIG_PARAM_OUTNCHANS`
	
	`pv_value`
	`&outnchans` – Pointer to the output channel count variable

Restrictions	Valid value: 2

Example

```
int outnchans;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_AACDEC_CONFIG_PARAM_OUTNCHANS,
                  (void *)&outnchans);
```

Errors

- Common API Errors

Table 3-13 XA_AACDEC_CONFIG_PARAM_CHANMAP subcommand

Subcommand	`XA_AACDEC_CONFIG_PARAM_CHANMAP`
Description	This parameter specifies how the channels are arranged in the output buffer. See Section 3.3.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_GET_CONFIG_PARAM`
	
	`i_idx`
	`XA_AACDEC_CONFIG_PARAM_CHANMAP`
	
	`pv_value`
	`&chanmap` – Pointer to the channel map variable

Restrictions	None

Example

```
int chanmap;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_AACDEC_CONFIG_PARAM_CHANMAP,
                  (void *) &chanmap);
```

Errors

- Common API Errors

Table 3-14 XA_AACDEC_CONFIG_PARAM_ACMOD subcommand

Subcommand	`XA_AACDEC_CONFIG_PARAM_ACMOD`
Description	This command gets the audio coding mode by identifying the channels encoded in the input bitstream. Details are described in Section 3.3.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_GET_CONFIG_PARAM`
	
	`i_idx`
	`XA_AACDEC_CONFIG_PARAM_ACMOD`
	
	`pv_value`
	`&acmod` – Pointer to the audio coding mode variable

Restrictions	None

Example

```
int acmod;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_AACDEC_CONFIG_PARAM_ACMOD,
                  (void *) &acmod);
```

Errors

- Common API Errors

Table 3-15 XA_AACDEC_CONFIG_PARAM_AAC_SAMPLERATE subcommand

Subcommand	`XA_AACDEC_CONFIG_PARAM_AAC_SAMPLERATE`
Description	This command gets the AAC sample rate.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_GET_CONFIG_PARAM`
	
	`i_idx`
	`XA_AACDEC_CONFIG_PARAM_AAC_SAMPLERATE`
	
	`pv_value`
	`&aac_samplerate` – Pointer to the AAC sample rate variable

Restrictions	None

Example

```
int aac_samplerate;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_AACDEC_CONFIG_PARAM_AAC_SAMPLERATE,
                  (void *) &aac_samplerate);
```

Errors

- Common API Errors

4. Introduction to the Example Test Bench

The supplied test bench consists of the following files:

- Test bench source files (found in `test/src`)
  - `xa_aac_dec_error_handler.c`
  - `xa_aac_dec_sample_testbench.c`
- Makefile to build the executable (`test/build`)
  - `makefile_testbench_sample`

4.1 Making the Executable

To build the application, follow the given steps:

1. Go to the path `test/build`.
2. At the command prompt, type:
   ```
   xt-make –f makefile_testbench_sample clean xa_bsac_dec_test
   ```

This will build the decoder example test bench `xa_bsac_dec_test`.

Note: If you have source code distribution, you must build the xa_bsac_dec.a 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 all install
   ```

This will build the xa_bsac_dec.a 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 parameters that the sample test bench accepts can be obtained by invoking the executable with the –h option on an Xtensa simulator

```
$       xt-run --turbo xa_bsac_dec_test -h
```

If no command line arguments are given, the application reads the commands from the parameter file `paramfilesimple.txt`

The syntax for writing the `paramfilesimple.txt` file is:

```
@Start

@Input_path <path to be appended to all input files>
@Output_path <path to be appended to all output files>
<command line 1>
<command line 2>
....

@Stop
```

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

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

5. References

[1] *ISO/IEC 14496-3: Information technology -- Coding of audio-visual objects -- Part 3: Audio* (MPEG-4)