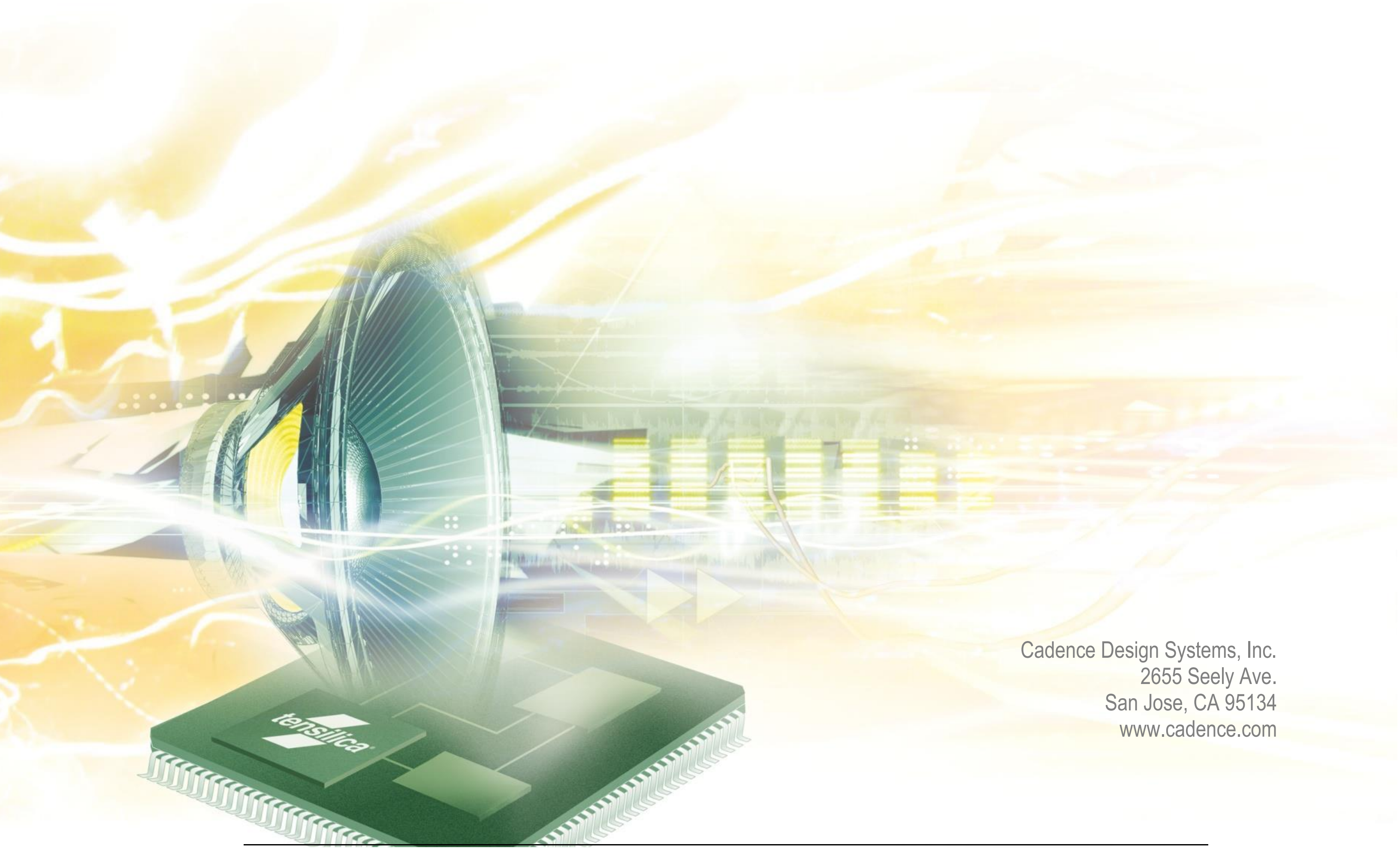
DRM Decoder

Programmer's Guide

For HiFi DSPs



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

Version 2.9
August 2017

Contents

Figures

Tables

Document Change History

Version	Changes
2.0	• New Implementation. • Includes xHE-AAC support and DRM+ support.
2.1	• Added new error codes, updated text in various places, performance numbers updated • Added new section for error concealment
2.2	• Revised Section 3.2 DRM Decoder Specific Error Codes to be consistent with the library header file • Updated Section 3.4 Concealment
2.3	• Upgraded to Fraunhofer reference code dated February 25, 2015. • Added a new error code for PARSE_ERROR in Section 3.2.3. • Added performance data for HiFi 4 in Section 1.5.
2.4	• Updated Memory and Timings in Section 1.5. • Added a new GET_CONFIG API CONCEAL_STATE and updated Section 3.4 regarding concealment strategy.
2.5	• Amended Section 3.4.
2.6	• Upgraded to Fraunhofer reference code dated January 25, 2017. • Added concealment API description in section 3.3.1 and updated section 3.4
2.7	• Upgraded to Fraunhofer reference code dated March 10, 2017. • Added EXECUTE_NONFATAL_NEED_DATA_SYNC_ERROR description in section 3.2.3
2.8	• Upgraded to Fraunhofer reference code dated July 06, 2017.

2.9	• Added performance data for HiFi 3z in Section 1.5.

1. Introduction to the HiFi DRM Decoder

The HiFi DSP DRM Decoder implements the Digital Radio Mondiale audio coding standard [1]. This decoder takes a DRM bitstream as input and outputs the decoded PCM samples.

The vendor source version of the software is Fraunhofer [2], [3] CDK DRM version 2.1.18 dated July 6, 2017.

1.1 DRM Description

The DRM (Digital Radio Mondiale) broadcasting system is standardized by the International Telecommunications Union (ITU), the International Electro-technical Committee (IEC), and the European Telecommunications Standard Institute (ETSI). It is the only open standard for digital broadcasting in the AM bands. The DRM system uses audio codecs and tools defined within the MPEG-4 audio standard (ISO/IEC 14496-3: High-Efficiency Advanced Audio Coding (HE-AAC) and ISO/IEC 23003-3: Extended High-Efficiency Advanced Audio Coding (xHE-AAC)) with some DRM-specific transport-related adjustments, such as audio super framing, intra frame interleaving and reordering, and CRC.

1.2 Organization of the HiFi DRM

The HiFi DRM Decoder is provided as a library and a sample player application, which uses the library to build a DRM sample player.

The DRM Decoder library implements decoding functionality for both HE-AAC and xHE-AAC. The decoder library accepts DRM super frames as input. The decoder uses SDC (Service Description Channel) type 9: Audio Information Data Entity configuration for decoding process. Both the DRM super frame and the configuration syntax are explained in the *Digital Radio Mondiale (DRM); System Specification* [1].

The DRM sample player test bench implements a “DRM audio decoder application”. The player uses the DRM decoder library to decode DRM super frames. The DRM test bench is described in Section 4.

1.3 Document Overview

This document covers all the information required to integrate the HiFi Audio Codecs into an application. The HiFi codec libraries implement a simple API to encapsulate the complexities of the coding operations and simplify the application and system implementation. Parts of the API are common to all the HiFi codecs; these are described in Section 2. Section 3 covers all the features and information specific to the DRM Decoder. Finally, the sample test bench is briefly described in Section 4.

The rest of this document refers to the decoder as either the HiFi DRM Decoder, or simply as the DRM Decoder.

1.4 HiFi DRM Decoder Specifications

The HiFi DSP DRM Decoder implements the following:

- Cadence Audio Codec API is used
- Audio Decoders:
  - A subset of MPEG-4 HE-AAC v2 decoder defined in ISO/IEC 14496-3 [4] with the “DRM specific usage” explained in section 5.4 of the DRM specification [1].
  - A subset of USAC decoder defined in ISO/IEC 23003-3 [5] with the “DRM specific usage explained in section 5.3 of the DRM specification [1].
- Sample rates and bit rate: All sampling rates and bitrates supported by the DRM specification [1].
- Output: Mono or Stereo Output (16 bit PCM data)
- Error Concealment

1.5 HiFi DRM Decoder Performance

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

- The API structure size returned by XA_API_CMD_GET_API_SIZE is approximately 480 bytes.
- The memory table structure size returned by XA_API_CMD_GET_MEMTABS_SIZE is approximately 150 bytes.

1.5.1Memory

Table 1-1 Memory

Text (kbytes)					Data	Run Time Memory (kbytes)				
HiFi Mini	HiFi 2	HiFi 3	HiFi 3z	HiFi 4	Kbytes	Persistent	Scratch	Stack	Input	Output
235.4	247.5	261.2	278.6	288.0	84.8	79.0	28.1	3.9	8.1	16.0

1.5.2 Timings

Table 1-2 Timings

Rate	Channels	Bit Rate	Average CPU Load (MHz)					
kHz		kbps	HiFi Mini	HiFi 2	HiFi 3	HiFi 3z	HiFi 4	
48	2	186	43.6	44.5	41.9	34.3	34.5	AAC
48	2	144	71.6	72.2	68.1	57.1	55.8	xHE-AAC

Note Performance specification measurements are carried out 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 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 that is common to all the HiFi audio codec libraries. The API facilitates any codec that works in the overall method shown in the following diagram.

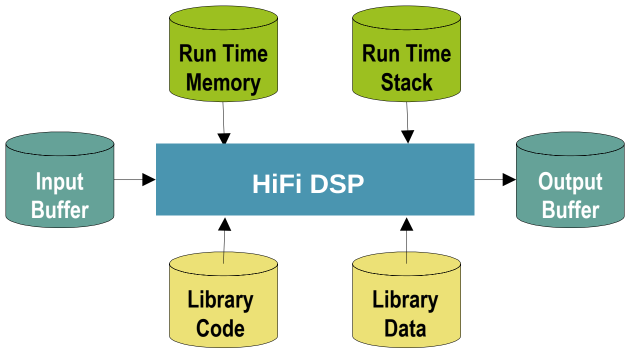


Figure 1 HiFi Audio Codec Interfaces

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

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

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

2.1 Memory Management

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

The 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.

2.1.1API Object

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

2.1.2API Memory Table

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

2.1.3Persistent Memory

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

2.1.4Scratch Memory

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

2.1.5Input 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 input data.

2.1.6Output Buffer

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

2.2 C Language API

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

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

Table 2-1 Codec API

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

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

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

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

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

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

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

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

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

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

2.3 Generic API Errors

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

Table 2-2 Error Codes Format

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

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

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

2.4 Commands

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

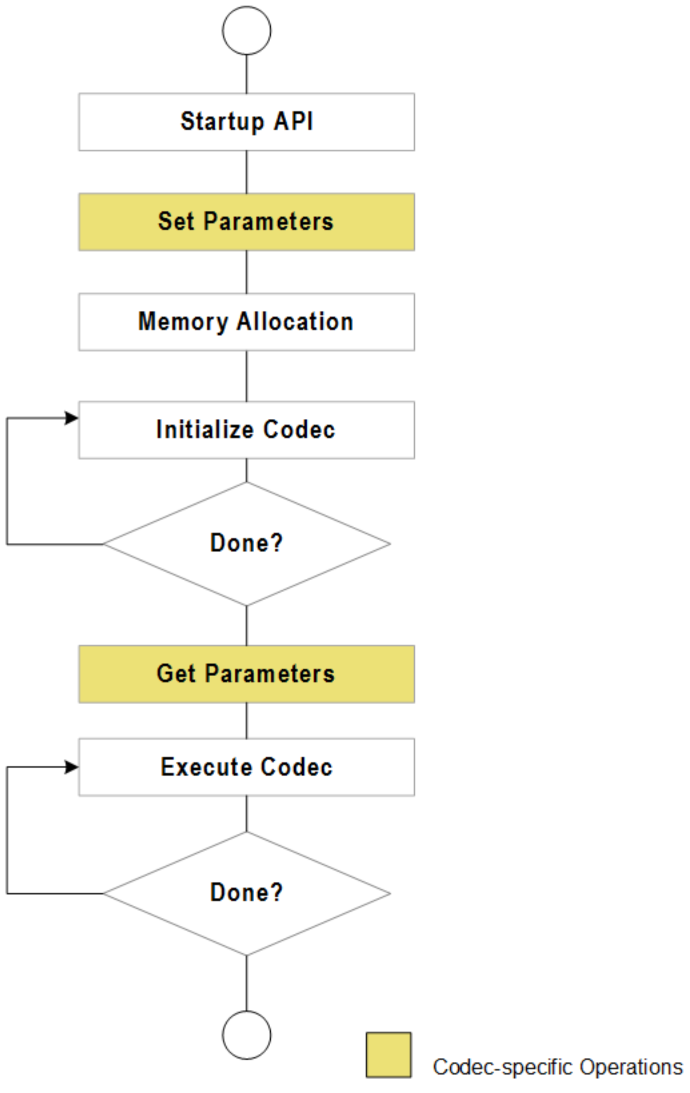


Figure 2 API Command Sequence Overview

2.4.1 Start-up API Stage

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

Table 2-3 Commands for Initialization

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

2.4.2 Set Codec-Specific Parameters Stage

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

Table 2-4 Commands for Setting Parameters

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

2.4.3Memory Allocation Stage

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

Table 2-5 Commands for Initial Table Allocation

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

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

Table 2-6 Commands for Memory Allocation

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

2.4.4Initialize Codec Stage

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

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

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

Table 2-7 Commands for Initialization

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

2.4.5Get Codec-Specific Parameters Stage

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

Table 2-8 Commands for Getting Parameters

Command / Subcommand	Description

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

2.4.6 Execute Codec Stage

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

Table 2-9 Commands for Codec Execution

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

2.5 Files Describing the API

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_error_handler.h`

  The macros and definition of the error handler

- `xa_memory_standards.h`

  The definitions for memory block allocation

- `xa_type_def.h`

  All the types required for the API calls

2.6 HiFi API Command Reference

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

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

The commands are followed by an example C call. Along with the call there is a definition of the variable types used. This is to avoid any confusion over the type of the 4th argument. The examples are not complete C code extracts as there is no initialization of the variables before they are used.

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

2.6.1 Common API Errors

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

- XA_API_FATAL_MEM_ALLOC

  `p_xa_module_obj` is `NULL`

- XA_API_FATAL_MEM_ALIGN

  `p_xa_module_obj` is not aligned to 4 bytes

- XA_API_FATAL_INVALID_CMD

  `i_cmd` is not a valid command

- XA_API_FATAL_INVALID_CMD_TYPE

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

2.6.2 XA_API_CMD_GET_LIB_ID_STRINGS

Table 2-10 XA_CMD_TYPE_LIB_NAME subcommand

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

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

Restrictions	None

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

Example

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

Errors

- XA_API_FATAL_MEM_ALLOC

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

- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

Table 2-11 XA_CMD_TYPE_LIB_VERSION subcommand

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

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

Restrictions	None

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

Example

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

Errors

- XA_API_FATAL_MEM_ALLOC

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

- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

Table 2-12 XA_CMD_TYPE_API_VERSION subcommand

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

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

Restrictions	None

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

Example

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

Errors

- XA_API_FATAL_MEM_ALLOC

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

- XA_API_FATAL_MEM_ALLOC

  `pv_`value is `NULL`

2.6.3XA_API_CMD_GET_API_SIZE

Table 2-13 XA_API_CMD_GET_API_SIZE command

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

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

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

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

Example

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

Errors

- XA_API_FATAL_MEM_ALLOC

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

- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.4XA_API_CMD_INIT

Table 2-14 XA_CMD_TYPE_INIT_API_PRE_CONFIG_PARAMS subcommand

Subcommand	`XA_CMD_TYPE_INIT_API_PRE_CONFIG_PARAMS`
Description	This command is used to set the default value of the configuration parameters. The configuration parameters can then be altered by using one of the codec-specific 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 2-15 XA_CMD_TYPE_INIT_API_POST_CONFIG_PARAMS subcommand

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

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

Restrictions	None

Example

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

Errors

- Common API Errors

Table 2-16 XA_CMD_TYPE_INIT_PROCESS subcommand

Subcommand	`XA_CMD_TYPE_INIT_PROCESS`
Description	This command initializes the codec. 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 2-17 XA_CMD_TYPE_INIT_DONE_QUERY subcommand

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

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

Restrictions	None

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.5XA_API_CMD_GET_MEMTABS_SIZE

Table 2-18 XA_API_CMD_GET_MEMTABS_SIZE command

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

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

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

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.6XA_API_CMD_SET_MEMTABS_PTR

Table 2-19 XA_API_CMD_SET_MEMTABS_PTR command

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

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

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

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_API_FATAL_MEM_ALIGN

  `pv_value` is not aligned to 4 bytes

2.6.7 XA_API_CMD_GET_N_MEMTABS

Table 2-20 XA_API_CMD_GET_N_MEMTABS command

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

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

Restrictions	None

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.8 XA_API_CMD_GET_MEM_INFO_SIZE

Table 2-21 XA_API_CMD_GET_MEM_INFO_SIZE command

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

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

Restrictions	None

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_API_FATAL_INVALID_CMD_TYPE

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

2.6.9 XA_API_CMD_GET_MEM_INFO_ALIGNMENT

Table 2-22 XA_API_CMD_GET_MEM_INFO_ALIGNMENT command

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

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

Restrictions	None

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_API_FATAL_INVALID_CMD_TYPE

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

2.6.10 XA_API_CMD_GET_MEM_INFO_TYPE

Table 2-23 XA_API_CMD_GET_MEM_INFO_TYPE command

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

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

Restrictions	None

Example

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

Table 2-24 Memory Type Indices

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_API_FATAL_INVALID_CMD_TYPE

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

2.6.11 XA_API_CMD_GET_MEM_INFO_PRIORITY

Table 2-25 XA_API_CMD_GET_MEM_INFO_PRIORITY command

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

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

Restrictions	None

Example

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

Table 2-26 Memory Priorities

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_API_FATAL_INVALID_CMD_TYPE

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

2.6.12 XA_API_CMD_SET_MEM_PTR

Table 2-27 XA_API_CMD_SET_MEM_PTR Command

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

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

Restrictions	The pointer must be correctly aligned to the requirements.

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_API_FATAL_INVALID_CMD_TYPE

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

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

2.6.13 XA_API_CMD_INPUT_OVER

Table 2-28 XA_API_CMD_INPUT_OVER command

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

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

Restrictions	None

Example

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

Errors

- Common API Errors

2.6.14 XA_API_CMD_SET_INPUT_BYTES

Table 2-29 XA_API_CMD_SET_INPUT_BYTES command

Subcommand	None
Description	This command sets the number of bytes available in the input buffer for the codec. It is used both in the initialization loop and execute loop. It is the number of valid bytes from the buffer pointer. The application fills at least one super frame in the beginning and maintains one more super frame before invoking EXEC_API of the DRM decoder.

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

Restrictions	None

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.15 XA_API_CMD_GET_CURIDX_INPUT_BUF

Table 2-30 XA_API_CMD_GET_CURIDX_INPUT_BUF command

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

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

Restrictions	None

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.16 XA_API_CMD_EXECUTE

Table 2-31 XA_CMD_TYPE_DO_EXECUTE subcommand

Subcommand	`XA_CMD_TYPE_DO_EXECUTE`
Description	This command executes the codec.

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

Restrictions	None

Example

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

Errors

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

Table 2-32 XA_CMD_TYPE_DONE_QUERY subcommand

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

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

Restrictions	None

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.17 XA_API_CMD_GET_OUTPUT_BYTES

Table 2-33 XA_API_CMD_GET_OUTPUT_BYTES command

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

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

Restrictions	None

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

;

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 till the start of the input buffer. This running counter is set to zero at the 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. **Note**: The current stream position for DRM streams will be in the super frame boundary as the decoder consumes the complete super frame.

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. **Note:** The behavior of this API is different for xHE-AAC streams and AAC streams. For xHE-AAC streams, this points to the position of the first byte of the AU. For AAC streams, this points to the position of the first byte of the super frame in which the AU is present. Due to UEP, it is difficult to mark the start position of the AU in a useful way.

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 DRM Decoder

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

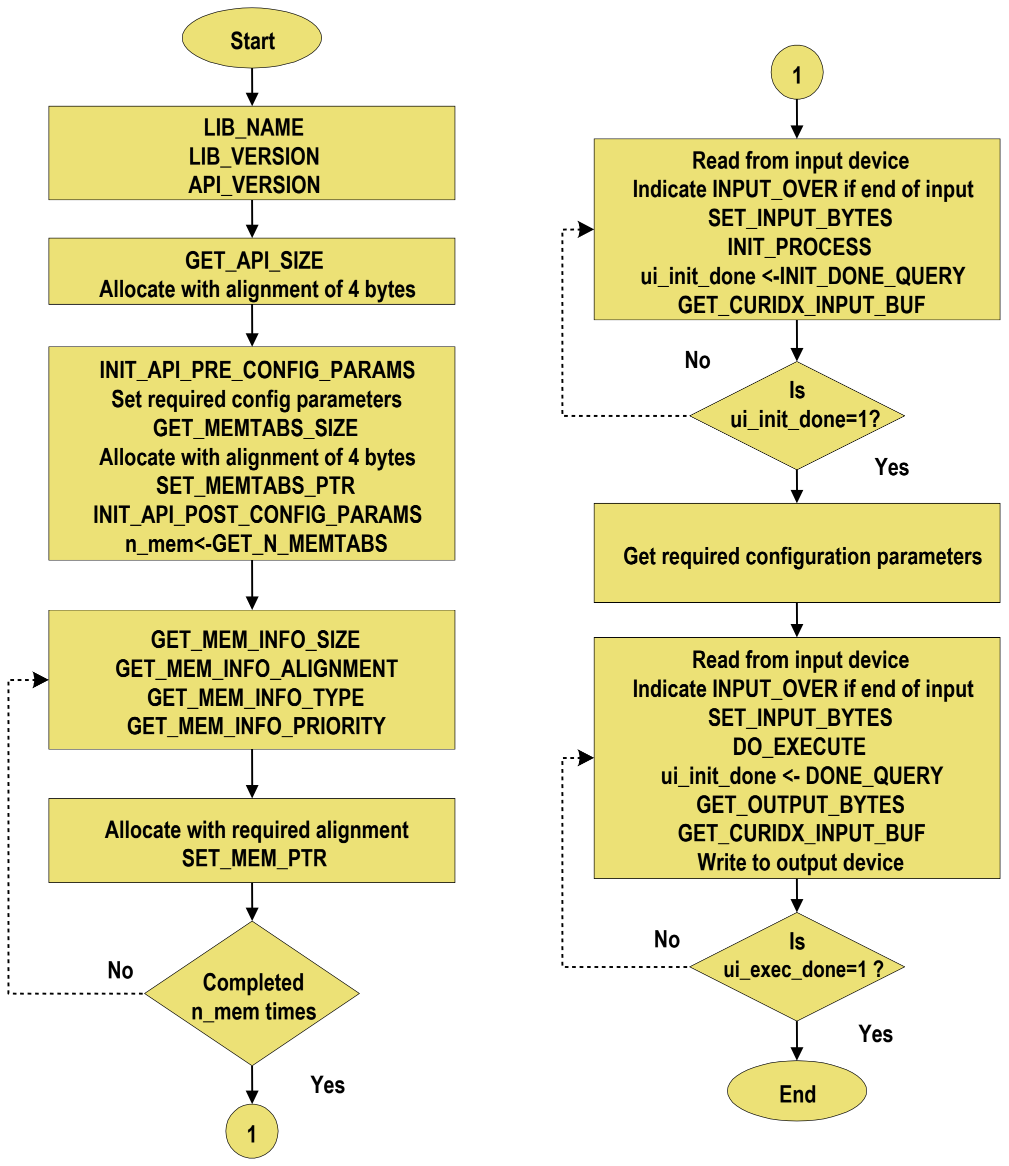


Figure 3 Flow Chart for DRM Decoder Integration

The HiFi DRM Decoder conforms to the generic codec API with decoder functionality.

Section 3.1 provides the file names and details of API calls specific to the DRM Decoder library. DRM Decoder-specific error codes and commands are described in Section 3.2.

3.1 Files and API Calls Specific to the DRM Decoder

DRM Decoder Parameter Header File (include/drm_dec/)

- `xa_drm_dec_api.h`

DRM Decoder Library Files

- `xa_drm_dec.a`

The DRM Decoder API call is defined as:

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

3.2 DRM Decoder Specific Error Codes

Other than common error codes explained in Section 2, the DRM Decoder may also report error codes specific to DRM decoding. 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_PROCESS

- 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

- DRM Configuration

  The content of interface header includes SDC configuration Type 9, which also includes audio codec specific configurations. This can be derived from the interface header in the DRM file or directly from SDC.

3.2.1API 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.

For example:

- INIT/EXEC API cannot be called before allocating required memories
- EXEC API cannot be called before a successful call of INIT API
- Specific config parameters can be obtained only after successful INIT

The API errors specific to DRM decoder libraries are:

- XA_DRM_DEC_API_FATAL_INVALID_API_SEQ

  Description: The command is issued when the API command being called is not allowed and the decoding process cannot continue as this command causes a fatal error.

  Required or suggested actions: The application should be modified to fix this error with the correct calling sequence.

- XA_DRM_DEC_API_NONFATAL_INVALID_API_SEQ

  Description: The command is issued when the API command being called is not allowed and ignored. However, the decoding process can still continue.

  Required or suggested actions: The application should not use the returned parameter, even if it is modified.

3.2.2Configuration 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 if the config parameter queried is not yet read from the stream. These errors can also be due to incorrect usage of config APIs for the given stream format. Config APIs may also return the common errors described in Section 2 and API errors described in section 3.2.1.

The following are the common errors reported by configuration subcommands.

- XA_DRM_DEC_CONFIG_FATAL_INVALID_PARAM

  Description: The user config parameter provided by the application is incorrect or out of range.

  Required or suggested actions: The application should validate the parameter value for its correctness and range before passing to the library.

- XA_DRM_DEC_CONFIG_FATAL_UNSUPPORTED_FORMAT

  Description: The input stream given by the user is not supported by the decoder.

  Required or suggested actions: The application will notify the user that the specified input is an unsupported format and it will stop the decoding process of the current stream.

- XA_DRM_DEC_CONFIG_NONFATAL_PARAMS_NOT_SET

  Description: The parameter value is not read / parsed from the stream or configuration.

  Required or suggested actions: The application should not use the returned parameter, even if it is modified.

- XA_DRM_DEC_CONFIG_NONFATAL_INVALID_GEN_STRM_POS

  Description: Query for the generated stream position is done when there is no output generated.

  Required or suggested actions: The application should not use the returned parameter, even if it is modified.

- XA_DRM_DEC_CONFIG_FATAL_UNSUPPORTED_BITRATE

  Description: The DRM decoder supports a bitrate up to 186kbps, the maximum bitrate supported by the DRM standard. If the input super frame length exceeds this limit, the decoder reports this error.

  Required or suggested actions: The decoder cannot decode the stream. The application should feed stream with supported bitrate.

- XA_DRM_DEC_CONFIG_FATAL_CONFIG_CHANGES_EXCEEDS_LIMIT

  Description: The decoder can buffer configuration changes up to four instances, however, if the number of config changes exceeds four, this error is reported. This scenario will not occur if the number of changes is not more than four within the 8k input buffer.

  Required or suggested actions: The application should wait till the configurations queued are applied and delay feeding the new input configuration and corresponding input.

3.2.3Execute 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
- Wrong input and output buffer settings
- Stream parsing errors

The following execute errors are specific to the DRM decoder:

- XA_DRM_DEC_EXECUTE_FATAL_PARSE_ERROR

  Description: This error occurs when the decoding process encounters an error while parsing various parameters, particularly variable length parameters. The decoder generates no output or generates concealed output. The decoder output for the subsequent frames may also be affected due to this error.

  Required or suggested actions: If it is an expected error (if frame corruption is possible), the application can continue decoding; if not, the application should stop decoding the current stream and feed the decoder with another stream. If the error is fatal, decoding the current stream may not be possible and the decoder may require to be initialized again.

- XA_DRM_DEC_EXECUTE_FATAL_INIT_ERROR

  Description: This error occurs when the INIT process fails in the decoder, which can happen due to inconsistent configuration parameters.

  Required or suggested actions: The application should re-check the values of the configuration parameters set to configure the decoder and it should not continue with the decoding process.

- XA_DRM_DEC_EXECUTE_NONFATAL_STREAM_CHANGE_DETECTED

  Description: Either the sample rate or the number of output channels is changed.

  Required or suggested actions: The application may re-configure the output handling device. If the reported error is FATAL, initializing the decoder again using INIT API is required. (A fatal error can be due to a codec change from xHE-AAC to AAC, or vice versa).

- XA_DRM_DEC_EXECUTE_NONFATAL_INSUFFICIENT_FRAME_DATA

  Description: The input buffer does not have sufficient data to decode one frame.

  Required or suggested actions: The application should add at least one complete super frame of data into the input buffer and set the input length appropriately.

- XA_DRM_DEC_EXECUTE_NONFATAL_CRC_ERROR_CONCEALED

  Description: The decoder has encountered an internal CRC error and the frame is concealed.

  Required or suggested actions: None.

- XA_DRM_DEC_EXECUTE_NONFATAL_PARSE_ERROR

  Description: The decoder could not parse the superframe header; therefore the current frame cannot be decoded. The same error is reported for all the frames in this superframe. The frame is concealed by the decoder.

  Required or suggested actions: None.

- XA_DRM_DEC_EXECUTE_NONFATAL_NEED_DATA_SYNC_ERROR

  Description: For xHE-AAC encoded input, at the start of execution the decoder parser will be in 'obtain-sync' state where it tries to acquire at-least two good audio-units (AUs) from the super-frame and then update the state to 'in-sync'. If the first super-frame doesn't provide the AUs to satisfy this condition with hold-off disabled (-os 0, user configurable option), this error is reported for such a super-frame.

  If hold-off is enabled (-os 1) all the AUs in the super frame are concealed by the decoder by inserting appropriate silence output frames and this error is not returned by the decoder library.

  Required or suggested actions: The application may insert mute frames to compensate for any underflow at the output buffer.

3.3 Configuration Parameters

The application can configure the DRM decoder using the "SET CONFIG" API. The application can read parameters specific to the decoded stream and the current context of decoding process using the "GET_CONFIG" API. These configuration parameters are explained in detail in Sections 3.3.1 and 3.3.2.

3.3.1XA_API_CMD_SET_CONFIG_PARAM

Table 3-1 XA_DRM_DEC_CONFIG_PARAM_OUTPUT_MODE subcommand

Subcommand	`XA_DRM_DEC_CONFIG_PARAM_OUTPUT_MODE`
Description	This command sets the expected output mode for the PCM. If the value is set to 0 (XA_DRM_DEC_OUTPUT_NORMAL), the number of channels is decided by the input stream. XA_DRM_DEC_OUTPUT_MODE is defined in xa_drm_dec_api.h

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	
	`i_idx`
	`XA_DRM_DEC_CONFIG_PARAM_OUTPUT_MODE`
	
	`pv_value`
	`& out_mode` – Pointer to the output mode variable

Restrictions	This is a WRITE ONLY parameter and the equivalent GET_CONFIG is not supported. The application can use OUT_NUM_CHANNELS to get the number of output channels.

Example

```
XA_DRM_DEC_OUTPUT_MODE out_mode = XA_DRM_DEC_OUTPUT_STEREO;
res =(*api_func)(api_obj,
        XA_API_CMD_SET_CONFIG_PARAM,
        XA_DRM_DEC_CONFIG_PARAM_OUTPUT_MODE
        (pVOID) & out_mode);
```

Errors

- Common Config and API Errors

Table 3-2 XA_DRM_DEC_CONFIG_PARAM_QMF_MODE subcommand

Subcommand	`XA_DRM_DEC_CONFIG_PARAM_QMF_MODE`
Description	This command sets the QMF mode for SBR. It is recommended to use XA_DRM_DEC_QMF_AUTO so that the decoder can decide the QMF mode appropriately. XA_DRM_DEC_QMF_MODE is defined in xa_drm_dec_api.h

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	
	`i_idx`
	`XA_DRM_DEC_CONFIG_PARAM_QMF_MODE`
	
	`pv_value`
	`& qmf_mode` – Pointer to the QMF mode variable

Restrictions	If XA_DRM_DEC_QMF_LOW_POWER is used, the PS decoding will not happen, thus the stream will be decoded as MONO only. The default value is XA_DRM_DEC_QMF_AUTO.

Example

```
XA_DRM_DEC_QMF_MODE qmf_mode = XA_DRM_DEC_QMF_HIGH_QUALITY;
res =(*api_func)(api_obj,
        XA_API_CMD_SET_CONFIG_PARAM,
        XA_DRM_DEC_CONFIG_PARAM_QMF_MODE,
        (pVOID) & qmf_mode);
```

Errors

- Common Config and API Errors

Table 3-3 XA_DRM_DEC_CONFIG_PARAM_DRM_AUDIO_CONFIG subcommand

Subcommand	`XA_DRM_DEC_CONFIG_PARAM_DRM_AUDIO_CONFIG`
Description	This command feeds the input configuration (the interface header derived from SDC). The interface header must have been read previously. XA_DRM_DEC_DRM_AUDIO_CONFIG is defined in xa_drm_dec_api.h

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	
	`i_idx`
	`XA_DRM_DEC_CONFIG_PARAM_DRM_AUDIO_CONFIG`
	
	`pv_value`
	`&drm_audio_config` – Pointer to interface structure

Restrictions	Refer to Section 4.3 for more details about the Interface Header. This is a WRITE ONLY parameter and the equivalent GET_CONFIG is not supported.

Example

```
XA_DRM_DEC_DRM_AUDIO_CONFIG drm_audio_config;
// Fill drm_audio_config structure with appropriate values
res =(*api_func)(api_obj,
      XA_API_CMD_SET_CONFIG_PARAM,
      XA_DRM_DEC_CONFIG_PARAM_DRM_AUDIO_CONFIG,
      (pVOID) & drm_audio_config);
```

Errors

- Common Config and API Errors

Table 3-4 XA_DRM_DEC_CONFIG_PARAM_CONCEAL_FADEIN subcommand

Subcommand	`XA_DRM_DEC_CONFIG_PARAM_CONCEAL_FADEIN`
Description	This command sets the fade-in number of frames. Valid values are from 0 to 15 (default 5). XA_DRM_DEC_CONFIG_PARAM_CONCEAL_FADEIN is defined in xa_drm_dec_api.h

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	
	`i_idx`
	`XA_DRM_DEC_CONFIG_PARAM_CONCEAL_FADEIN`
	
	`pv_value`
	`&fadein_length` – Pointer to integer type variable

Restrictions	This is a WRITE ONLY parameter and the equivalent GET_CONFIG is not supported.

Example

```
int fadein_length = 13;
res =(*api_func)(api_obj,
      XA_API_CMD_SET_CONFIG_PARAM,
      XA_DRM_DEC_CONFIG_PARAM_CONCEAL_FADEIN,
      (pVOID) & fadein_length);
```

Errors

- Common Config and API Errors

Table 3-5 XA_DRM_DEC_CONFIG_PARAM_CONCEAL_FADEOUT subcommand

Subcommand	`XA_DRM_DEC_CONFIG_PARAM_CONCEAL_FADEOUT`
Description	This command sets the fade-out length as number of frames. Valid values are from 0 to 15 (default 6). XA_DRM_DEC_CONFIG_PARAM_CONCEAL_FADEOUT is defined in xa_drm_dec_api.h

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	
	`i_idx`
	`XA_DRM_DEC_CONFIG_PARAM_CONCEAL_FADEOUT`
	
	`pv_value`
	`&fadeout_length` – Pointer to integer type variable

Restrictions	This is a WRITE ONLY parameter and the equivalent GET_CONFIG is not supported.

Example

```
int fadeout_length = 5;
res =(*api_func)(api_obj,
      XA_API_CMD_SET_CONFIG_PARAM,
      XA_DRM_DEC_CONFIG_PARAM_CONCEAL_FADEOUT,
      (pVOID) & fadeout_length);
```

Errors

- Common Config and API Errors

Table 3-6 XA_DRM_DEC_CONFIG_PARAM_CONCEAL_MUTE_RELEASE subcommand

Subcommand	`XA_DRM_DEC_CONFIG_PARAM_CONCEAL_MUTE_RELEASE`
Description	This command sets the 'number of consecutive good frames' required to change the concealment state from mute to fade-in. Valid values are from 0 to 31 (default 3). XA_DRM_DEC_CONFIG_PARAM_CONCEAL_MUTE_RELEASE is defined in xa_drm_dec_api.h

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	
	`i_idx`
	`XA_DRM_DEC_CONFIG_PARAM_CONCEAL_MUTE_RELEASE`
	
	`pv_value`
	`&mute_rel_length` – Pointer to integer type variable

Restrictions	This is a WRITE ONLY parameter and the equivalent GET_CONFIG is not supported.

Example

```
int mute_rel_length = 8;
res =(*api_func)(api_obj,
      XA_API_CMD_SET_CONFIG_PARAM,
      XA_DRM_DEC_CONFIG_PARAM_CONCEAL_MUTE_RELEASE,
      (pVOID) & mute_rel_length);
```

Errors

- Common Config and API Errors

Table 3-7 XA_DRM_DEC_CONFIG_PARAM_CONCEAL_COMFORT_NOISE subcommand

Subcommand	`XA_DRM_DEC_CONFIG_PARAM_CONCEAL_COMFORT_NOISE`
Description	This command sets the level of comfort noise added in the mute state of concealment. Valid values are from 0 to 0x7fffffff (default 0x100000). Noise addition will be of minimum scale for 0, maximum scale for 0x7fffffff. XA_DRM_DEC_CONFIG_PARAM_CONCEAL_COMFORT_NOISE is defined in xa_drm_dec_api.h

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	
	`i_idx`
	`XA_DRM_DEC_CONFIG_PARAM_CONCEAL_COMFORT_NOISE`
	
	`pv_value`
	`&comfort_noise_level` – Pointer to integer type variable

Restrictions	This is a WRITE ONLY parameter and the equivalent GET_CONFIG is not supported.

Example

```
int comfort_noise_level = 0x130000;
res =(*api_func)(api_obj,
      XA_API_CMD_SET_CONFIG_PARAM,
      XA_DRM_DEC_CONFIG_PARAM_CONCEAL_COMFORT_NOISE,
      (pVOID) & comfort_noise_level);
```

Errors

- Common Config and API Errors

Table 3-8 XA_DRM_DEC_CONFIG_PARAM_XHEAAC_HOLDOFF_MODE subcommand

Subcommand	`XA_DRM_DEC_CONFIG_PARAM_XHEAAC_HOLDOFF_MODE`
Description	This command can be used to get silent frames from the decoder during xHE-AAC synchronization at codec initialization. Valid values are 0 to disable or 1 to enable the silence (default 0). XA_DRM_DEC_CONFIG_PARAM_XHEAAC_HOLDOFF_MODE is defined in xa_drm_dec_api.h

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	
	`i_idx`
	`XA_DRM_DEC_CONFIG_PARAM_XHEAAC_HOLDOFF_MODE`
	`pv_value`
	`&holdoff_mode` – Pointer to integer type variable

Restrictions	This is a WRITE ONLY parameter and the equivalent GET_CONFIG is not supported.

Example

```
int holdoff_mode = 1;
res =(*api_func)(api_obj,
      XA_API_CMD_SET_CONFIG_PARAM,
      XA_DRM_DEC_CONFIG_PARAM_XHEAAC_HOLDOFF_MODE,
      (pVOID) & holdoff_mode);
```

Errors

- Common Config and API Errors

3.3.2XA_API_CMD_GET_CONFIG_PARAM

Table 3-9 XA_DRM_DEC_CONFIG_PARAM_SAMPLERATE subcommand

Subcommand	`XA_DRM_DEC_CONFIG_PARAM_SAMPLERATE`
Description	This command gets the sample rate of the PCM data. For xHE-AAC, the value is determined by the application's configuration. For AAC, the value is decided based on the SBR usage and read from the RAW stream.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_GET_CONFIG_PARAM`
	
	`i_idx`
	`XA_DRM_DEC_CONFIG_PARAM_SAMPLERATE`
	
	`pv_value`
	`& sample_rate` – Pointer to the sample rate variable

Restrictions	This value is available only after successful initialization

Example

```
int sample_rate
res =(*api_func)(api_obj,
       XA_API_CMD_GET_CONFIG_PARAM,
       XA_DRM_DEC_CONFIG_PARAM_SAMPLERATE,
       (pVOID) & sample_rate);
```

Errors

- Common Config and API Errors

Table 3-10 XA_DRM_DEC_CONFIG_PARAM_OUT_NUM_CHANNELS subcommand

Subcommand	`XA_DRM_DEC_CONFIG_PARAM_OUT_NUM_CHANNELS`
Description	This command gets the number of channels in the output PCM data. The number of output channels is decided by the output mode. • If the mode is XA_DRM_DEC_OUTPUT_AUTO, out_num_chan will be same as NUM_CHANNELS • If the mode is XA_DRM_DEC_OUTPUT_MONO, out_num_chan will be 1 • If the mode is XA_DRM_DEC_OUTPUT_STEREO, out_num_chan will be 2

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

Restrictions	This value is available only after successful initialization.

Example

```
int out_num_chan;
res =(*api_func)(api_obj,
      XA_API_CMD_GET_CONFIG_PARAM,
      XA_DRM_DEC_CONFIG_PARAM_OUT_NUM_CHANNELS,
      (pVOID) & out_num_chan);
```

Errors

- Common Config and API Errors

Table 3-11 XA_DRM_DEC_CONFIG_PARAM_NUM_CHANNELS subcommand

Subcommand	`XA_DRM_DEC_CONFIG_PARAM_NUM_CHANNELS`
Description	This command gets the number of channels in the input stream.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_GET_CONFIG_PARAM`
	
	`i_idx`
	`XA_DRM_DEC_CONFIG_PARAM_NUM_CHANNELS`
	
	`pv_value`
	`& num_chan` – Pointer to the channel count variable

Restrictions	This value is available only after successful initialization

Example

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

Errors

- Common Config and API Errors

Table 3-12 XA_DRM_DEC_CONFIG_PARAM_PCM_WDSZ subcommand

Subcommand	`XA_DRM_DEC_CONFIG_PARAM_PCM_WDSZ`
Description	This command gets the width of the PCM data.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_GET_CONFIG_PARAM`
	
	`i_idx`
	`XA_DRM_DEC_CONFIG_PARAM_PCM_WDSZ`
	
	`pv_value`
	`& pcm_width` – Pointer to the PCM width variable

Restrictions	For the current implementation, this API always returns 16.

Example

```
int pcm_width
res =(*api_func)(api_obj,
      XA_API_CMD_GET_CONFIG_PARAM,
      XA_DRM_DEC_CONFIG_PARAM_PCM_WDSZ,
      (pVOID) & pcm_width);
```

Errors

- Common Config and API Errors

Table 3-13 XA_DRM_DEC_CONFIG_PARAM_FRAME_SIZE subcommand

Subcommand	`XA_DRM_DEC_CONFIG_PARAM_FRAME_SIZE`
Description	This command gets the frame size of the access unit (or raw AAC/xHE-AAC) frame. For AAC, the value will be 960 /1920 based on SBR usage. For xHE-AAC, the value will be 1024, 2048, or 4096.

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_GET_CONFIG_PARAM`
	
	`i_idx`
	`XA_DRM_DEC_CONFIG_PARAM_FRAME_SIZE`
	
	`pv_value`
	`& frame_size` – Pointer to the frame size variable

Restrictions	This value is available only after successful initialization

Example

```
int frame_size;
res =(*api_func)(api_obj,
       XA_API_CMD_GET_CONFIG_PARAM,
       XA_DRM_DEC_CONFIG_PARAM_FRAME_SIZE,
       (pVOID) & frame_size);
```

Errors

- Common Config and API Errors

Table 3-14 XA_DRM_DEC_CONFIG_PARAM_QMF_MODE subcommand

Subcommand	`XA_DRM_DEC_CONFIG_PARAM_QMF_MODE`
Description	This command gets the QMF mode set by the application. If the application did not set the QMF mode, it returns the default mode. XA_DRM_DEC_QMF_MODE is defined in xa_drm_dec_api.h

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_GET_CONFIG_PARAM`
	
	`i_idx`
	`XA_DRM_DEC_CONFIG_PARAM_QMF_MODE`
	
	`pv_value`
	`& qmf_mode` – Pointer to the QMF mode variable

Restrictions	None

Example

```
XA_DRM_DEC_QMF_MODE qmf_mode;
res =(*api_func)(api_obj,
       XA_API_CMD_GET_CONFIG_PARAM,
       XA_DRM_DEC_CONFIG_PARAM_QMF_MODE,
       (pVOID) & qmf_mode);
```

Errors

- Common Config and API Errors

Table 3-15 XA_DRM_DEC_CONFIG_PARAM_CONCEAL_STATE subcommand

Subcommand	`XA_DRM_DEC_CONFIG_PARAM_CONCEAL_STATE`
Description	This command gets the concealment state set by internal decoder and returns one of the following states ■ XA_DRM_CONCEAL_STATE_NORMAL No concealment or single frame (interpolation) concealment without signal level attenuation. ■ XA_DRM_CONCEAL_STATE_FADE_IN The decoder is in fade-in mode (decoded but attenuated signal). ■ XA_DRM_CONCEAL_STATE_MUTE The decoder is in mute mode (silence or comfort noise). ■ XA_DRM_CONCEAL_STATE_FADE_OUT The decoder is in fade-out mode (concealed and attenuated signal). The enumeration type XA_DRM_DEC_CONCEAL_STATE is defined in xa_drm_dec_api.h The concealment states are explained in Section 3.4

Actual Parameters	`p_xa_module_obj`
	`api_obj` – Pointer to API Structure
	
	`i_cmd`
	`XA_API_CMD_GET_CONFIG_PARAM`
	
	`i_idx`
	`XA_DRM_DEC_CONFIG_PARAM_CONCEAL_STATE`
	
	`pv_value`
	`&conceal_state` – Pointer to the concealment state variable

Restrictions	None

Example

```
XA_DRM_DEC_CONCEAL_STATE conceal_state;
res =(*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_DRM_DEC_CONFIG_PARAM_CONCEAL_STATE,
                  (pVOID) & conceal_state);
```

Errors

- Common Config and API Errors

3.4 Concealment

When more than one frame loss or corruption occurs, the DRM decoder triggers concealment. Section 3.4.1 explains the concealment strategy implemented in the DRM decoder.

The decoder triggers concealment automatically when super frame parsing returns an error. If the application has information on the number of lost or corrupt super frames, the application can feed the same number of all-zero super frames into the input buffer of the decoder as an indication of the loss to the decoder.

In AAC, the error protection can also happen through the error resilient tools in the core decoder at a parameter level; such an error may not trigger frame concealment.

3.4.1 Concealment Strategy

When bad or lost frames are encountered, the decoder starts applying a fade-out slope (in frames). This process of fade-out continues for FADEOUT_FRAMES frames. Once the consecutive bad frame count reaches FADEOUT_FRAMES, the decoder fills the output with low amplitude random noise until good frames are received (called MUTE frames).

If the extent of input audio missed is unknown, in case of xHE-AAC, the decoder automatically fills in concealed MUTE frames to compensate for the lost output time. The elapsed time is derived from a known duration of consumed super frames. The amount of inserted concealed frames is an estimate. However, it is guaranteed that in the long run the margin of error remains constant depending on some wiggle room needed for bit reservoir changes.

When good frames start arriving, the decoder continues filling the output with random noise until the number of good frames reaches MUTE_RELEASE_FRAMES, then the decoder starts the fade-in process. This is to avoid a pumping output signal that can be generated by consecutive sequences of fade-in and fade-out. If the good frames continue, the decoder starts the fade-in slope (in frames) so that it reaches a normal range within FADEIN_FRAMES frames. When the number of good frames reaches FADEIN_FRAMES, the decoder comes out of the concealment process and outputs normally.

Note If the decoder receives bad or lost frames during the fade-in process, the process of fade-out starts again and in this scenario the fade-out process can happen for less than FADEOUT_FRAMES. In case of AAC coding, due to the interpolation concealment method used in the algorithm, single frame losses are considered as good frames when updating the concealment state. This means that single frame losses do not trigger a fade-out process and do not interrupt a fade-in process.

The internal algorithm for xHE-AAC has two paths, one for music audio and another for speech. The concealment procedure for music audio path is the same as that of AAC. The speech path implements the concealment as defined in 3GPP; it may fade out more quickly than the music audio path. This is because the speech coded signals tend to be less stationary, such that long fade-out slopes will cause concealment artifacts.

The default values for FADEOUT_FRAMES and FADEIN_FRAMES are set to 6 and 5, respectively. The default value for MUTE_RELEASE_FRAMES is set to 3. These values can be changed by the application through user configurable concealment parameters (Section 3.3.1). Note, due to the nature of concealment algorithm, the actual fade-in and fade-out applied may vary slightly from what is requested by the application.

There are two methods of concealment: *noise* and *interpolation*. In the *noise* method, a single frame error forces the concealment state machine to an intermediate state and starts fade-out. If a bad frame is followed by a good frame, then the concealment state machine comes out of the intermediate state into normal or error-free mode. However, if user-configured fade-out frames are zero, then the rest of the concealment state machine is also activated and the state passes through appropriate states of mute and fade-in. In the *interpolation* method the concealment state machine starts fading-out only when two consecutive frames are received in error and any erroneous frame between two good frames is interpolated thereby maintaining energy continuity in the time-domain. This results in a *delay* of one frame at the decoded output. The concealment method is chosen automatically based on the input stream type. *Noise* method is chosen instead of *interpolation* method for instances where input stream type is either USAC/RSVD50 or low-delay where additional delay is not recommended.

4. Introduction to the DRM Decoder Application

The provided sample test bench is a test application for the DRM decoding functionalities. It makes calls to API functions in the DRM Decoder library as needed. It contains the following files:

Test bench source files (`test/src`)

- `xa_drm_dec_sample_testbench.cpp`
- `xa_drm_dec_error_handler.cpp`
- `xa_drm_dec_utils.cpp`
- `xa_drm_dec_drm_utils.cpp`
- `xa_waveio.cpp`

Test bench header files (`test/include`)

- `xa_drm_dec_sample_testbench.h`
- `xa_drm_dec_usage.h`
- `xa_drm_dec_drm_utils.h`
- `xa_profiler.h`
- `xa_waveio.h`

The sample application is used to test the DRM Decoder library. This application implements command-line applications with file based input/output. This application can read DRM files (.drm), feed the configuration and super frame to the DRM decoder library, and write the output PCM data into a file. The DRM file format is explained in Section 4.3.

This application can also be used to measure the MCPS and Memory usage of the library.

Note that the command-line options are a subset of the Fraunhofer-provided command-line options [2], [3].

4.1 Making the Executable

To build the application, follow these steps:

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

This will build the example application `xa_drm_dec_test`.

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

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

This will build the DRM Decoder library `xa_drm_dec.a` and copy it to the `lib` directory.

4.2 Usage

The sample application executable can be run with command-line options.

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, as follows:

```
$ xt-run --turbo xa_drm_dec_test -h
```

If no command line arguments are given, the application tries to read the commands from a parameter file named `paramfilesimple.txt`. A sample `paramfilesimple.txt` is provided in the `test/build` folder.

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 The syntax for command lines in the parameter file is the same as the syntax for specifying options on the command line to the test bench program. 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 into two parts on two different lines, use the `@New_line` command.

Example:

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

Note Blank lines will be ignored.

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

4.3 DRM Container Format

As there is no DRM audio format defined within DRM, for file based DRM decoding, the SDC configuration and the audio super frames are packed into a single container using a custom format. In this format, the super frames are preceded by an interface header (which includes a configuration parameter from SDC):

DRM Stream: *[Interface Header] [Audio Super Frame] [IH] [ASF] [IH] [ASF] …*

The interface header is of a variable length that includes the configuration parameter from SDC including codec-specific configurations.

The DRM library accepts the interface header (read into a pre-defined structure) as a configuration and accepts only super frames (as stored in MSC) as input. Parsing the DRM file is implemented in `xa_drm_dec_drm_utils.cpp`.

5. References

[1] *Digital Radio Mondiale (DRM); System Specification. ETSI ES 201 980 V4.1.1 (2014-01)*

[2] *CDK Digital Radio Mondiale (DRM30 / DRM+) Audio Decoder library Interface Description, Revision 2.1.18, July 6, 2017, From Fraunhofer IIS*

[3] *CDK Digital Radio Mondiale (DRM30 / DRM+) Audio Decoder Implementation, From Fraunhofer IIS*

[4] *ISO/IEC 14496 Coding of audio-visual objects – Part 3: Audio*

[5] *ISO/IEC 23003 MPEG Audio technologies – Part 3: Unified speech and audio coding*