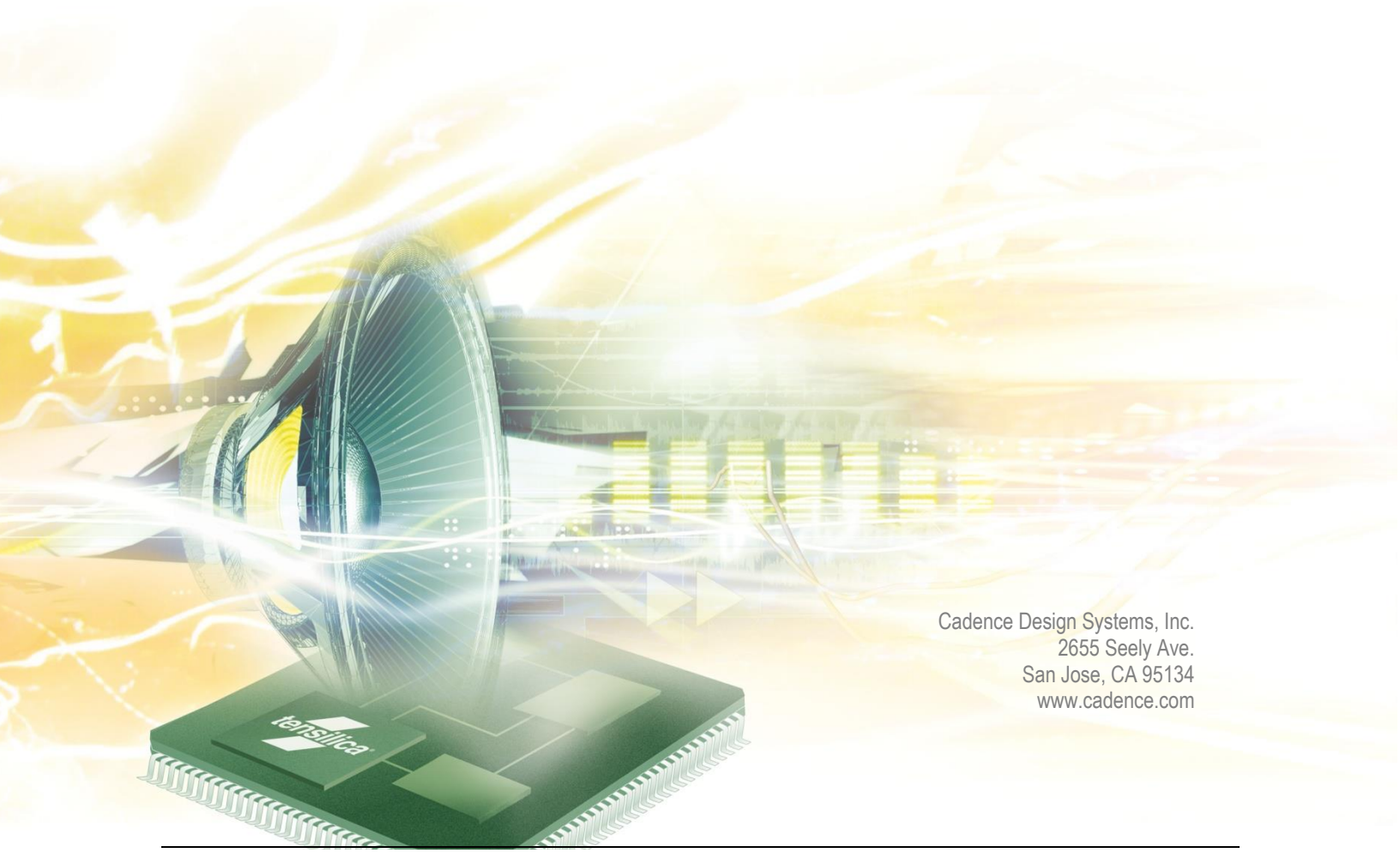
cadence®

AAC Decoder

Programmer's Guide

For HiFi DSPs



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

Version 3.4
August 2017

Contents

1. Introduction to the HiFi AAC Decoder 1
1.1 AAC Description 1
1.2 Document Overview 2
1.3 HiFi AAC Decoder Specifications 2
1.4 HiFi AAC Decoder Performance 4
1.4.1 Memory AAC Stereo 4
1.4.2 Timings AAC Stereo 4
1.4.3 Memory aacPlus V1 Stereo 4
1.4.4 Timings aacPlus V1 Stereo 5
1.4.5 Memory aacPlus V2 Stereo 5
1.4.6 Timings aacPlus V2 Stereo 5
1.4.7 Memory AAC Multi-Channel 5
1.4.8 Timings AAC Multi-Channel 6
1.4.9 Memory aacPlus v1 Multi-Channel 6
1.4.10 Timings aacPlus v1 Multi-Channel 6
1.4.11 Memory aacPlus v2 Multi-Channel 6
1.4.12 Timings aacPlus v2 Multi-Channel 7
2. Generic HiFi Audio Codec API 8
2.1 Memory Management 8
2.2 C Language API 9
2.3 Generic API Errors 11
2.4 Commands 11
2.4.1 Start-up API Stage 13
2.4.2 Set Codec-Specific Parameters Stage 13
2.4.3 Memory Allocation Stage 14
2.4.4 Initialize Codec Stage 15
2.4.5 Get Codec-Specific Parameters Stage 16
2.4.6 Execute Codec Stage 16
2.5 Files Describing the API 17
2.6 HiFi API Command Reference 17
2.6.1 Common API Errors 18
2.6.2 XA_API_CMD_GET_LIB_ID_STRINGS 19
2.6.3 XA_API_CMD_GET_API_SIZE 22
2.6.4 XA_API_CMD_INIT 23
2.6.5 XA_API_CMD_GET_MEMTABS_SIZE 27
2.6.6 XA_API_CMD_SET_MEMTABS_PTR 28
2.6.7 XA_API_CMD_GET_N_MEMTABS 29
2.6.8 XA_API_CMD_GET_MEM_INFO_SIZE 30

2.6.9 XA_API_CMD_GET_MEM_INFO_ALIGNMENT 31
2.6.10 XA_API_CMD_GET_MEM_INFO_TYPE 32
2.6.11 XA_API_CMD_GET_MEM_INFO_PRIORITY 33
2.6.12 XA_API_CMD_SET_MEM_PTR 35
2.6.13 XA_API_CMD_INPUT_OVER 36
2.6.14 XA_API_CMD_SET_INPUT_BYTES 37
2.6.15 XA_API_CMD_GET_CURIDX_INPUT_BUF 38
2.6.16 XA_API_CMD_EXECUTE 39
2.6.17 XA_API_CMD_GET_CONFIG_PARAM 43
2.6.18 XA_API_CMD_SET_CONFIG_PARAM 45
3. HiFi DSP AAC Decoder 46
3.1 Files Specific to the AAC Decoder 47
3.2 AAC Specific Error Codes 48
3.2.1 API Errors 48
3.2.2 Configuration Errors 49
3.2.3 Execute Errors 50
3.3 Configuration Parameters 53
3.3.1 XA_API_CMD_SET_CONFIG_PARAM 55
3.3.2 XA_API_CMD_GET_CONFIG_PARAM 75
4. Introduction to the Example Testbench 95
4.1 Making the Executable 95
4.2 Usage 97
5. References 99

Figures

Figure 1 HiFi Audio Codec Interfaces 8
Figure 2 API Command Sequence Overview 12
Figure 3 Flow Chart for AAC Decoder Integration 46

Tables

Table 2-1 API of AAC Decoder 10
Table 2-2 Commands for Initialization 13
Table 2-3 Commands for Setting Parameters 13
Table 2-4 Commands for Initial Table Allocation 14
Table 2-5 Commands for Memory Allocation 14
Table 2-6 Commands for initialization 15
Table 2-7 Commands for Getting Parameters 16
Table 2-8 Commands for Codec Execution 16
Table 2-9 XA_CMD_TYPE_LIB_NAME subcommand 19
Table 2-10 XA_CMD_TYPE_LIB_VERSION subcommand 20
Table 2-11 XA_CMD_TYPE_API_VERSION subcommand 21
Table 2-12 XA_API_CMD_GET_API_SIZE command 22
Table 2-13 XA_CMD_TYPE_INIT_API_PRE_CONFIG_PARAMS subcommand 23
Table 2-14 XA_CMD_TYPE_INIT_API_POST_CONFIG_PARAMS subcommand 24
Table 2-15 XA_CMD_TYPE_INIT_PROCESS subcommand 25
Table 2-16 XA_CMD_TYPE_INIT_DONE_QUERY subcommand 26
Table 2-17 XA_API_CMD_GET_MEMTABS_SIZE command 27
Table 2-18 XA_API_CMD_SET_MEMTABS_PTR command 28
Table 2-19 XA_API_CMD_GET_N_MEMTABS command 29
Table 2-20 XA_API_CMD_GET_MEM_INFO_SIZE command 30
Table 2-21 XA_API_CMD_GET_MEM_INFO_ALIGNMENT command 31
Table 2-22 XA_API_CMD_GET_MEM_INFO_TYPE command 32
Table 2-23 Memory Type Indices 32
Table 2-24 XA_API_CMD_GET_MEM_INFO_PRIORITY command 33
Table 2-25 Memory Priorities 33
Table 2-26 XA_API_CMD_SET_MEM_PTR command 35

Table 2-27 XA_API_CMD_INPUT_OVER command 36
Table 2-28 XA_API_CMD_SET_INPUT_BYTES command 37
Table 2-29 XA_API_CMD_GET_CURIDX_INPUT_BUF command 38
Table 2-30 XA_CMD_TYPE_DO_EXECUTE subcommand 39
Table 2-31 XA_CMD_TYPE_DONE_QUERY subcommand 40
Table 2-32 XA_CMD_TYPE_DO_RUNTIME_INIT subcommand 41
Table 2-33 XA_API_CMD_GET_OUTPUT_BYTES command 42
Table 2-34 XA_CONFIG_PARAM_CUR_INPUT_STREAM_POS subcommand 43
Table 2-35 XA_CONFIG_PARAM_GEN_INPUT_STREAM_POS subcommand 44
Table 2-36 XA_CONFIG_PARAM_CUR_INPUT_STREAM_POS subcommand 45
Table 3-1 Library Files (in the lib directory): 47
Table 3-2 XA_AACDEC_CONFIG_PARAM_EXTERNALSAMPLINGRATE subcommand 55
Table 3-3 XA_AACDEC_CONFIG_PARAM_EXTERNALBSFORMAT subcommand 56
Table 3-4 XA_AACDEC_CONFIG_PARAM_TO_STEREO subcommand 58
Table 3-5 XA_AACDEC_CONFIG_PARAM_PCM_WDSZ subcommand 59
Table 3-6 XA_AACDEC_CONFIG_PARAM_OUTNCHANS subcommand 60
Table 3-7 XA_AACDEC_CONFIG_PARAM_CHANROUTING subcommand 61
Table 3-8 XA_AACDEC_CONFIG_PARAM_ZERO_UNUSED_CHANS subcommand 63
Table 3-9 XA_AACDEC_CONFIG_PARAM_INPUT_BITOFFSET subcommand 64
Table 3-10 XA_AACDEC_CONFIG_PARAM_BDOWNSAMPLE subcommand 65
Table 3-11 XA_AACDEC_CONFIG_PARAM_BBITSTREAMDOWNMIX subcommand 66
Table 3-12 XA_AACDEC_CONFIG_PARAM_SBR_SIGNALING subcommand 67
Table 3-13 XA_AACDEC_CONFIG_PARAM_ENABLE_APPLY_PRL subcommand 68
Table 3-14 XA_AACDEC_CONFIG_PARAM_TARGET_LEVEL subcommand 69
Table 3-15 XA_AACDEC_CONFIG_PARAM_ENABLE_APPLY_DRC subcommand 70
Table 3-16 XA_AACDEC_CONFIG_PARAM_DRC_COMPRESS_FAC subcommand 71
Table 3-17 XA_AACDEC_CONFIG_PARAM_DRC_BOOST_FAC subcommand 72
Table 3-18 XA_AACDEC_CONFIG_PARAM_ENABLE_FRAME_BY_FRAME_DECODE subcommand 73
Table 3-19 XA_AACDEC_CONFIG_PARAM_OUT_SAMPLERATE subcommand 75
Table 3-20 XA_AACDEC_CONFIG_PARAM_NUM_CHANNELS subcommand 76
Table 3-21 XA_AACDEC_CONFIG_PARAM_PCM_WDSZ subcommand 77
Table 3-22 XA_AACDEC_CONFIG_PARAM_SBR_TYPE subcommand 78
Table 3-23 XA_AACDEC_CONFIG_PARAM_AAC_SAMPLERATE subcommand 79
Table 3-24 XA_AACDEC_CONFIG_PARAM_DATA_RATE subcommand 80
Table 3-25 XA_AACDEC_CONFIG_PARAM_CHANMAP subcommand 81
Table 3-26 XA_AACDEC_CONFIG_PARAM_ACMOD subcommand 82
Table 3-27 XA_AACDEC_CONFIG_PARAM_AAC_FORMAT subcommand 83

Table 3-28 XA_AACDEC_CONFIG_PARAM_OUTNCHANS subcommand 84
Table 3-29 XA_AACDEC_CONFIG_PARAM_DRC_EXT_PRESENT subcommand 85
Table 3-30 XA_AACDEC_CONFIG_PARAM_MPEG_ID subcommand 86
Table 3-31 XA_AACDEC_CONFIG_PARAM_ORIGINAL_OR_COPY subcommand 87
Table 3-32 XA_AACDEC_CONFIG_PARAM_COPYRIGHT_ID_PTR subcommand 88
Table 3-33 XA_AACDEC_CONFIG_PARAM_PCE_STATUS subcommand 89
Table 3-34 XA_AACDEC_CONFIG_PARAM_DWNMIX_METADATA subcommand 90
Table 3-35 XA_AACDEC_CONFIG_PARAM_DWNMIX_LEVEL_DVB subcommand 91
Table 3-36 XA_AACDEC_CONFIG_PARAM_PARSED_DRC_INFO subcommand 92
Table 3-37 XA_AACDEC_CONFIG_PARAM_PROG_REF_LEVEL subcommand 93
Table 3-38 XA_AACDEC_CONFIG_PARAM_ENABLE_FRAME_BY_FRAME_DECODE subcommand 94

Document Change History

Version	Changes
3.0	■ History section added. ■ Updated Section1 for LATM support and added memory and timing performance data for HiFi Mini and HiFi 3. ■ Revised Section 3.
3.1	■ Clarified the supported decoding formats in Section 1.3. ■ Added new API command `XA_AACDEC_CONFIG_PARAM_ENABLE_FRAME_BY_FRAME_DECODE`.
3.2	■ Added performance data for HiFi 4 in Section 1.4.
3.3	■ Added LOAS/LATM support variants for all AAC Decoder versions.
3.4	■ Added performance data for HiFi 3z in Section 1.4.

1. Introduction to the HiFi AAC Decoder

The HiFi DSP AAC Decoder is a general term for three different levels of AAC technology supplied by Cadence Tensilica as separate libraries:

- The basic library implements the AAC specification recommended by the Moving Picture Experts Group (MPEG), a working group of ISO/IEC.
- The second library, aacPlus v1 (HE-AAC), implements the same AAC functionality with the addition of the SBR (Spectral Band Replication) tool from Coding Technologies.
- The third library, aacPlus v2 (HE-AAC v2), implements everything above and additionally the PS (Parametric Stereo) tool from Coding Technologies.

Stereo (as well as 8-channel) versions of each of these three libraries are available from Cadence Tensilica. The stereo version has smaller data memory requirements. For both the stereo and 8-channel versions of these three libraries, a LOAS/LATM supported variant is also available.

Note that all 8-channel versions of these three libraries support extraction of DRC metadata, and 8-channel HE-AAC-V2 LOAS/LATM supported library supports extraction as well as application of DRC metadata to the output audio.

The rest of this document refers to the HiFi DSP AAC Decoder set of libraries simply as the HiFi AAC Decoder.

The vendor source version is Coding Technologies' aacPlus Fixed Point Firmware Reference Decoder v4.0.3.

1.1 AAC Description

AAC (Advanced Audio Coding) was created as part of the MPEG-2 specification ISO/IEC 13818-7. [1]. It was originally referred to as "non-backward compatible". This was referring to the move away from compatibility with the existing audio specification (MP3). AAC was then specified and provided with more tools in the MPEG-4 specification ISO/IEC 14496-3. [2]

Following is a quote from Coding Technologies with regard to the aacPlus v1 and v2 enhancements:

"MPEG-4 aacPlus is the combination of three MPEG technologies comprising Advanced Audio Coding (AAC), coupled with Coding Technologies' Spectral Band Replication (SBR), and Parametric Stereo (PS) technologies. SBR is a unique bandwidth extension technique, which enables audio codecs to deliver the same quality at half the bit rate. PS significantly increases the codec efficiency a second time for low bit rate stereo signals." [3] [4]

The target of 1 bit per sample for AAC results in a bitstream of 96kbits/s for stereo 48kHz audio. This is a compression of 16 times over the uncompressed PCM Audio. With the additional technologies from Coding Technology, this compression increases dramatically.

The MPEG-2 and MPEG-4 specifications mentioned above also extended the AAC audio coding technology to support encoding and decoding of multi-channel audio streams. Later, the plain AAC was also combined with Coding Technologies' Spectral Band Replication (SBR) technique for encoding multi-channel streams to create multi-channel audio content at very low bit rates of up to 128 kbits/s, while the Parametric Stereo (PS) technology is restricted to the stereo content only.

1.2 Document Overview

This document covers all the information required to integrate the HiFi Audio Codecs into an application. The HiFi codec libraries implement a simple API to encapsulate the complexities of the coding operations and simplify the application and system implementation. Parts of the API are common to all the HiFi codecs; these are described after the introduction. The next section covers all the features and information particular to the HiFi AAC Decoder. Finally, the example testbenches are described.

1.3 HiFi AAC Decoder Specifications

The HiFi DSP AAC Decoder from Cadence Tensilica implements the following features:

- Cadence Audio Codec API is used.
- An ISO/IEC 14496 (MPEG-4) compliant decoder supporting the AAC profile (level 5), the HE-AAC profile (level 5), and the HE-AAC v2 profile (level 5), which means the AAC-LC, SBR, and PS object types for up to eight channels are supported
- An ISO/IEC 13818 (MPEG-2) compliant AAC LC decoder supporting SBR
- A 3GPP TS 26.410 (enhanced aacPlus) compliant decoder
- The library supports parsing of the following standard formats along with support for decoding raw payloads as follows:
  - The ADTS (Audio Data Transport Stream) packets with one raw data block per ADTS frame.
  - The ADIF (Audio Data Interchange Format) with one Program Config Element (PCE).
- The LATM (Low Overhead Audio Transport Multiplex) with and without LOAS (Low Overhead Audio Stream). In LOAS, out of three types, only AudioSyncStream() is supported with the restriction of single program, single layer with no subframes. This feature is only available for the LOAS supported libraries, see below.
- The library supports the following sample rates: 8000, 11025, 12000, 16000, 22050, 24000, 32000, 44100, 48000, 64000, 88200 and 96000, in Hz.
- Per MPEG specification, the maximum supported bit rate depends on both the sample rate and the number of channels. At a 48000 Hz sample rate, a 2-channel stream can reach 576 kbps, and an 8-channel stream can reach 2304 kbps. The HiFi AAC Decoder is MPEG-compliant.

There are twelve libraries available for the AAC Decoder.

The following six libraries represent three levels of AAC profile support with a stereo-only and an 8-channel version for each level. These libraries support ADTS, ADIF, and raw formats:

- The AAC Decoder library supports the AAC LC profile
- The aacPlus v1 Decoder library supports the HE-AAC profile, along with capabilities of the AAC Decoder library
- The aacPlus v2 Decoder library additionally supports the HE-AAC v2 profile, along with the capabilities of the aacPlus v1 Decoder

The additional six libraries represent three levels of AAC profile support mentioned above with a stereo-only and an 8-channel version for each level with LOAS/LATM support. These libraries support ADTS, ADIF, raw, LOAS, and LATM formats.

Note The following library from above twelve variants is provided as a full featured package; it is a superset of the rest of the packages listed above.

- The aacPlus v2 multi-channel with LOAS library. Offered only in 8-channel package, it supports all features in the aacPlus v2 Decoder library (8-channel), along with LOAS format support and DRC metadata application.

The libraries provide the following user options:

- Arbitrary routing of input channels to interleaved output channels
- Support for decoding of up to 7.1-channel encoded streams (8-channel libraries only)
- Implicit and explicit SBR signaling (aacPlus libraries only)
- Optional downsampling of SBR output (aacPlus libraries only)
- Optional mono downmix for SBR processing (aacPlus libraries only)
- Partial data feeding is allowed in bitstream formats ADTS and LOAS, which can be enabled by means of a compile time switch SMALL_INPUT_CHUNK. Note however, that this is not supported in case of ADIF, raw, or LATM formats.

This decoder implementation has been certified by Coding Technologies. The implementation is based on [5].

1.4 HiFi AAC Decoder Performance

The HiFi DSP AAC Decoders from Tensilica were 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 ranges from 160 to 210 bytes depending on which particular library is used.
- The memory table structure size returned by XA_API_CMD_GET_MEMTABS_SIZE is approximately 150 bytes.

1.4.1 Memory AAC Stereo

LOAS Support	Text (Kbytes)					Data (Kbytes)	Runtime Memory (Kbytes)				
	HiFi Mini	HiFi 2	HiFi 3	HIFi 3z	HiFi 4		Persistent	Scratch	Stack	Input	Output
No	33.2	36.5	37.4	39.3	41.1	23.8	5.4	12.2	0.9	1.8	8.0
Yes	36.5	39.7	40.7	42.9	44.8	23.8	6.2	12.2	1.2	8.3	8.0

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

1.4.2 Timings AAC Stereo

Rate kHz	Channels	Bit Rate kbps	Average CPU Load (MHz)				
			HiFi Mini	HiFi 2	HiFi 3	HiFi 3z	HiFi 4
44.1	2	96	7.2	7.1	5.6	4.9	4.9
48	2	128	8.1	8.0	6.3	5.5	5.5
48	2	320	9.4	9.3	7.4	6.5	6.5

1.4.3 Memory aacPlus V1 Stereo

LOAS Support	Text (Kbytes)					Data (Kbytes)	Runtime Memory (Kbytes)				
	HiFi Mini	HiFi 2	HiFi 3	HiFi 3z	HiFi 4		Persistent	Scratch	Stack	Input	Output
No	59.3	65.7	66.3	70.6	73.5	34.6	24.3	22.2	0.9	1.8	16.0
Yes	62.6	69.0	69.7	74.2	77.1	34.6	25.0	22.2	1.7	8.3	16.0

Note In this case, the output buffer requirement is the same for both 16- and 24-bit PCM data.

1.4.4 Timings aacPlus V1 Stereo

Rate kHz	Channels	Bit Rate kbps	Average CPU Load (MHz)				
			HiFi Mini	HiFi 2	HiFi 3	HiFi 3z	HiFi 4
44.1	2	64	18.9	18.8	16.0	14.3	13.2
48	2	128	20.8	20.7	17.6	15.7	14.5

1.4.5 Memory aacPlus V2 Stereo

LOAS Support	Text (Kbytes)					Data (Kbytes)	Runtime Memory (Kbytes)				
	HiFi Mini	HiFi 2	HiFi 3	HiFi 3z	HiFi 4		Persistent	Scratch	Stack	Input	Output
No	66.2	73.8	74.0	78.7	81.8	36.5	28.8	22.2	0.9	1.8	16.0
Yes	69.4	77.1	77.4	82.3	85.5	36.5	29.6	22.2	1.8	8.3	16.0

Note In this case, the output buffer requirement is the same for both 16- and 24-bit PCM data.

1.4.6 Timings aacPlus V2 Stereo

Rate kHz	Channels	Bit Rate kbps	Average CPU Load (MHz)				
			HiFi Mini	HiFi 2	HiFi 3	HiFi 3z	HiFi 4
44.1	2	48	19.9	19.5	17.1	15.3	14.1
48	2	64	21.3	20.9	18.2	16.3	15.0

1.4.7 Memory AAC Multi-Channel

LOAS Support	nch	Text (Kbytes)					Data (Kbytes)	Runtime Memory (Kbytes)				
		HiFi Mini	HiFi 2	HiFi 3	HiFi 3z	HiFi 4		Persistent	Scratch	Stack	Input	Output
No	6	35.7	38.9	39.8	41.9	43.8	23.8	14.9	35.4	0.9	4.8	24.0
	8	35.7	38.9	39.8	41.9	43.8	23.8	19.0	47.2	0.9	6.3	32.0
Yes	6	38.9	42.1	43.2	45.5	47.5	23.8	15.7	35.4	1.3	8.3	24.0
	8	38.9	42.1	43.2	45.5	47.5	23.8	19.8	47.2	1.3	8.3	32.0

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

1.4.8 Timings AAC Multi-Channel

Rate kHz	Channels	Bit Rate kbps	Average CPU Load (MHz)				
			HiFi Mini	HiFi 2	HiFi 3	HiFi 3z	HiFi 4
48	6	384	18.3	18.0	13.2	11.8	11.5
48	8	576	31.3	30.9	23.4	20.8	20.9

1.4.9 Memory aacPlus v1 Multi-Channel

LOAS Support	nch	Text (Kbytes)					Data (Kbytes)	Runtime Memory (Kbytes)				
		HiFi Mini	HiFi 2	HiFi 3	HiFi 3z	HiFi 4		Persistent	Scratch	Stack	Input	Output
No	6	61.7	68.1	68.8	73.2	76.2	34.6	71.4	35.4	0.9	4.8	48.0
	8	61.7	68.1	68.8	73.2	76.2	34.6	94.4	47.2	0.9	6.3	64.0
Yes	6	65.0	71.4	72.2	76.8	79.8	34.6	72.2	35.4	1.3	8.3	48.0
	8	65.0	71.4	72.2	76.8	79.8	34.6	95.2	47.2	1.3	8.3	64.0

Note In this case, the output buffer requirement is the same for both 16- and 24-bit PCM data.

1.4.10 Timings aacPlus v1 Multi-Channel

Rate kHz	Channels	Bit Rate kbps	Average CPU Load (MHz)				
			HiFi Mini	HiFi 2	HiFi 3	HiFi 3z	HiFi 4
48	6	128	53.2	53.0	44.0	39.8	36.2
48	8	192	71.0	70.7	58.8	53.4	48.6

1.4.11 Memory aacPlus v2 Multi-Channel

LOAS Support	nch	Text (Kbytes)					Data (Kbytes)	Runtime Memory (Kbytes)				
		HiFi Mini	HiFi 2	HiFi 3	HiFi 3z	HiFi 4		Persistent	Scratch	Stack	Input	Output
No	6	68.7	76.3	76.6	81.5	84.6	36.5	73.8	35.4	0.9	4.8	48.0
	8	68.7	76.3	76.6	81.5	84.6	36.5	96.8	47.2	0.9	6.3	64.0
Yes *	6	75.7	83.7	84.1	89.4	92.8	36.6	75.8	35.4	1.3	8.3	48.0
	8	75.7	83.7	84.1	89.4	92.8	36.6	98.8	47.2	1.3	8.3	64.0

Note In this case, the output buffer requirement is the same for both 16- and 24-bit PCM data.

Note *This library also supports the application of DRC metadata embedded in the stream.

1.4.12 Timings aacPlus v2 Multi-Channel

The timing specifications of the aacPlus v2 multi-channel library are identical to the timing specifications of the aacPlus v1 multi-channel library for AAC and aacPlus v1 streams, and to the timing specifications of the aacPlus v2 stereo library for aacPlus v2 streams.

Note: Performance specification measurements are carried out 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 3/HiFi 3z/HiFi 4/HiFi Mini are obtained by running the test that is recompiled from the HiFi 2 source code in the HiFi 3/HiFi 3z/HiFi 4/HiFi Mini configuration. No specific optimization is performed for HiFi 3/HiFi 3z/HiFi 4/HiFi Mini except for AAC Stereo (aac_dec). For AAC Stereo library, specific optimization is performed for HiFi 3, and the MCPS numbers for /HiFi 3z/HiFi 4 are obtained by running the test that is recompiled from the HiFi 3 source code in the HiFi 3z/HiFi 4 configuration.

Note: The increase in Average CPU Load (MHz) for LOAS/LATM processing is negligible (<0.1 MHz).

2. Generic HiFi Audio Codec API

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

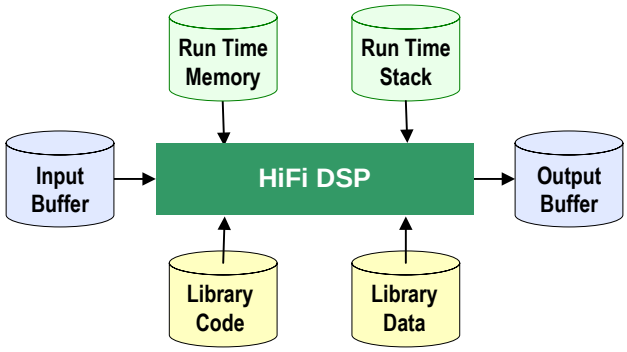


Figure 1 HiFi Audio Codec Interfaces

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

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

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

2.1 Memory Management

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

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

API Object

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

API Memory Table

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

Persistent Memory

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

Scratch Memory

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

Input Buffer

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

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

Output Buffer

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

2.2 C Language API

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

Table 2-1 API of AAC Decoder

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

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

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

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

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

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

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

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

2.3 Generic API Errors

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

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

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

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

2.4 Commands

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

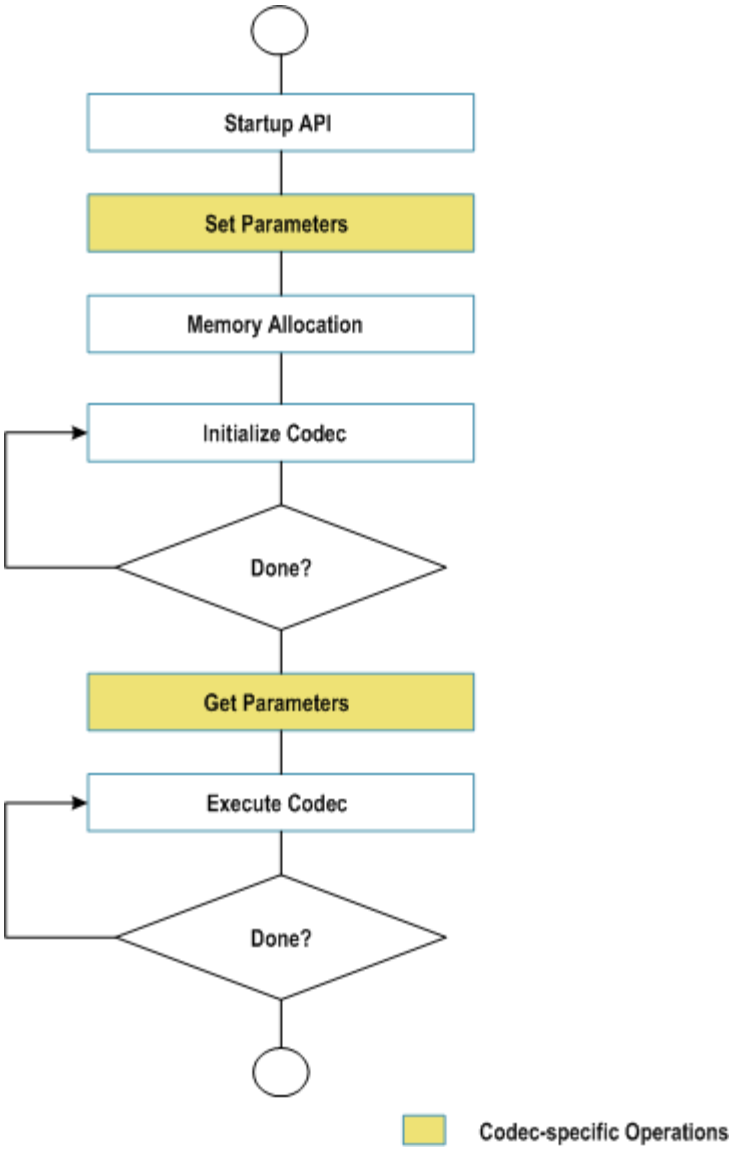


Figure 2 API Command Sequence Overview

2.4.1 Start-up API Stage

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

Table 2-2 Commands for Initialization

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

2.4.2 Set Codec-Specific Parameters Stage

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

Table 2-3 Commands for Setting Parameters

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

2.4.3 Memory Allocation Stage

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

Table 2-4 Commands for Initial Table Allocation

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

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

Table 2-5 Commands for Memory Allocation

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

2.4.4 Initialize Codec Stage

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

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

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

Table 2-6 Commands for initialization

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

2.4.5 Get Codec-Specific Parameters Stage

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

Table 2-7 Commands for Getting Parameters

Command / Subcommand	Description
XA_API_CMD_GET_CONFIG_PARAM XA_<*codec*>_CONFIG_PARAM_<*param_name*>	Get the value of the parameter from the codec. See the codec-specific section for parameter definitions.

2.4.6 Execute Codec Stage

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

Table 2-8 Commands for Codec Execution

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

2.5 Files Describing the API

The common include files (`include`)

- `xa_apicmd_standards.h`
  The command definitions for the generic API calls
- `xa_error_standards.h`
  The macros and definitions for all the generic errors
- `xa_memory_standards.h`
  The definitions for memory block allocation
- `xa_type_def.h`
  All the types required for the API calls

2.6 HiFi API Command Reference

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

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

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

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

2.6.1 Common API Errors

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

- XA_API_FATAL_MEM_ALLOC

  `p_xa_module_obj` is NULL

- XA_API_FATAL_MEM_ALIGN

  `p_xa_module_obj` is not aligned to 4 bytes

- XA_API_FATAL_INVALID_CMD

  `i_cmd` is not a valid command

- XA_API_FATAL_INVALID_CMD_TYPE

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

2.6.2 XA_API_CMD_GET_LIB_ID_STRINGS

Table 2-9 XA_CMD_TYPE_LIB_NAME subcommand

Subcommand	XA_CMD_TYPE_LIB_NAME
Description	This command obtains the name of the library in the form of a string. The maximum length of the string that the library will provide is 30 bytes. Therefore the application shall pass a pointer to a buffer of a minimum size of 30 bytes. This command is optional.
Actual Parameters	p_xa_module_obj **NULL** i_cmd XA_API_CMD_GET_LIB_ID_STRINGS i_idx XA_CMD_TYPE_LIB_NAME pv_value process_name – Pointer to a character buffer in which the name of the library is returned.
Restrictions	None

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

Example

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

Errors

- XA_API_FATAL_MEM_ALLOC

  This error is suppressed as p_xa_module_obj is NULL.

- XA_API_FATAL_MEM_ALLOC

  pv_value is NULL.

Table 2-10 XA_CMD_TYPE_LIB_VERSION subcommand

Subcommand	XA_CMD_TYPE_LIB_VERSION
Description	This command obtains the version of the library in the form of a string. The maximum length of the string that the library will provide is 30 bytes. Therefore the application shall pass a pointer to a buffer of a minimum size of 30 bytes. This command is optional.
Actual Parameters	p_xa_module_obj **NULL** i_cmd XA_API_CMD_GET_LIB_ID_STRINGS i_idx XA_CMD_TYPE_LIB_VERSION pv_value lib_version – Pointer to a character buffer in which the version of the library is returned
Restrictions	None

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

Example

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

Errors

- XA_API_FATAL_MEM_ALLOC

  This error is suppressed as p_xa_module_obj is NULL.

- XA_API_FATAL_MEM_ALLOC

  pv_value is NULL.

Table 2-11 XA_CMD_TYPE_API_VERSION subcommand

Subcommand	XA_CMD_TYPE_API_VERSION
Description	This command obtains the version of the API in the form of a string. The maximum length of the string that the library will provide is 30 bytes. Therefore the application shall pass a pointer to a buffer of a minimum size of 30 bytes. This command is optional.
Actual Parameters	p_xa_module_obj **NULL** i_cmd XA_API_CMD_GET_LIB_ID_STRINGS i_idx XA_CMD_TYPE_API_VERSION pv_value api_version – Pointer to a character buffer in which the version of the API is returned.
Restrictions	None

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

Example

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

Errors

- XA_API_FATAL_MEM_ALLOC

  This error is suppressed as p_xa_module_obj is NULL.

- XA_API_FATAL_MEM_ALLOC

  pv_value is NULL

2.6.3 XA_API_CMD_GET_API_SIZE

Table 2-12 XA_API_CMD_GET_API_SIZE command

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

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

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

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

Example

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

Errors

- XA_API_FATAL_MEM_ALLOC

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

- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.4 XA_API_CMD_INIT

Table 2-13 XA_CMD_TYPE_INIT_API_PRE_CONFIG_PARAMS subcommand

Subcommand	XA_CMD_TYPE_INIT_API_PRE_CONFIG_PARAMS
Description	This command is used to set the default value of the configuration parameters. The configuration parameters can then be altered by using one of the codec-specific parameter setting commands. Refer to the codec-specific section.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_INIT i_idx XA_CMD_TYPE_INIT_API_PRE_CONFIG_PARAMS pv_value **NULL**
Restrictions	None

Example

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

Errors

- Common API Errors

Table 2-14 XA_CMD_TYPE_INIT_API_POST_CONFIG_PARAMS subcommand

Subcommand	XA_CMD_TYPE_INIT_API_POST_CONFIG_PARAMS
Description	This command is used to calculate the sizes of all the memory blocks required by the application. It should occur after the codec-specific parameters have been set.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_INIT i_idx XA_CMD_TYPE_INIT_API_POST_CONFIG_PARAMS pv_value **NULL**
Restrictions	None

Example

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

Errors

- Common API Errors

Table 2-15 XA_CMD_TYPE_INIT_PROCESS subcommand

Subcommand	XA_CMD_TYPE_INIT_PROCESS
Description	This command initializes the codec. In the case of a decoder, it searches for the valid header and performs the header decoding to get the encoded stream parameters. This command is part of the initialization loop. It must be repeatedly called until the codec signals it has finished. In the case of an encoder, the initialization of codec is performed. No output data is created during initialization.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_INIT i_idx XA_CMD_TYPE_INIT_PROCESS pv_value **NULL**
Restrictions	None

Example

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

Errors

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

Table 2-16 XA_CMD_TYPE_INIT_DONE_QUERY subcommand

Subcommand	XA_CMD_TYPE_INIT_DONE_QUERY
Description	This command checks to see if the initialization process has completed. If it has, the flag value is set to 1 else it is set to zero. A pointer to the flag variable is passed as an argument.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_INIT i_idx XA_CMD_TYPE_INIT_DONE_QUERY pv_value &init_done – Pointer to flag that indicates the completion of initialization process.
Restrictions	None

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  pv_value is NULL

2.6.5 XA_API_CMD_GET_MEMTABS_SIZE

Table 2-17 XA_API_CMD_GET_MEMTABS_SIZE command

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

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

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

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.6 XA_API_CMD_SET_MEMTABS_PTR

Table 2-18 XA_API_CMD_SET_MEMTABS_PTR command

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

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

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

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_API_FATAL_MEM_ALIGN

  `pv_value` is not aligned to 4 bytes

2.6.7 XA_API_CMD_GET_N_MEMTABS

Table 2-19 XA_API_CMD_GET_N_MEMTABS command

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

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

Restrictions	None

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.8 XA_API_CMD_GET_MEM_INFO_SIZE

Table 2-20 XA_API_CMD_GET_MEM_INFO_SIZE command

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

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

Restrictions	None

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_API_FATAL_INVALID_CMD_TYPE

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

2.6.9 XA_API_CMD_GET_MEM_INFO_ALIGNMENT

Table 2-21 XA_API_CMD_GET_MEM_INFO_ALIGNMENT command

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

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

Restrictions	None

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_API_FATAL_INVALID_CMD_TYPE

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

2.6.10 XA_API_CMD_GET_MEM_INFO_TYPE

Table 2-22 XA_API_CMD_GET_MEM_INFO_TYPE command

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

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

Restrictions	None

Example

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

Table 2-23 Memory Type Indices

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_API_FATAL_INVALID_CMD_TYPE

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

2.6.11 XA_API_CMD_GET_MEM_INFO_PRIORITY

Table 2-24 XA_API_CMD_GET_MEM_INFO_PRIORITY command

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

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

Restrictions	None

Example

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

Table 2-25 Memory Priorities

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_API_FATAL_INVALID_CMD_TYPE

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

2.6.12 XA_API_CMD_SET_MEM_PTR

Table 2-26 XA_API_CMD_SET_MEM_PTR command

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

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

Restrictions	The pointer must be correctly aligned to the requirements

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_API_FATAL_INVALID_CMD_TYPE

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

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

2.6.13 XA_API_CMD_INPUT_OVER

Table 2-27 XA_API_CMD_INPUT_OVER command

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

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

Restrictions	None

Example

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

Errors

- Common API Errors

2.6.14 XA_API_CMD_SET_INPUT_BYTES

Table 2-28 XA_API_CMD_SET_INPUT_BYTES command

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

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

Restrictions	None

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.15 XA_API_CMD_GET_CURIDX_INPUT_BUF

Table 2-29 XA_API_CMD_GET_CURIDX_INPUT_BUF command

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

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

Restrictions	None

Example

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

Errors

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

2.6.16 XA_API_CMD_EXECUTE

Table 2-30 XA_CMD_TYPE_DO_EXECUTE subcommand

Subcommand	XA_CMD_TYPE_DO_EXECUTE
Description	This command executes the codec.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_EXECUTE i_idx XA_CMD_TYPE_DO_EXECUTE pv_value **NULL**
Restrictions	None

Example

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

Errors

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

Table 2-31 XA_CMD_TYPE_DONE_QUERY subcommand

Subcommand	XA_CMD_TYPE_DONE_QUERY
Description	This command checks to see if the end of processing has been reached. If it is, the flag value is set to 1; else it is set to zero. The pointer to the flag is passed as an argument. Processing by the codec can continue for several invocations of the DO_EXECUTE command after the last input data has been passed to the codec, so the application should not assume that the codec has finished generating all its output until so indicated by this command.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_EXECUTE i_idx XA_CMD_TYPE_DONE_QUERY pv_value &flag – Pointer to the flag variable
Restrictions	None

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  pv_value is NULL

Table 2-32 XA_CMD_TYPE_DO_RUNTIME_INIT subcommand

Subcommand	XA_CMD_TYPE_DO_RUNTIME_INIT
Description	This command resets the decoder's history buffers. It can be used to avoid distortions and clicks by facilitating playback ramping up and down during trick-play. 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

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`

2.6.17 XA_API_CMD_GET_CONFIG_PARAM

Table 2-34 XA_CONFIG_PARAM_CUR_INPUT_STREAM_POS subcommand

Subcommand	`XA_CONFIG_PARAM_CUR_INPUT_STREAM_POS`

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

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

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

Example

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

Errors

- Common API Errors

Table 2-35 XA_CONFIG_PARAM_GEN_INPUT_STREAM_POS subcommand

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

Example

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

Errors

- Common API Errors

2.6.18 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 AAC Decoder

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

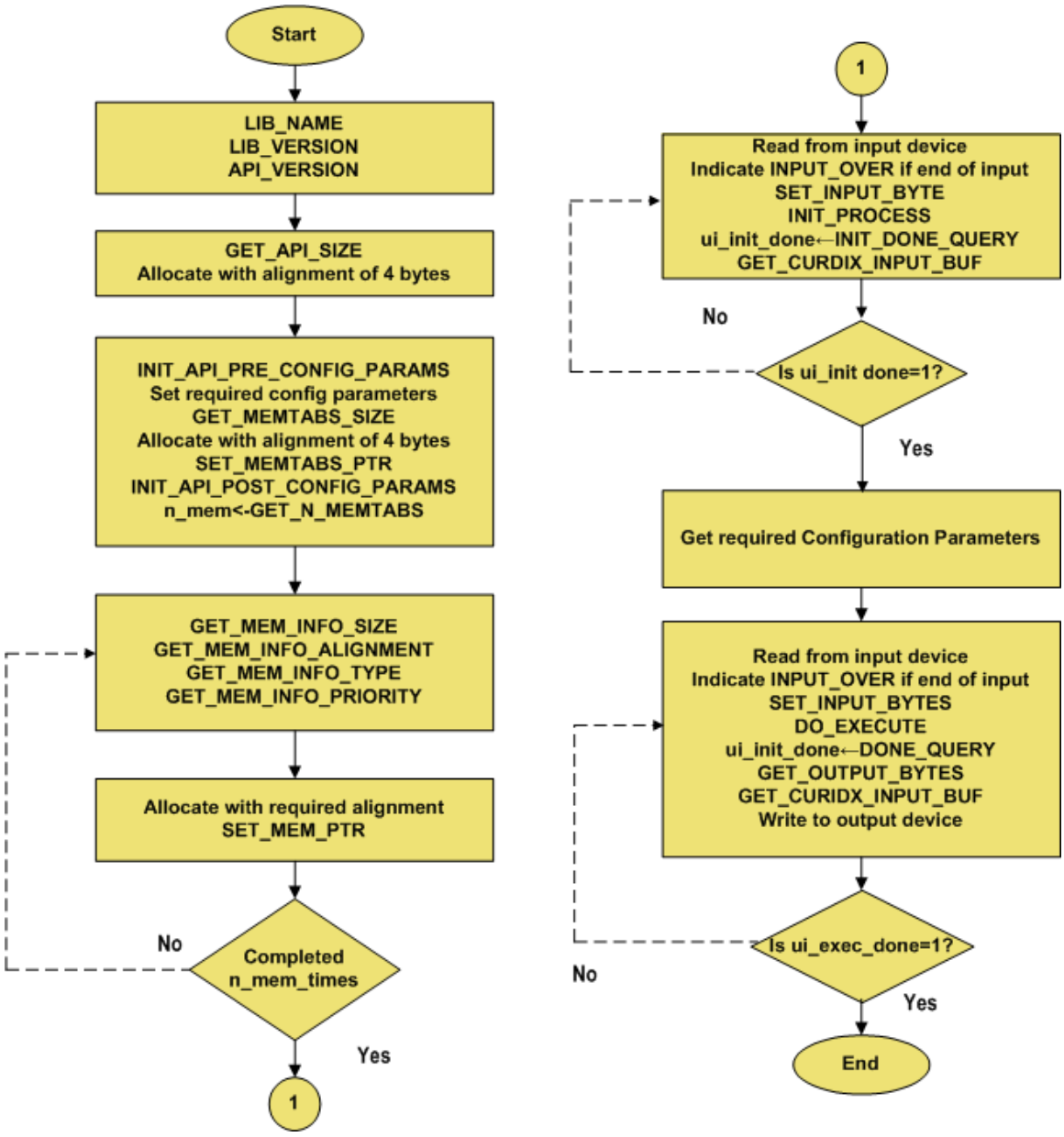


Figure 3 Flow Chart for AAC Decoder Integration

3.1 Files Specific to the AAC Decoder

Each AAC Decoder package contains an API header file (.h) and one of the seven libraries (.a). The name and description of each file is listed below. In this document, each file may be referred to with a shorter name as mentioned in the table.

AAC Decoder API header file (in the include/aac_dec directory):

- xa_aac_dec_api.h

Table 3-1 Library Files (in the lib directory):

Library Name	Description	Referred to as:
xa_aac_dec.a	stereo plain AAC	aac
xa_aacplus_dec.a	stereo aacPlus v1	aacplus
xa_aacplus_v2_dec.a	stereo aacPlus v2	aacplus_v2
xa_aacmch_dec.a	multi-channel plain AAC	aacmch
xa_aacmchplus_dec.a	multi-channel aacPlus v1	aacmchplus
xa_aacmchplus_v2_dec.a	multi-channel aacPlus v2	aacmchplus_v2
xa_aac_loas_dec.a	stereo plain AAC with LOAS/LATM support	aac_loas
xa_aacplus_loas_dec.a	stereo aacPlus v1 with LOAS/LATM support	aacplus_loas
xa_aacplus_v2_loas_dec.a	stereo aacPlus v2 with LOAS/LATM support	aacplus_v2_loas
xa_aacmch_loas_dec.a	multi-channel plain AAC with LOAS/LATM support	aacmch_loas
xa_aacmchplus_loas_dec.a	multi-channel aacPlus v1 with LOAS/LATM support	aacmchplus_loas
xa_aacmchplus_v2_loas_dec.a	multi-channel aacPlus v2 with LOAS/LATM support and DRC Processing	aacmchplus_v2_loas

Note aacmch, aacmchplus, aacmchplus_v2, aacmch_loas, aacmchplus_loas, and aacmchplus_v2_loas libraries are collectively called “Multi-Channel libraries”

Note aac, aacplus, aacplus_v2, aac_loas, aacplus_loas, and aacplus_v2_loas libraries are grouped as “Stereo libraries”

The AAC decoder API call is defined as:

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

3.2 AAC Specific Error Codes

Other than common error codes explained in Chapter 2, AAC decoder APIs may also report error codes specific to AAC decoder libraries. 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 with any subcommand.
- Transport layer:
  The layer of decoder that parses the transport header (ADTS, LOAS/LATM) or storage header (ADIF) and extracts the raw_data_block() (Refer to subpart 4 of **[**2**])** from the input data.
- Raw Decoder:
  The layer of the decoder which parses raw_data_block() and decodes it and produces PCM output.

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.

For example:

- INIT/EXEC API cannot be called before allocating required memories
- EXEC API cannot be called before successful call of INIT API
- Specific Config parameters can be obtained only after successful INIT
- Specific Config parameters may not be supported for the bitstream format under decoding (e.g., MPEGID is not present in LOAS/LATM streams)

The API errors specific to AAC decoder libraries are explained below.

- XA_AACDEC_API_NONFATAL_INVALID_API_SEQ

  Description: This error is reported in case of invalid API sequence or unexpected API call. Decoder ignores the call.

  Required or suggested actions: Application should not use the return parameters. The application programmer may also consider modifying the code to avoid this error. Refer to Figure 3 for the correct API sequence and refer to Section 3.3 for accepted config API calls.

- XA_AACDEC_API_FATAL_INVALID_API_SEQ

  Description: This error is reported in case of an invalid API sequence or unexpected API call and the decoder cannot proceed further.

  Required or suggested actions: Application code shall be modified. Refer to Figure 3 for the correct API sequence and refer to Section 3.3 for accepted config API calls.

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. 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 2.6.1 and API errors described in Section 3.2.1.

Following are the common errors 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: Application shall not use the returned parameter.

- 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 an API error. It is only reported by config APIs.

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

Configuration errors that are unique to specific configuration subcommands are explained in sections 3.3.1 and 3.3.2, along with the configuration subcommands.

3.2.3 Execute Errors

Execute errors are errors that occur during the initialization or execution process. Typically, these errors are caused by (but not limited to) the following:

- Invalid or missing configuration parameters
- Wrong input and output buffer settings
- Stream parsing errors

The following execute errors are specific to the AAC 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 needs to 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 for SBR streams. The output ramp down of SBR decoding extends to two frames and this status code indicates that the ramp down has not finished yet

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

- 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. Both of these two errors can occur during the INIT or EXEC process. They should be addressed differently.

  Required or suggested actions:
  - Case1: FATAL error during INIT. The decoder cannot initialize the raw decoder due to invalid sampling frequency or memory corruption (detected as NULL pointers). The application should revalidate the sampling frequency and other memory allocations, and retry.
  - Case 2: FATAL error during EXEC. The application should stop the decoding process and feed another stream.
  - Case 3: NONFATAL error during INIT. The application can discard `bytes_consumed` or a single byte and continue INIT process.
  - Case 4: NONFATAL error during EXEC. The application can continue decoding without any action. The current frame data will be discarded.

- In the case of ADIF/ raw, the application must feed the data from the next frame start. It is assumed that the application has the knowledge about frame boundary in this case. **Note:** In the EXEC process of ADTS, LOAS, or LATM streams, the decoder reports only NONFATAL errors.

- XA_AACDEC_EXECUTE_NONFATAL_STREAM_CHANGE

  Description: In the case of ADTS/LOAS/LATM streams, this error may be reported if the decoder detects change in one or more stream parameters

  Required or suggested actions: If the application confirms this as a stream change, it should stop the EXEC process and proceed with the INIT process so that the decoder starts decoding the new stream. If the application believes this error may be due to stream error, it can discard one single byte and continue decoding. The current frame in which the parameter change is detected will be discarded.

- XA_AACDEC_EXECUTE_NONFATAL_HEADER_NOT_FOUND

  Description: This error is returned in the case of ADTS/LATM streams when there is no syncword found in the data available in the input buffer, **or** there are bytes to discard before the detected syncword.

  Required or suggested actions: The application can continue normal process.[1]

- XA_AACDEC_EXECUTE_NONFATAL_UNSUPPORTED_FEATURE
  XA_AACDEC_EXECUTE_FATAL_UNSUPPORTED_FEATURE

  Description: The decoder may report one of the UNSUPPORTED_FEATURE errors if it detects an unsupported feature in the stream. FATAL error is reported for streams of type ADIF or raw NONFATAL error is reported for streams of type ADTS, LOAS, or LATM.

  Required or suggested actions: In either case, if the application confirms that the stream contains raw features not supported by the decoder, it should stop decoding. If the application thinks this error may be reported due to stream corruption,

  - In the case of ADTS, LOAS, or LATM, since the bytes consumed is handled by the transport layer of the decoder, the application can continue decoding without further action.
  - In the case of raw or ADIF, 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.

[1] Be aware that the raw decoder is internally reset, as the decoder cannot confirm if any frame drop has caused this error.

- XA_AACDEC_EXECUTE_NONFATAL_PARTIAL_LAST_FRAME

  Description: The decoder finds the data in the input buffer is not sufficient to decode a frame after receiving `input_over` from application. The decoder consumes all the bytes in the buffer before it returns. It also sets `exec_done` in this scenario.

  Required or suggested actions: The application should handle the `exec_done` message from the decoder.

- XA_AACDEC_EXECUTE_NONFATAL_HEADER_ERROR

  Description: The decoder reports this error when it detects an error while parsing the transport header

  Required or suggested actions: Although this is a non-fatal error, the application may consider this as a fatal error in case of LATM/ADIF decoding and start feeding data from the next frame start. In the case of ADTS/LOAS, no special action is required form the application. Resync occurs in the decoder in the next EXEC / INIT API call.

- XA_AACDEC_EXECUTE_NONFATAL_EMPTY_INPUT_BUFFER
  XA_AACDEC_EXECUTE_FATAL_EMPTY_INPUT_BUFFER

  Description: This error is reported when the decoder is called without any input data (specifically, transition in input state) and `input_over` is set. In the case of EXEC process, a NONFATAL error indicates end of decoding. In the case of INIT, a FATAL error indicates unsuccessful initialization.

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

- 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: The application should reconfigure the channel routing specification and call the decoder again.

- XA_AACDEC_EXECUTE_FATAL_UNKNOWN_STREAM_FORMAT

  Description: This error is reported when the decoder could not understand the stream format.

  Required or suggested actions: The application may use SET_CONFIG_PARAM with subcommand EXTERNALBSFORMAT to specify the bitstream format.

  - For raw and LATM streams, the application should provide the stream format
  - For ADTS and LOAS streams, the auto-detection logic might have failed due to stream error, and the application may set the stream format. If the failure continues, the stream may be corrupted.

- XA_AACDEC_EXECUTE_FATAL_ADIF_HEADER_NOT_FOUND

  Description: The decoder could not find the ADIF header.

  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 application can configure the AAC decoder using the "SET CONFIG" API. The application can read parameters specific to decoded stream and 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.

List of parameters that can be configured by SET_CONFIG:

Configuration parameters supported by all AAC decoder libraries:

- external_sampling_rate (refer to Table 3-1)
- external_bsformat (refer to Table 3-2)
- mono_to_stereo (refer to Table 3-3)
- pcm_wdsz[2] (refer to Table 3-4)
- outnchans[2] (refer to Table 3-5)
- chanrouting[2] (refer to Table 3-6)
- zero_unused_chans (refer to Table 3-7)
- input_bitoffset (refer to Table 3-8)

Configuration parameters supported by libraries that can decode HE-AAC streams:

- bdownsample (refer to Table 3-9)
- downmix (refer to Table 3-10)
- sbr_signaling (refer to Table 3-11)

Configuration parameters supported only by the aacmchplus_v2_loas library:

- apply_prl (refer to Table 3-12)
- target_level (refer to Table 3-13)
- apply_drc (refer to Table 3-14)
- drc_compress (refer to Table 3-15)
- drc_boost (refer to Table 3-16)

List of parameters that can be queried by GET CONFIG:

The application can query for parameters related to the input stream / decoding process. Except for those mentioned below, these parameters can be queried only after a successful INIT.

Parameters that can be queried from any library:

- output_sample_rate (refer to Table 3-17)

[2] Thepcm_wdsz, outnchans, and chanrouting parameters must be set before the postconfig INITAPI call.

- num_channels (refer to Table 3-18)
- pcm_wdsz[3] (refer to Table 3-19)
- sbr_type (refer to Table 3-20)
- aac_samplerate (refer to Table 3-21)
- data_rate[4] (refer to Table 3-22)
- chanmap (refer to Table 3-23)
- acmod (refer to Table 3-24)
- aac_format (refer to Table 3-25)
- out_num_channels (refer to Table 3-26)
- drc_ext_present (refer to Table 3-27)
- mpeg_id (refer to Table 3-28)
- is_original (refer to Table 3-29)
- copyright_id (refer to Table 3-30)

Parameters that can be queried from Multi-Channel libraries:

- pce_status (refer to Table 3-31)
- downmix_metadata (refer to Table 3-32)
- downmix_level_dvb (refer to Table 3-33)
- parsed_drc_info (refer to Table 3-34)

Parameters that can be queried from the aacmchplus_v2_loas library only:

- prog_ref_level (refer to Table 3-35)

[3] This parameter can be queried any time after preconfig init.

[4] This parameter can be queried only after one successful frame decoding.

3.3.1 XA_API_CMD_SET_CONFIG_PARAM

Table 3-2 XA_AACDEC_CONFIG_PARAM_EXTERNALSAMPLINGRATE subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_EXTERNALSAMPLINGRATE
Description	This command sets the sampling rate of the basic AAC stream in case of raw bitstreams.
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	■ This subcommand is required only for raw stream decoding. For all other formats, the value is "ignored" ■ Note that for raw SBR streams the sampling rate of the basic AAC content may be half the output sampling rate. ■ Valid values 8000 11025 12000 16000 22050 24000 32000 44100 48000 64000 88200 96000 ■ Default value is 44100

Example

```
int externalsr = 48000;
res = (*api_func)(api_obj,
XA_API_CMD_SET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_EXTERNALSAMPLINGRATE,
(void *) &externalsr);
```

Errors

- When the input value is not valid, XA_AACDEC_CONFIG_FATAL_INVALID_EXTERNALSAMPLINGRATE

Table 3-3 XA_AACDEC_CONFIG_PARAM_EXTERNALBSFORMAT subcommand

Subcommand	`XA_AACDEC_CONFIG_PARAM_EXTERNALBSFORMAT`
Description	This command sets the bitstream format for the given stream (from application). enum type XA_AACDEC_EBITSTREAM_TYPE is defined in xa_aac_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_AACDEC_CONFIG_PARAM_EXTERNALBSFORMAT`
	`pv_value`
	`&bsformat` – Pointer to the bitstream format variable

Restrictions	■ The current version of the decoder supports one of the following bitstream types only. For all libraries: XA_AACDEC_EBITSTREAM_TYPE_AAC_RAW XA_AACDEC_EBITSTREAM_TYPE_AAC_ADTS XA_AACDEC_EBITSTREAM_TYPE_AAC_ADIF Libraries with loas also supports XA_AACDEC_EBITSTREAM_TYPE_AAC_LOAS XA_AACDEC_EBITSTREAM_TYPE_AAC_LATM ■ If the external bitstream format is set, the auto detection is disabled and the library tries to decode the given stream with the specified format only. For example, if the bitstream format is set to ADTS, and an LOAS stream is provided, the decoding process will fail as the decoder tries to interpret LOAS stream as ADTS and the generated output will be unpredictable. ■ If the external bitstream format is not set or set to XA_AACDEC_EBITSTREAM_TYPE_AAC_UNKNOWN (default), the decoder activates the auto detect functionality and tries to detect ADIF, ADTS, and LOAS streams. ■ For raw and LATM streams, this parameter must be set by the application.

Example

```
XA_AACDEC_EBITSTREAM_TYPE
bsformat=XA_AACDEC_EBITSTREAM_TYPE_AAC_RAW
res = (*api_func)(api_obj,
XA_API_CMD_SET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_EXTERNALBSFORMAT,
(void *) &bsformat);
```

Errors

- When the input value is not valid, XA_AACDEC_CONFIG_FATAL_INVALID_EXTERNALBSFORMAT

Table 3-4 XA_AACDEC_CONFIG_PARAM_TO_STEREO subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_TO_STEREO
Description	This command enables or disables conversion 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	■ Valid values: 0 - mono streams are presented as a single channel 1 - mono streams are presented as two identical channels (default) ■ For Non-mono streams, this setting is ignored.

Example

```
int mono_to_stereo = 0;
res = (*api_func)(api_obj,
XA_API_CMD_SET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_TO_STEREO,
(void *) &mono_to_stereo);
```

Errors

- When the input value is not valid, XA_AACDEC_CONFIG_FATAL_INVALID_TO_STEREO

Table 3-5 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 bits.
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 width of the PCM sample variable
Restrictions	■ In 24-bit format, the samples are stored in the 24 MSBs of each output 32-bit word; the 8 LSBs are set to 0. ■ In 16-bit format the samples are stored in 16-bit words. ■ Valid values 16 - 16-bit PCM samples 24 - 24-bit PCM samples (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

- When the input value is not valid, XA_AACDEC_CONFIG_FATAL_INVALID_PCM_WDSZ

Table 3-6 XA_AACDEC_CONFIG_PARAM_OUTNCHANS subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_OUTNCHANS
Description	This command sets the maximum number of decoded channels to be placed 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_SET_CONFIG_PARAM i_idx XA_AACDEC_CONFIG_PARAM_OUTNCHANS pv_value &outnchans – Pointer to the number of output channels variable
Restrictions	Valid values Stereo libraries: the default and only valid value is 2. Multi-Channel libraries: 2 through 8 (default 8)

Example

```
int outnchans = 6;
res = (*api_func)(api_obj,
XA_API_CMD_SET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_OUTNCHANS,
(void *)&outnchans);
```

Errors

- When the input value is not valid, XA_AACDEC_CONFIG_FATAL_INVALID_OUTNCHANS

Table 3-7 XA_AACDEC_CONFIG_PARAM_CHANROUTING subcommand

<table>
<tr><td>Subcommand</td><td>XA_AACDEC_CONFIG_PARAM_CHANROUTING</td></tr>
<tr><td>Description</td><td>Controls the output channel routing in the output PCM buffer.
To set this parameter, channel index is specified in byte 0 (bits 0-7) and the sample offset in the interleaving order is specified in byte 1 (bits 8 – 15) of the int variable chanrouting.
Channel indices are as below:
<table>
<tr><td>0</td><td>L</td></tr>
<tr><td>1</td><td>C</td></tr>
<tr><td>2</td><td>R</td></tr>
<tr><td>3</td><td>l</td></tr>
</table>
<table>
<tr><td>4</td><td>r</td></tr>
<tr><td>5</td><td>Sbl (or Cs)</td></tr>
<tr><td>6</td><td>Sbr</td></tr>
<tr><td>7</td><td>LFE</td></tr>
</table></td></tr>
<tr><td>Actual Parameters</td><td>p_xa_module_obj
api_obj – Pointer to API Structure
i_cmd
XA_API_CMD_SET_CONFIG_PARAM
i_idx
XA_AACDEC_CONFIG_PARAM_CHANROUTING
pv_value
&chanrouting – Pointer to the channel routing variable</td></tr>
<tr><td>Restrictions</td><td>■ To set up correctly, this parameter needs to be set outnchans times to completely specify the routing of all the output channels. Incomplete setup may lead to failure of post config initialization or during decoder initialization process.
■ The sample offset value and the channel index must be less than outnchans. This setup is validated during post configuration and also during raw decoder initializing process.
■ In case this parameter is not set, the samples are routed as per their natural order (as they appear in the bit-stream) in the output buffer. To know the natural order or the order currently used, the application can query for chanmap (refer to Table 3-23)</td></tr>
</table>

Example

```
unsigned int chanrouting;
/* Route the L channel to sample offset 0 in the
output */
chanrouting = (0 << 8) | 0; /* L → 0 */
res = (*api_func)(api_obj,
XA_API_CMD_SET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_CHANROUTING,
(void *) &chanrouting);
/* Route the R channel to sample offset 1 in the
output */
chanrouting = (1 << 8) | 2; /* R → 2 */
res = (*api_func)(api_obj,
XA_API_CMD_SET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_CHANROUTING,
(void *) &chanrouting);
```

Errors

- When the input value is not valid, XA_AACDEC_CONFIG_FATAL_INVALID_CHANROUTING

Table 3-8 XA_AACDEC_CONFIG_PARAM_ZERO_UNUSED_CHANS subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_ZERO_UNUSED_CHANS
Description	Enable (1) or disable (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 – Output PCM samples corresponding to unused channels are left unmodified (Default) ■ 1 – enable zeroing of unused output channels ■ Enabling this may lead to higher CPU load and memory bandwidth requirements in cases when the number of decoded channels (num_channels) is less than the maximum number of channels to be placed in the input buffer (outnchans).

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

- When the input value is not valid, XA_AACDEC_CONFIG_FATAL_INVALID_ZERO_UNUSED_CHANS

Table 3-9 XA_AACDEC_CONFIG_PARAM_INPUT_BITOFFSET subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_INPUT_BITOFFSET
Description	This API is used to discard a few bits in the input bitstream and applicable only for raw stream decoding. In general, the decoder reads data from the first byte. In some cases such as LATM, the raw data may not start from first bit of the byte. In this case, the application can call this API to discard the bits before beginning the decoding process
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_INPUT_BITOFFSET pv_value &bit_offset – bit offset value (usually between 0-7)
Restrictions	■ This parameter can be changed during run-time ■ Valid Values: 0-7; This API can also be used to discard more number of bytes by providing the value in bits. i.e., setting this value to 32 will discard first four bytes in the bitstream. However, this feature is not fully tested, hence recommend to use the values between 0 and 7 ■ The value set through this API will be cleared after every INIT/EXEC API call. ■ If this API is not called, bit offset will be assumed as 0. ■ If the input stream format is not raw, bit offset value set through this API will be ignored. ■ If the bit_offset value is negative, it will be ignored.

Example

```
int bit_offset = 3;
res = (*api_func)(api_obj,
XA_API_CMD_SET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_INPUT_BITOFFSET,
(void *) &bit_offset);
```

Errors

- No specific error

Table 3-10 XA_AACDEC_CONFIG_PARAM_BDOWNSAMPLE subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_BDOWNSAMPLE
Description	This command sets the SBR downsample flag
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_BDOWNSAMPLE pv_value &bdownsample – Pointer to the SBR downsample flag variable
Restrictions	■ Valid values ■ 0 – disabled (default) ■ 1 – enabled ■ This setting will be ignored for non-SBR streams. ■ This subcommand is not available for aac, aac_loas, aacmch, and aacmch_loas libraries.

Example

```
int bdownsample = 1;
res = (*api_func)(api_obj,
XA_API_CMD_SET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_BDOWNSAMPLE,
(void *) &bdownsample);
```

Errors

- When the input value is not valid, XA_AACDEC_CONFIG_FATAL_INVALID_BDOWNSAMPLE

Table 3-11 XA_AACDEC_CONFIG_PARAM_BBITSTREAMDOWNMIX subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_BBITSTREAMDOWNMIX
Description	This command sets the downmix flag. The flag is used to configure the SBR tool to perform SBR processing on a downmixed mono channel, instead of channel pair. The mono channel is derived by downmixing the input channel pair.
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_BBITSTREAMDOWNMIX pv_value &downmix – Pointer to the downmix flag variable
Restrictions	■ Valid values ▪ 0 – disabled (default) ▪ 1 – enabled ■ Enabling this flag applies SBR processing on mono channel to reduce computational requirement of the processing. If the input to SBR processing is a channel pair, the channels are downmixed into a mono channel and then passed through SBR processing. ■ This setting will be ignored for non-SBR streams. ■ This subcommand is not available for aac, aac_loas, aacmch, and aacmch_loas libraries.

Example

```
int downmix = 1;
res = (*api_func)(api_obj,
XA_API_CMD_SET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_BBITSTREAMDOWNMIX,
(void *) &downmix);
```

Errors

- When the input value is not valid, XA_AACDEC_CONFIG_FATAL_INVALID_BBITSTREAMDOWNMIX

Table 3-12 XA_AACDEC_CONFIG_PARAM_SBR_SIGNALING subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_SBR_SIGNALING
Description	This command enables/disables the SBR processing or sets it to an auto mode for using the SBR tool in decoding of the input streams
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_SBR_SIGNALING pv_value &sbr_signaling – Pointer to the flag
Restrictions	■ Valid values ▪ 0 – disable the SBR tool during decoding of all types of input bitstreams ▪ 1 – Enable and apply the SBR tool during decoding of all types of input bitstreams. This forces SBR processing even for plain AAC input streams ▪ 2 – Apply the SBR tool only for decoding of input bitstreams containing SBR information; Auto mode (default) ■ The sbr_signaling parameter controls and modifies the audio object type (sbr_type) returned by the library (Refer to Table 3-20). If sbr_signaling is 0, then sbr_type is always 0 (plain AAC audio object type). If sbr_signaling is 1, then sbr_type is either 1 or 2 (indicating aacPlus object decoding). ■ This subcommand is not available for aac, aac_loas, aacmch and aacmch_loas libraries.

Example

```
int sbr_signaling = 1;
res = (*api_func)(api_obj,
XA_API_CMD_SET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_SBR_SIGNALING,
(void *)&sbr_signaling);
```

Errors

- When the input value is not valid, XA_AACDEC_CONFIG_FATAL_INVALID_SBR_SIGNALING

Table 3-13 XA_AACDEC_CONFIG_PARAM_ENABLE_APPLY_PRL subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_ENABLE_APPLY_PRL
Description	Enable or disable application of program reference level scaling to a desired target value specified through XA_AACDEC_CONFIG_PARAM_ENABLE_TARGET_LEVEL (Refer to table 3-13).
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_ENABLE_APPLY_PRL pv_value &apply_prl – Pointer to a binary flag variable controlling the application of program ref level scaling
Restrictions	■ This configuration is available only with the aacmchplus_v2_loas library ■ This parameter can be changed during runtime ■ Valid values: ▪ 0 – No scaling is applied to the decoder output (default) ▪ 1 – Dynamic range scaling is applied in the spectral domain

Example

```
int apply_prl = 1;
res = (*api_func)(api_obj,
XA_API_CMD_SET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_APPLY_PRL,
(void *) &apply_prl);
```

Errors

- When the input value is not valid, XA_AACDEC_CONFIG_FATAL_INVALID_ PRL_PARAMS

Table 3-14 XA_AACDEC_CONFIG_PARAM_TARGET_LEVEL subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_TARGET_LEVEL
Description	This is a 7-bit value to set the desired target level of the decoder output. This is only applicable if the program reference level is available in the bit-stream. If the program reference level information is **not** available in the stream, then the output does not undergo any scaling. In addition, the output of the decoder is scaled to the desired target output level only if the parameter apply_prl mentioned in Table 3-12 is set to 1.
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_TARGET_LEVEL pv_value &target_level – Pointer to a 7-bit value indicating desired output level
Restrictions	■ This configuration is available only with aacmchplus_v2_loas Library ■ This parameter can be changed during runtime ■ This is a 7-bit unsigned value stored in a 32-bit word. The corresponding desired output level can be decided based on the value as follows: desired_target_level_in_dB = -0.25 * target_level ■ Valid values: 0 to 127 (default: 124)

Example

```
float desired_output_level_db = -24.0;
int        target_level        =        (int)(-4*
desired_output_level_db);
res = (*api_func)(api_obj,
XA_API_CMD_SET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_TARGET_LEVEL,
(void *) &target_level);
```

Errors

- When the input value is not valid, XA_AACDEC_CONFIG_FATAL_INVALID_ PRL_PARAMS

Table 3-15 XA_AACDEC_CONFIG_PARAM_ENABLE_APPLY_DRC subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_ENABLE_APPLY_DRC
Description	Enable (1) or disable (0) application of dynamic range compression. The percentage of dynamic scaling is controlled by the drc_compress and drc_boost values described in Table 3-15 and Table 3-16.
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_ENABLE_APPLY_DRC pv_value &apply_drc – Pointer to a binary flag variable controlling the application of dynamic range compression
Restrictions	■ This configuration is available only with aacmchplus_v2_loas Library ■ This parameter can be changed during runtime ■ Valid values: ▪ 0 – No dynamic range scaling is applied to the decoder output (Default) ▪ 1 – Dynamic range scaling is applied in the spectral domain

Example

```
int apply_drc = 1;
res = (*api_func)(api_obj,
XA_API_CMD_SET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_APPLY_DRC,
(void *) &apply_drc);
```

Errors

- When the input value is not valid, XA_AACDEC_CONFIG_FATAL_INVALID_ DRC_PARAMS

Table 3-16 XA_AACDEC_CONFIG_PARAM_DRC_COMPRESS_FAC subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_DRC_COMPRESS_FAC
Description	This parameter controls the scaling of loud signals so that they are compressed as mentioned in MPEG-4 Audio standard [2]. This parameter is passed as a fraction stored in a 9.23 format. The scaling is applied only if the apply_drc flag described in Table 3-14 is set.
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_DRC_COMPRESS_FAC pv_value &drc_compress – Pointer to a fractional variable controlling the percentage of the application of dynamic range compression (cut)
Restrictions	■ This configuration is available only with the aacmchplus_v2_loas library ■ This parameter can be changed during runtime ■ These are integers representing values in a 9.23 format. For example, dynamic range compression (cut) of 100% is represented as 32-bit integer value of 0x00800000. Dynamic range compression of 50% shall be represented by 0x00400000

Example

```
float percent_drc_cut = 60.0;
int
drc_compress=(int)(percent_drc_cut/100.0*(1<<23))
;
res = (*api_func)(api_obj,
XA_API_CMD_SET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_DRC_COMPRESS_FAC,
(void *) &drc_compress);
```

Errors

- When the input value is not valid, XA_AACDEC_CONFIG_FATAL_INVALID_ DRC_PARAMS

Table 3-17 XA_AACDEC_CONFIG_PARAM_DRC_BOOST_FAC subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_DRC_BOOST_FAC
Description	This parameter controls the scaling of soft signals so that they are boosted as mentioned in the MPEG-4 Audio standard [2]. This parameter is passed as a fraction in a 9.23 format. The scaling is applied only if the apply_drc flag described in Table 3-14 is set.
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_DRC_BOOST_FAC pv_value &drc_boost – Pointer to a fractional variable controlling the percentage of the application of dynamic range compression (boost)
Restrictions	■ This configuration is available only with the aacmchplus_v2_loas library ■ This parameter can be changed during runtime ■ These are integers representing values in a 9.23 format. For example, dynamic range boost of 100% is represented as 32-bit integer value of 0x00800000. Dynamic range boost of 25 % shall be represented by 0x00200000

Example

```
float percent_drc_boost = 40.0;
int drc_boost
=(int)(percent_drc_boost/100.0*(1<<23));
res = (*api_func)(api_obj,
XA_API_CMD_SET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_DRC_BOOST_FAC,
(void *) &drc_boost);
```

Errors

- When the input value is not valid, XA_AACDEC_CONFIG_FATAL_INVALID_ DRC_PARAMS

Table 3-18 XA_AACDEC_CONFIG_PARAM_ENABLE_FRAME_BY_FRAME_DECODE subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_ENABLE_FRAME_BY_FRAME_DECODE
Description	The command sets the operating mode for the decoder: streaming mode or frame-by-frame mode for ADTS/LOAS streams. In streaming mode, the decoder is designed in such a way that it confirms the validity of the frame by looking ahead and verifying the next frame header. This mechanism allows the decoder to check and notify the application about stream change by connecting the frame info between current, previous, and future frames headers. In frame-by-frame mode, the decoder will not look ahead for the next frame header to check the frame validity and stream change detection is disabled. The frame-by-frame mode is less robust in error prone systems. It is up to the application to detect events such as bitstream errors and stream changes and handle the decoder properly when such events happen.
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_ENABLE_FRAME_BY_FRAME_DECODE pv_value &enableframeByFrameDecode – Pointer to the operating mode variable
Restrictions	■ This parameter can be changed during runtime ■ Enabling this mode will disable stream change detection. ■ Behavior of auto format detection during init will not be affected by this setting and look ahead is still required to detect stream. User must set the stream format to use frame-by-frame decode mode correctly. ■ Valid value: 0 (streaming mode, the default), 1 (frame-by-frame mode).

Example

```
Int enableframeByFrameDecode = 1;
res = (*api_func)(api_obj,
XA_API_CMD_SET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_DRC_BOOST_FAC
XA_AACDEC_CONFIG_PARAM_ENABLE_FRAME_BY_FRAME_DECO
DE
(void *) &enableframeByFrameDecode);
```

Errors

- No specific error

3.3.2 XA_API_CMD_GET_CONFIG_PARAM

Table 3-19 XA_AACDEC_CONFIG_PARAM_OUT_SAMPLERATE subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_OUT_SAMPLERATE
Description	This command gets the output sample rate (in Hz)
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_OUT_SAMPLERATE 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_OUT_SAMPLERATE,
(void *) &samp_freq);
```

Errors

- No specific error

Table 3-20 XA_AACDEC_CONFIG_PARAM_NUM_CHANNELS subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_NUM_CHANNELS
Description	This command gets the number of decoded channels present in the output buffer. Values in the range of 1 to 8.
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 number of channels 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

- No specific error

Table 3-21 XA_AACDEC_CONFIG_PARAM_PCM_WDSZ subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_PCM_WDSZ
Description	This command gets the output bit-width. Possible return values are 16 and 24. This returns a default value or the value set by the application using XA_API_CMD_SET_CONFIG_PARAM with subcommand XA_AACDEC_CONFIG_PARAM_PCM_WDSZ explained in Table 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_AACDEC_CONFIG_PARAM_PCM_WDSZ pv_value &pcm_wdsz – Pointer to the width of the PCM sample variable
Restrictions	This API is optional and the information provided by this API can also be determined by the application

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

- No specific error

Table 3-22 XA_AACDEC_CONFIG_PARAM_SBR_TYPE subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_SBR_TYPE
Description	This command gets the sbr type for the stream under decoding. 0 Plain AAC 1 AAC+ V1 (SBR) 2 AAC+ V2 (Parametric Stereo)
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_SBR_TYPE pv_value &sbr_type – Pointer to the SBR type variable
Restrictions	None

Example

```
int sbr_type;
res = (*api_func)(api_obj,
XA_API_CMD_GET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_SBR_TYPE,
(void *) &sbr_type);
```

Errors

- No specific error

Table 3-23 XA_AACDEC_CONFIG_PARAM_AAC_SAMPLERATE subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_AAC_SAMPLERATE
Description	This command gets the sample rate (in Hz) at which the plain AAC decoder is operating. When SBR is in use, this figure can be half 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_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

- No specific error

Table 3-24 XA_AACDEC_CONFIG_PARAM_DATA_RATE subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_DATA_RATE
Description	This command gets the approximate average data rate (in bits) of the encoded stream. For ADIF streams, the data rate is read from ADIF header and the value remains same throughout the decoding. For non-ADIF stream, the data rate is calculated and updated after every successful decoding of a frame.
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 input data rate variable
Restrictions	The average bit rate is stabilized over time, hence the application may read after a few (>15) frames of data.

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

- XA_AACDEC_CONFIG_NONFATAL_DATA_RATE_NOT_SET is returned if this API is called before first successful frame decoding.

Table 3-25 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. The nibbles 0 to 7 of the chanmap variable are set to channel index values based on their sample offsets in the interleaved output PCM buffer. Channel index to Channel mapping is provided in Table 3-6. If a channel (with channel index C) appears at sample offset N in the interleaved output PCM buffer, then Nth nibble of the chanmap parameter is set to C. The unused sample offsets are set to value 0xF. For example: chanmap = 0xFFFF5201 indicates that: the center channel (channel index 1) is present at sample offset of 0, the left channel (channel index 0) is present at sample offset 1, the right channel (channel index 2) is present at sample offset 2, the center surround (channel index 5) is present at sample offset 4. There are no other decoded channels present at the remaining sample offsets.
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 chanmap 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

- No specific error

Table 3-26 XA_AACDEC_CONFIG_PARAM_ACMOD subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_ACMOD
Description	Information about the audio coding mode (encoded channel configuration) of the input bitstream. enum type XA_AACDEC_CHANNELMODE is defined in xa_aac_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_AACDEC_CONFIG_PARAM_ACMOD pv_value &acmod – Pointer to the audio coding mode variable
Restrictions	None

Example

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

Errors

- No specific error

Table 3-27 XA_AACDEC_CONFIG_PARAM_AAC_FORMAT subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_AAC_FORMAT
Description	This command gets the stream format of the input stream. If the stream format was set using XA_AACDEC_CONFIG_PARAM_EXTERNALBSFORMAT, the same will be returned. If the stream format was detected automatically by the decoder, the detected value will be returned. enum type XA_AACDEC_EBITSTREAM_TYPE is defined in xa_aac_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_AACDEC_CONFIG_PARAM_AAC_FORMAT pv_value &aac_format – Pointer to the AAC format variable
Restrictions	None

Example

```
XA_AACDEC_EBITSTREAM_TYPE aac_format;
res = (*api_func)(api_obj,
XA_API_CMD_GET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_AAC_FORMAT,
(void *)&aac_format);
```

Errors

- No specific error

Table 3-28 XA_AACDEC_CONFIG_PARAM_OUTNCHANS subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_OUTNCHANS
Description	This returns a default value or the value set by the application using XA_API_CMD_SET_CONFIG_PARAM with subcommand XA_AACDEC_CONFIG_PARAM_OUTNCHANS explained in Table 3-5.
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 – Number of output channels (both valid and invalid)
Restrictions	This API is optional and the information provided by this API can also be determined by the application

Example

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

Errors

- No specific error

Table 3-29 XA_AACDEC_CONFIG_PARAM_DRC_EXT_PRESENT subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_DRC_EXT_PRESENT
Description	This parameter returns a flag to indicate whether the payload contains DRC extension payload or not. Return value 1 indicates that DRC extension is present, and 0 indicates that it is absent
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_DRC_EXT_PRESENT pv_value &drc_ext_flag – Pointer to the DRC present variable
Restrictions	None

Example

```
int drc_ext_flag;
res = (*api_func)(api_obj,
XA_API_CMD_GET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_DRC_EXT_PRESENT,
(void *) &drc_ext_flag);
```

Errors

- No specific error

Table 3-30 XA_AACDEC_CONFIG_PARAM_MPEG_ID subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_MPEG_ID
Description	This parameter returns a flag to indicate whether the payload is MPEG2-AAC or MPEG4-AAC. The value is set to 1 if the audio data in the ADTS stream is MPEG-2 AAC (see ISO/IEC 13818-7), and set to 0 if the audio data is MPEG-4.
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_MPEG_ID pv_value &Mpeg_ID – Pointer to a variable Mpeg_ID
Restrictions	This value is available only if the payload is in ADTS format (as explained in Section 1.A.4.3 of ISO/IEC 14496-3)

Example

```
int Mpeg_ID;
res = (*api_func)(api_obj,
XA_API_CMD_GET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_MPEG_ID,
(void *) &Mpeg_ID);
```

Errors

- No specific error

Table 3-31 XA_AACDEC_CONFIG_PARAM_ORIGINAL_OR_COPY subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_ORIGINAL_OR_COPY
Description	This parameter returns a flag to indicate whether the bitstream is original or copy. The value is set to 1 if the bitstream is original, and set to 0 if the bitstream is copy.
Actual Parameters	p_xa_module_obj api_obj – Pointer to API Structure i_cmd XA_API_CMD_GET_CONFIG_PARAM i_idx XA_AACDEC_CONFIG_PARAM_ORIGINAL_OR_COPY pv_value &orig_ID – Pointer to the original/copy indication variable
Restrictions	This value is available only if the bitstream is in ADTS or ADIF format (as explained in Section 2.4.2.3 of ISO/IEC 11172-3)

Example

```
int orig_ID;
res = (*api_func)(api_obj,
XA_API_CMD_GET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_ ORIGINAL_OR_COPY,
(void *) &orig_ID);
```

Errors

- No specific error

Table 3-32 XA_AACDEC_CONFIG_PARAM_COPYRIGHT_ID_PTR subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_COPYRIGHT_ID_PTR
Description	This parameter returns a pointer to an array of nine unsigned characters (representing 72 bits of Copyright Identification information received from the bitstream).
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_COPYRIGHT_ID_PTR pv_value &copyright_info – Pointer to the copyright_info array
Restrictions	■ This API exposes the pointer to an internal element of the decoder. The application should treat this pointer as READ-ONLY and reads only 9 bytes from the pointer. ■ This information is available only if the bitstream is in ADTS or ADIF format (as explained in Section 1.A.3 of ISO/IEC 14496-3).

Example

```
unsigned char *copyright_info;
res = (*api_func)(api_obj,
XA_API_CMD_GET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_COPYRIGHT_ID_PTR,
(void *) &copyright_info);
```

Errors

- No specific error

Table 3-33 XA_AACDEC_CONFIG_PARAM_PCE_STATUS subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_PCE_STATUS
Description	This parameter returns a value to indicate whether a PCE (Program Config Element) is present in the encoded frame enum type xa_aac_dec_pce_status is defined in xa_aac_dec_api.h LOAS/LATM — PCE status is read from LATMheader. If PCE is not present in LATM header, it is read from raw frame ADTS, raw — PCE status is read from raw frame ADIF — PCE status read from ADIF header
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_PCE_STATUS pv_value &PCEStatus – Pointer to a variable PCE status as defined below
Restrictions	This parameter is available only with Multi-Channel Libraries

Example

```
xa_aac_dec_pce_status PCEStatus;
res = (*api_func)(api_obj,
XA_API_CMD_GET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_DWNMIX_METADATA,
(void *) &PCEStatus);
```

Errors

- No specific error

Table 3-34 XA_AACDEC_CONFIG_PARAM_DWNMIX_METADATA subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_DWNMIX_METADATA
Description	This parameter is a pointer to a structure that is filled with data related to downmix present in (PCE – program config element) as specified in the MPEG4-AAC audio document ISO/IEC 14496-3 in Section 4.4.1.1. This information is either a part of the LATMheader or a part of syntax element of the core encoded frame. struct type xa_aac_dec_dwnmix_metadata_t is defined in xa_aac_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_AACDEC_CONFIG_PARAM_DWNMIX_METADATA pv_value &pDmxMetaData – Pointer to a structure containing downmix data as defined below
Restrictions	This parameter is available only with Multi-Channel Libraries

Example

```
xa_aac_dec_dwnmix_metadata_t dmx_meta_data;
res = (*api_func)(api_obj,
XA_API_CMD_GET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_DWNMIX_METADATA,
(void *) &dmx_meta_data);
```

Errors

- No specific error

Table 3-35 XA_AACDEC_CONFIG_PARAM_DWNMIX_LEVEL_DVB subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_DWNMIX_LEVEL_DVB
Description	This parameter is a pointer to a structure, which is filled with data related to downmix levels present in ancillary data (DSE), as specified in ETSI TS101.154 v1.9.1. These are relevant for downmixing the outputs to stereo for DVB-like applications. struct type xa_aac_dec_dwnmix_level_dvb_info_t is defined in xa_aac_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_AACDEC_CONFIG_PARAM_DWNMIX_LEVEL_DVB pv_value &dvb_dmx_info – Pointer to the structure containing downmix level data as defined below
Restrictions	■ This parameter is available only with Multi-Channel Libraries ■ The dvb info structure values retain their old values if the element new_dvb_downmix_data is equal to 0. If the parsed data element does not contain timecodes, then the *_timecode_value and *_timecode_on parameters remain 0.

Example

```
xa_aac_dec_dwnmix_level_dvb_info_t dvb_dmx_info;
res = (*api_func)(api_obj,
XA_API_CMD_GET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_DWNMIX_LEVEL_DVB,
(void *) &dvb_dmx_info);
```

Errors

- No specific error

Table 3-36 XA_AACDEC_CONFIG_PARAM_PARSED_DRC_INFO subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_PARSED_DRC_INFO
Description	This parameter expects a pointer to the array of MAX_NUM_CHANNELS (8) elements of type xa_aac_dec_parsed_drc_info_t; it fills these elements with DRC information, which is parsed from payload.
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_PARSED_DRC_INFO pv_value &drc_info – Pointer to MAX_NUM_CHANNELS sized array of structures of type xa_aac_dec_parsed_drc_info_t
Restrictions	■ This parameter is available only with Multi-Channel Libraries ■ Parsed_drc_info values are updated after every successful frame decoding. ■ The flag drc_info_valid (of structure xa_aac_dec_parsed_drc_info_t) in each element indicates if the DRC information in that element is valid or not for given payload.

Example

```
xa_aac_dec_parsed_drc_info_t
drc_info[MAX_NUM_CHANNELS];
res = (*api_func)(api_obj,
XA_API_CMD_GET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_PARSED_DRC_INFO,
(void *) drc_info);
```

Errors

- No specific error

Table 3-37 XA_AACDEC_CONFIG_PARAM_PROG_REF_LEVEL subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_PROG_REF_LEVEL
Description	This parameter returns the program reference level if present in the bitstream. This is a 7-bit unsigned value stored in a 32-bit word. The value indicates the PCM level of the decoded output. The dB value corresponding to the output level can be determined as follows: output_level_in_dB = -0.25*prog_ref_level
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_PROG_REF_LEVEL pv_value &prog_ref_level – Pointer to a 32-bit variable containing a 7-bit parameter parsed from the bit-stream
Restrictions	■ This parameter is available only with the aacmchplus_v2_loas library ■ If prog_ref_level is not available in the input bitstream or until the decoder detects the presence of a program_ref_level parameter in the input stream, then the returned value is 0xffff ffff. In this case, the application should use the default value -31dB.

Example

```
int prog_ref_level;
float  prog_ref_level_in_db;
res = (*api_func)(api_obj,
XA_API_CMD_GET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_DWNMIX_METADATA,
(void *) &prog_ref_level);
if (prog_ref_level == -1) {
printf("program ref level is not available in the bitstream
(default value is assumed to be -31dB)\n");
}
else {
prog_ref_level_in_db = -0.25*prog_ref_level;
}
```

Errors

- No specific error

Table 3-38 XA_AACDEC_CONFIG_PARAM_ENABLE_FRAME_BY_FRAME_DECODE subcommand

Subcommand	XA_AACDEC_CONFIG_PARAM_ENABLE_FRAME_BY_FRAME_DECODE
Description	This API returns the operating mode of the decoder for ADTS/LOAS streams. If the decoder is running in frame-by-frame decoding mode, it returns 1. If the decoder is running in streaming mode, it returns 0.
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_ENABLE_FRAME_BY_FRAME_DECODE pv_value &enableframeByFrameDecode – Pointer to the operating mode variable
Restrictions	None

Example

```
int enableframeByFrameDecode;
res = (*api_func)(api_obj,
XA_API_CMD_GET_CONFIG_PARAM,
XA_AACDEC_CONFIG_PARAM_ENABLE_FRAME_BY_FRAME_DECO
DE
(void *) &enableframeByFrameDecode);
```

Errors

- No specific error

4. Introduction to the Example Testbench

The supplied testbench consists of the following files:

- Testbench 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`
- Sample parameter file to run the testbench (`test/build`)
  - `paramfilesimple.txt`

4.1 Making the Executable

To build the application, follow these steps:

1. Go to `test/build`.
2. In the console, type: (where <*lib*> is one of `aac`, `aacplus`, `aacplus_v2`, `aacmch`, `aacmchplus`, `aacmchplus_v2, aacmchplus_v2_loas`)

   `xt-make –f makefile_testbench_sample clean` <*lib*>

This will build the decoder example testbench `xa_<lib>_dec_test`.

3. To build the decoder testbench with LOAS/LATM support for other decoder variants (other than `aacmchplus_v2_loas,` which can be built by step 2 above), in the console, type: (where <*lib*> is one of `aac`, `aacplus`, `aacplus_v2`, `aacmch`, `aacmchplus`)

   `xt-make –f makefile_testbench_sample clean` <*lib*> LINK_LOAS=1

This will build the decoder example testbench `xa_<lib>_loas_dec_test`.

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

1. Go to the `build` directory.
2. Type:

   `$xt-make clean <lib> install`

If <lib> is one of `aacmchplus_v2, aacmchplus_v2_loas:`

- This will build the `xa_<lib>_dec.a` library and copy it to the `lib` directory.

If <lib> is one of `aac`, `aacplus`, `aacplus_v2`, `aacmch`, `aacmchplus`:

- This will build the `xa_<lib>_dec.a` and `xa_<lib>_loas_dec.a` libraries and copy them to the `lib` directory.

The aac, aacplus, aacplus_v2, aacmch, aacmchplus object xws packages will use the non-loas library by default. To switch to the loas library, follow the steps below (here aac is used as an example).

1. In the ProjectExplorer area of Xplorer, click the triangle to the left of libxa_aac_dec to expand the folder; then right-click ulibxa_aac_dec/lib/xa_aac_dec_loas.a and select **Unmanaged Binary Info**. Then select the appropriate config (e.g. AE_HiFi3_LE5) and click **OK**.

2. Right-click testxa_aac_dec and select **Library Dependencies…**, ulibxa_aac_dec/lib/xa_aac_dec_loas.a (AE_HiFi3_LE5) is now shown in the Available libraries area. Select this file and click **Add**. The file is now shown in the Selected libraries area.

3. Select ulibxa_aac_dec/lib/xa_aac_dec.a (AE_HiFi3_LE5) and click **Remove**, then **Apply** and **OK**.

To build the loas library with the aac, aacplus, aacplus_v2, aacmch, or aacmchplus source xws package, follow the steps below (again using aac as an example).

1. In the ProjectExplorer area of Xplorer, click the triangle to the left of libxa_aac_dec to expand the folder; then double-click Makefile.include to open the file. Change the line OBJS = xxx to OBJS = algo/transport/src/loas.o xxx and click **Save**. Next browse to algo/transport/src/loas.c and right-click to select **Build->Include**.

2. Right-click libxa_aac_dec and select **Build Properties…**. Select **CommonTarget** as target.

   a. Click to select the **Symbols** tab; then click the **Add symbol** icon (green plus sign) “” to add two symbols AACLOAS_SUPPORT and LOAS_SUBFRAME_SUPPORT.

   b. From the folder structure on the left, browse to algo/transport/src/loas.c left. Select the **Addl compiler** tab and type -Os –x c++ in the Local: area.

   c. Verify that the new symbols and options are shown in the All Options area at the bottom.

3. Select P:libxa_aac_dec and click **Build Active**. Please note that the library is still called libxa_aac_dec.a, but it now supports LOAS.

4.2 Usage

The sample application executable can be run with command-line options or with a parameter file. The command-line usage is as follows:

```
xt-run xa_<lib>_dec_test -ifile:<infile> -ofile:<outfile>
            [-b<bsformat>]
            [-d<bdownsample>]
            [-f<to_stereo>]
            [-m<downmix>]
            [-n<outnchans>]
            [-p<extsr>]
            [-w<pcm_wdsz>]
            [-x<sbr_signaling>]
            [-0..7<ch>]
```

The following options are available for a LOAS build:

```
            [--d<drc>]
            [--dC<drc_compress>]
            [--dB<drc_boost>]
            [--p<prl>]
            [--pL<target_level>]
```

Where:

<lib>	One of `aac`, `aacplus`, `aacplus_v2`, `aacmch`, `aacmchplus`, `aacmchplus_v2` `aacmchplus_v2_loas`, `aac_loas`, `aacplus_loas`, `aacplus_v2_loas`, `aacmch_loas`, `aacmchplus_loas`
<infile>	Name of the AAC input file
<outfile>	Name of the output ".wav" file
<bsformat>	Bitstream format (`-b raw, adif, adts, latm, loas`)
<bdownsample>	Enables (1) or disables (0) downsampled SBR mode (disabled by default)
<to_stereo>	Enables (1) or disables (0) duplication of mono output to interleaved stereo (enabled by default)
<downmix>	Enables (1) or disables (0) SBR mono downmix (disabled by default)
<outnchans>	Maximum number of channels to be decoded (2 to 8)
<extsr>	The sample rate of a raw bitstream (`-p 44100`)
<pcm_wdsz>	Output PCM word size – 16 or 24 (default)
<sbr_signaling>	Disable (0), enable (1) or turn SBR processing in auto mode (2) (auto by default)

<*ch*>	Input channel (L, C, R, l, r, Sbl, Sbr, LFE) to route to an arbitrary output buffer offset (0 through 7): -0L -1C -2R -3l -4r -5Sbl -6Sbr -7LFE
<*drc*>	Enable (1) / Disable (0) flag for applying DRC (default = 0)
<*drc_compress*>	DRCCompression factor between 0.0 to 100.0 (default value is 0.0)
<*drc_boost*>	DRC Boost factor between 0.0 to 100.0 (default value is 0.0)
<*prl*>	Enable (1) / Disable flag (0) for applying Program Ref level. (default value 0)
<*target_level*>	Target Level value between 0 to 127, indicating level in dB as -0.25*target_level (default value is 124)

Refer to the parameter definitions in Section 3.3 for a full description of their usage. Note that the space between the option name and the option value is optional.

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

Following is the syntax for writing the paramfilesimple.txt file:

```
@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
```

The AACMCH Decoder can be run for multiple test files using the different command lines. The syntax for command lines in the parameter file is the same as the syntax for specifying options on the command line to the testbench program.

Note	All the @<command>s should be at the first column of a line except the @New_line command.
Note	All the @<command>s are case sensitive. If the command line in the parameter file has to be divided into two parts on two different lines, use the @New_line command, as shown in the following example. <*command line part 1*> @New_line <*command line part 2*>.
Note	Blank lines will be ignored.
Note	Individual lines can be commented out using "//" at the beginning of the line.

5. References

[1] *ISO/IEC 13818-7 Information technology -- Generic coding of moving pictures and associated audio information -- Part 7: Advanced Audio Coding (AAC)*. (MPEG-2)

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

[3] *ISO/IEC 14496-3:2001/Amd1, Bandwidth Extension (MPEG-4)*

[4] *ISO/IEC 14496-3:2001/Amd2, Parametric Audio for High Quality Audio (MPEG-4)*

[5] *Coding Technologies' aacPlus Fixed Point Firmware Reference Decoder v4.0.3*