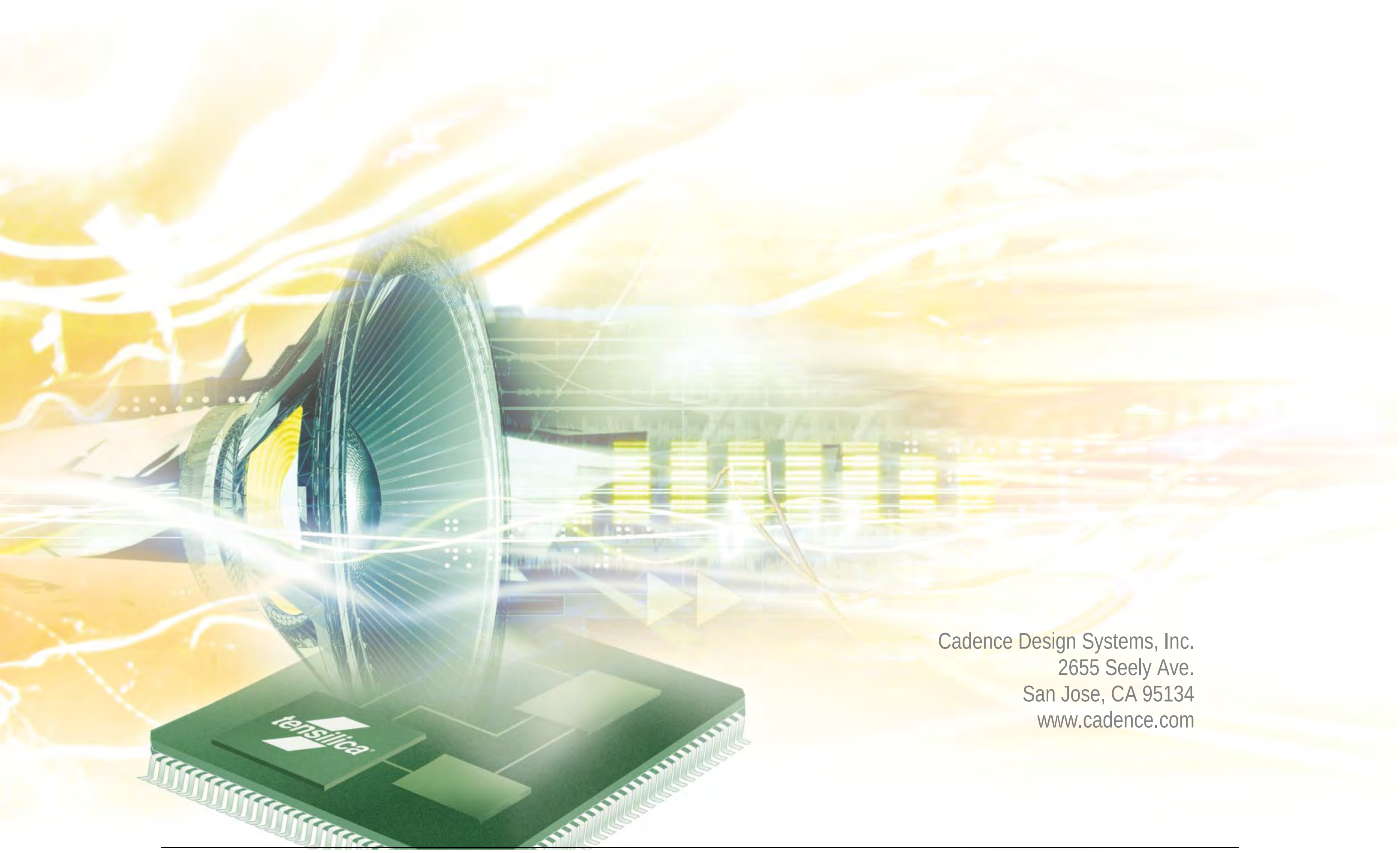
Sample Rate Converter

Programmer's Guide

For HiFi DSPs



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

For further assistance, contact Cadence Online Support at https://support.cadence.com/.

Version: 1.10
Last Updated: June 2023

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

Contents

Figures

Tables

Document Change History

Version	Changes

1.2	■ Introduced History section.
	■ Changed generic references of HiFi 2 to HiFi.
	■ Deleted references to Diamond 330HiFi.
	■ Added memory and timing data for HiFi Mini and HiFi 3.
	■ Changed input and output PCM data alignment from right justified to left justified.
	■ Removed error related to XA_SRC_PP_CONFIG_FATAL_INVALID_NUM_STAGES
	■ Added three new error codes:
	  ■ XA_SRC_PP_EXECUTE_NON_FATAL_INVALID_CONFIG_SEQ
	  ■ XA_SRC_PP_EXECUTE_NON_FATAL_INVALID_API_SEQ
	  ■ XA_SRC_PP_EXECUTE_FATAL_ERR_EXECUTE
	■ Added new Section 4.3, Customizing the Library

1.3	■ Added new API XA_SRC_PP_CONFIG_PARAM_BYTES_PER_SAMPLE and related command line option pcmwidth. ■ Updated memory and MCPS numbers.
1.4	■ Added new SET_CONFIG API XA_SRC_PP_CONFIG_PARAM_ENABLE_ASRC and related command line option enable_asrc ■ Added new SET_CONFIG API XA_SRC_PP_CONFIG_PARAM_DRIFT_ASRC and related command line option drift_asrc ■ Added new SET_CONFIG API XA_SRC_PP_CONFIG_PARAM_ENABLE_CUBIC and related command line option enable_cubic ■ Added new GET_CONFIG API XA_SRC_PP_CONFIG_PARAM_GET_DRIFT_FRACT_ASRC ■ Updated the memory and MCPS performance numbers, only HiFi 3 numbers are listed.

1.4	■ Added three new error codes:
	  ■ XA_SRC_PP_CONFIG_NON_FATAL_INVALID_ENABLE_ASRC
	  ■ XA_SRC_PP_CONFIG_NON_FATAL_INVALID_DRIFT_ASRC
	  ■ XA_SRC_PP_CONFIG_NON_FATAL_INVALID_ENABLE_CUBIC
	■ Removed SET_CONFIG and command-line items related to custom mode.
	■ Updated Section 4.3 custom mode configuration information.

1.6	■ Updated performance data in Section 1.4.
1.7	■ Updated the memory and MCPS performance numbers for HiFi 1 core
1.8	■ Updated the memory and MCPS performance numbers for Fusion F1, HiFi 1, HiFi 3, HiFi 3Z, HiFi 4, and HiFi 5 cores.
1.9	■ Updated description on custom library generation in section 4.3 ■ Deprecated support for FRIO mode and custom filter coefficients
1.10	■ Updated memory and MCPS performance numbers in section 1.4. ■ Updated usage notes in section 3.3.

1. Introduction to the HiFi SRC

The HiFi Sample Rate Converter (SRC) is designed for high quality sample rate conversions for audio and speech applications. HiFi SRC supports input and output sampled at all the standard audio sample rates from 8 kHz to 192 kHz. It performs the sample rate conversion on the input signals using one or more sets of linear-phase FIR filters and hence does not introduce any phase distortion during this conversion. The SRC filters used for integer conversion ratios have 100 dB or more stop band attenuation. While the SRC filters used for non-integer conversion ratios have about 80 dB stop band attenuation. Examples of integer conversion ratios are 1/3, 2/3, 3/1, 3/2, 1/4, etc.; and examples of non-integer conversion ratios are 32/44.1, 44.1/48, 48/88.2, etc.

1.1 SRC Description

The internal architecture of the SRC module is based on processing the input signal (or input sample buffers) through a cascade of basic SRC conversion blocks. Any sample-rate conversion ratio can be considered as a product of integer conversion ratio(s) and a non-integer (fractional) conversion ratio. In other words, any SRC ratio can be expressed as,

$$\text{fs\_ratio} = \left\{\prod_i \text{fs\_int\_ratio (i)}\right\} * \text{fs\_frac\_ratio}.$$

Where, fs_frac_ratio is a non-integer value between 1/2 and 3/2 and fs_int_ratio(i) can be any value from the basic ratios set = {1/2, 3/2, 3/1, 2/1, 2/3, 1/3, 3/4, 4/3 }.

In other words, any arbitrary SRC conversion ratio can be implemented as a series of one or more interpolation or decimation stages (each implementing an integer conversion ratio from the basic integer conversion ratio set) and an optional non-integer (fractional) stage. Each SRC stage for implementing the basic integer conversion ratio processes the input using an up sampler followed by an anti-imaging filter (or anti-aliasing filter followed by down sampler). The SRC stage designed for fractional conversion ratio uses a polyphase filter bank. The filter bank enables upsampling the input signal by an oversampling factor of 32 or more. And the final output is calculated by downsampling and linear-interpolation of this oversampled signal based on the fractional conversion ratio [1]. The following is a brief overview of the basic integer conversion ratios based on interpolation and decimation and a polyphase filter bank based fractional converter [2, 3].

1.1.1 Interpolation (Upsampling the Input Signal by a Factor of L)

The SRC block increases the input sample rate by a factor of L. This is achieved by inserting L-1 uniformly spaced zero value samples between every two consecutive input samples, followed by an anti-imaging filter to remove spectral images created by insertion of zeros.

Let x(n) be the input sequence and v(n) be the sequence with L-1 zeros inserted. Let y(n) denote the final output sequence after filtering the signal through an anti-imaging filter with coefficients h(0), h(1)…h(M-1) }. Then,

$$y(n) = \sum_{i=0}^{M-1} h(i)v(n-i)$$

Since L-1 zeros were inserted in the sequence x(n) to get v(n), thus, v(n-i) = 0, unless n-i is a multiple of L, the interpolation factor. Thus, the upsampling operation can be simplified to

$$y(mL+k) = \sum_{i'=0}^{M/L-1} h(i'L+k)x(m-i') = \sum_{i=0}^{M-1} h(i)v(mL+k-i)$$

Note The preceding equation is derived based on the fact that v(.) is non-zero only at k-i = i'*L. Therefore, each output of the interpolation is generated by processing the input signal using different sets of M/L filter coefficients of the anti-imaging filter.

1.1.2 Decimation (Downsampling the Input Signal by a Factor of D)

Decimation is the reduction in the sample rate by a factor D, which is achieved by keeping only every Dth sample and discarding the remaining D-1 samples. This causes the higher (f > 1/2D*fs_in) frequencies in the input signal to appear as aliased frequencies in the output. To avoid this, an anti-aliasing FIR filter must be applied to the input signal before the decimation process.

If x(n) is the input signal and h(0), h(1)…h(M-1) are the coefficients of the anti-aliasing low pass filter, the FIR filter output is given by:

$$v(n) = \sum_{i=0}^{M-1} h(i)x(n-i)$$

If y(n) is the final decimated output signal, and y(n) = v(nD), where D is the decimation factor, then the combination of downsampling and anti-aliasing filtering can be represented in the following decimation equation:

$$y(n) = \sum_{i=0}^{M-1} h(i)x(nD - i)$$

In other words, every decimated output point is computed using the full set of filter coefficients.

1.1.3Fractional Conversion using a Polyphase Filter

When the sample rates of the input and output signals are such that their ratios are non-integer (fractional) ratios, (for example, 44.1 and 48 or 32 and 44.1, etc.), a polyphase filter bank is used to perform the sample rate conversion. An oversampled version of the input signal is computed using a very long FIR filter. The oversampling factor is typically 32 and a typical length of the filter used is 1024.

A polyphase filter bank decomposition method allows decomposing of this large FIR filter of length M into a smaller set of filters of length K=M/I, where M is selected to be a multiple of I (where I is the oversampling factor). Typical values of M and I are 1024 and 32, respectively.

Let us denote the interpolation filter as h(.) and corresponding sets of polyphase filter coefficients as follows:

$p_k(i) = h(k + i*l), \qquad k = 0, 1…I-1 \;\&\; i = 0, 1…M/I - 1$

If x(n) is the input signal, then one can use the above filter to generate an interim oversampled version of the input signal { x_ov (s) } as follows:

$$x\_ov(s) = \sum_{i=0}^{M/I-1} pk(i) * x(s/I - i)$$

where k = s % I.

Now, consider an SRC setting, where the task is to perform a sample rate conversion of ratio 'r', where r is a non-integer (fractional) ratio. Let {x (n)} denote the input sequence to the sample rate converter. If Tx is the input sampling period, then the SRC module is expected to generate a sequence of output samples that represent signal sampled at Tx/r time interval. This is done by "resampling" the interim oversampled signal { x_ov(s) } corresponding to the input signal { x(n) }.

Let I represent the oversampling factor used for generating the interim oversampled signal. The interim oversampled signal can be generated by processing the input signal {x(n)} through the above mentioned polyphase filter bank. The oversampled intermediate signal has output samples that are placed at Tx/I time interval. In other words, the intermediate oversampled signal is a signal that has samples at time instances n*(k/I)*Tx (for all k=0 to I-1).

However, for the final output of SRC, we need to generate output samples at time instances that are integral multiples of Tx/r (output sampling interval). Specifically, the desired output time instances where the output sample must be generated will be:

Toutput_m = (m* Tx/r).

These final output samples at the above time instances are generated by a linear interpolation of the samples of the interim oversampled signal. We optimize the cycles required for generating that interim oversampled signal using the fact that the final output samples are only required to be computed at time instances, m*Tx/r. This is done as follows.

1. Select a pair of samples (from the interim oversampled signal) around the desired output time instance from the interpolated outputs of the polyphase filter bank. Perform the computations to generate only these relevant two samples of the oversampled signal.
2. Next, linearly interpolate these two samples to compute the final output, which is exactly at the desired output time instance (i.e. mTx/r). This is done using the following equations.

   Without loss of generality, we can express the fractional part of m/r as:

   $$Fract\ (m/r) = k/I + \beta_m$$

   Where k and I are positive integers and β_m is a fraction in the range $0 \le \beta \le 1/I$.

   Also note *that* $k/I \le Fract(m/r) < (k+1)/I$.

Let $x\_ov_k(m)$ & $x\_ov_{(k+1)}(m)$ represent polyphase filter bank output samples of k^{th} and $(k+1)^{th}$ phases. These two samples represent oversampled outputs at time instances (k/I)*Tx and ((k+1)/I)*Tx.

Then the final output at the desired output sampling instance (m*Tx/r) is approximated as:

$$y(m) = (1 - \alpha_m)\ x\_ov_k(m) + \alpha_m\ x\_ov_{(k+1)}(m)$$

and

$$\alpha_m = I * (Fract(m/r) - k/I) = I * \beta_m$$

In summary, to calculate the output at m*Tx/r, we compute the k^{th} and $(k+1)^{th}$ polyphase filter bank outputs, and linearly interpolate them with α_m to generate the final output sample.

The same fraction value (α_m) can be used to generate output with the “cubic” interpolation method, which is described in section 1.1.4.

1.1.4Cubic Interpolation for a Polyphase Filter

Cubic interpolation aims for better quality in terms of THD. The current implementation uses the same polyphase coefficient tables that are used for linear implementation.

For every sample, the coefficient values are interpolated using the cubic interpolation method and then the resulting coefficient set is used for filtering.

The following equations describe the cubic interpolation of the filter coefficients for the current phase. Suppose:

- coeff0 is the filter coefficient set for phase s
- coeff1 is the filter coefficient set for phase s+1
- coeff2 is the filter coefficient set for phase s+2
- coeff3 is the filter coefficient set for phase s+3

Where "s" is the index used for oversampled signal.

Let f = phase_frac, which is equivalent to $\alpha_{m,}$ (the fractional distance mentioned in the earlier section), then:

- w0 =-f^3/6 + f^2/2 - f/3;
- w1 = f^3/2 - f^2/1 - f/2+1;
- w2 =-f^3/2 + f^2/2 + f;
- w3 = f^3/6 - f/6;

And

final_coeff = (w0 * coeff0) + (w1 * coeff1) + (w2 * coeff2) + (w3 * coeff3).

Cubic interpolation improves THD performance:

- With 32 taps per phase, THD of 85 dB is achieved for all the frequencies up to 0.45 of the sampling frequency. With linear interpolation, THD drops with increasing frequency.
- With 40 taps per phase, THD is close to100 dB.

Compared to linear interpolation, cubic interpolation requires significantly more MCPS. However, for a higher number of channels, the MCPS requirements are comparable with that of linear interpolation.

1.1.5Asynchronous SRC (ASRC) using a Polyphase Filter

The HiFi ASRC processes drift values in the range of -4% to 4% with a 1ppm (0.000001) step. The library accepts the drift value from the application and applies it to the current frame. The specified drift value is applied to every output sample produced. The application is expected to calculate this value based on the actual drift in the two clocks and feed it to the ASRC module in the library.

Suppose:

- TS_Out: Time period of the Output signal.
- TS_In: Time period of the Input signal.

  Δts = TS_Out - TS_In
  i.e., it is the difference in the time period of the Input signal and Output signal.

Then:

  Drift = Δts / TS_Out;

Design Details

The SRC polyphase (PP) kernel is modified to be used as the core ASRC block.

The phase calculation logic in the polyphase filter handles the drift value provided by the application. The ASRC module accordingly skips or accepts extra input samples for the output computations to adjust the timings associated with the specified drift value.

Following are other details for this design:

- Polyphase coefficients designed for SRC module are reused for the ASRC mode.
- For fractional ratios, a single polyphase block is used, which serves two purposes: fractional ratio conversion and drift adjustment.
- There are two interpolation options: linear interpolation and cubic interpolation; the latter option gives better quality at higher MCPS.

Note The ASRC mode can only be enabled at initialization time. Once ASRC is enabled, the drift value can be changed at runtime for every frame.

1.2 Document Overview

This document covers all the information required to integrate the HiFi SRC into an application. The HiFi libraries implement a simple API to encapsulate the complexities of the operations and simplify the application and system implementation. Parts of the API that are common to all the HiFi codecs are described after the introduction. The next section covers all the features and information particular to the HiFi SRC. Finally, the example test bench is described.

1.3 HiFi SRC Specifications

The HiFi Audio SRC implements the following features.

- Sample rates: 4, 6, 8, 11.025, 12, 16, 22.05, 24, 32, 44.1, 48, 64, 88.2, 96, 128, 176.4, 192, 352.8, 384 kHz.
- Flexible architecture to implement sample rate conversion from any input to output sample rates with the appropriate use of built-in filters.
- Flexible architecture where library can be configured based on customer specified features such as ASRC support, Cubic Interpolation support, Multichannel support, and various sample rate conversions.
- Linear phase digital filtering.
- Multi-channel support: Currently supports up to 24 channels, can be extended to any number of channels. Both the input and output channels are in interleaved format.
- Stopband attenuation: 100 dB for integer conversions and 80 dB for fractional conversions, such as 32 <--> 44.1, 48<-->44.1, etc.
- Passband ripple: 0.2 dB.
- Passband gain: Less than 1 dB.
- The transition band allows frequencies up to 0.81 to 0.9 times of the full bandwidth of the output signal to be passed without distortion.
- Input chunk size: from 4 to 512 samples.
- Input alignment: 4 bytes.
- Bypass mode when the input and output rates are the same (except in ASRC mode).

- Input and output data format: 16-bit or 24-bit PCM. The 16-bit PCM data occupies a 16-bit word. The 24-bit PCM data occupies a 32-bit word, left justified (the PCM data is in the 24 MSB section of the 32-bit word). The input and output data have the same width.
- Option of using cubic interpolation for polyphase filters at initialization time.
  - Better quality (THD more than 80 dB and it is consistent across various input frequencies.)

1.3.1ASRC Features

ASRC features include SRC features listed in section 1.3, in addition to the following: ASRC is enabled at initialization time.

- ASRC drift range: -0.04 to 0.04 (with step size of 0.000001, or 1 ppm) of every output sample.
- ASRC modes:
  - Low computation mode. In this mode, it uses linear interpolation (with THD values 80+ dB).
  - High quality mode. In this mode, cubic interpolation is used to achieve better THD values across all the fractional sampling frequencies ratios. This results in the cost of extra computation load for the sample rate conversion.

1.4 HiFi SRC Performance

The Sample Rate Converter (SRC) was characterized on the HiFi 5-stage DSP. The memory usage and performance figures are provided for design reference.

- The API structure size returned by XA_API_CMD_GET_API_SIZE is approximately 2190 bytes for SRC.
- The memory table structure size returned by XA_API_CMD_GET_MEMTABS_SIZE is approximately 80 bytes.

1.4.1Memory

Library Memory Usage

Text (Kbytes)					Data
HiFi 1	HiFi 3	HiFi 3z	HiFi 4	HiFi 5	Kbytes
57.44	33.37	37.34	37.92	86.84	22.7

SRC Runtime Memory

PCM Width	Interpolation	Conversion Rate (Hz)		SRC Runtime Memory (bytes)				
		Input	Output	Persistent	Scratch	Stack	Input	Output
16	linear	8000	44100	640	21312	384	1024	5728
16	linear	16000	44100	528	12736	384	1024	2880
16	linear	44100	44100	32	4096	384	1024	1024
16	linear	48000	44100	800	8256	384	1024	992
16	cubic	48000	44100	800	16448	384	1024	992
24	linear	8000	48000	368	16384	384	2048	12288
24	linear	16000	48000	208	8192	384	2048	6144
24	linear	44100	48000	288	8448	384	2048	2304
24	cubic	44100	48000	288	16640	384	2048	2304
24	linear	48000	48000	32	4096	384	2048	2048
24	linear	96000	48000	464	3072	384	2048	1024
24	linear	192000	48000	656	3072	384	2048	512
24	linear	48000	96000	256	6144	384	2048	4096
24	cubic	48000	96000	256	6144	384	2048	4096
24	linear	96000	96000	32	4096	384	2048	2048
24	cubic	96000	96000	32	4096	384	2048	2048
24	linear	192000	96000	464	3072	384	2048	1024
24	cubic	192000	96000	464	3072	384	2048	1024
24	linear	48000	192000	368	12288	384	2048	8192
24	cubic	48000	192000	368	12288	384	2048	8192
24	linear	96000	192000	256	6144	384	2048	4096
24	cubic	96000	192000	256	6144	384	2048	4096
24	linear	192000	192000	32	4096	384	2048	2048
24	cubic	192000	192000	32	4096	384	2048	2048

ASRC Runtime Memory

PCM Width	Interpolation	Conversion Rate (Hz)		SRC Runtime Memory (bytes)				
		Input	Output	Persistent	Scratch	Stack	Input	Output
16	linear	8000	44100	640	23680	384	1024	6954
16	linear	16000	44100	528	13440	384	1024	3326
16	linear	44100	44100	288	8384	384	1024	1176
16	linear	48000	44100	800	8448	384	1024	1142
16	cubic	48000	44100	800	8448	384	1024	1142
24	linear	8000	48000	640	23232	384	2048	15252
24	linear	16000	48000	496	13248	384	2048	7256
24	linear	44100	48000	288	8576	384	2048	2552
24	cubic	44100	48000	288	8576	384	2048	2552
24	linear	48000	48000	288	8384	384	2048	2352
24	linear	96000	48000	752	7232	384	2048	1276
24	linear	192000	48000	944	7232	384	2048	672
24	linear	48000	96000	528	10944	384	2048	4836
24	cubic	48000	96000	528	10944	384	2048	4836
24	linear	96000	96000	288	8384	384	2048	2352
24	cubic	96000	96000	288	8384	384	2048	2352
24	linear	192000	96000	752	7232	384	2048	1276
24	cubic	192000	96000	752	7232	384	2048	1276
24	linear	48000	192000	640	18432	384	2048	10212
24	cubic	48000	192000	640	18432	384	2048	10212
24	linear	96000	192000	528	10944	384	2048	4836
24	cubic	96000	192000	528	10944	384	2048	4836
24	linear	192000	192000	288	8384	384	2048	2352
24	cubic	192000	192000	288	8384	384	2048	2352

The above memory requirements are for single channel. For the multichannel audio cases,

1. The run-time memory requirements (except stack memory), grow linearly with the increase in input number of channels.
2. Persistent buffer requirements in some conversion ratios, can be slightly higher due to optimization for performance.
3. Scratch buffer size for most of the conversion ratios has been reduced due to memory optimizations in specific kernels.
4. Stack memory requirements remain the same irrespective of the number of input channels.
5. Recommended to use the SRC testbench application in the release package to find the accurate run-time memory requirements for the required number of channels of interest.

Note The input chunk size is assumed to be 512 samples.

1.4.2 Timings

SRC Timings

Table 1-1 SRC Timings for HiFi 1

PCM Width	Interpolation	Rate (kHz)		Average CPU Load (MHz)			
		In	Out	1ch	2ch	6ch	8ch
16	linear	8	44.1	2.4	4.7	15	19.7
16	linear	16	44.1	3	5.9	19.4	25.5
16	linear	44.1	44.1	0	0.1	0.1	0.2
16	linear	48	44.1	5.3	10.4	35.9	46.3
16	cubic	48	44.1	13.3	17.8	46.2	57.3
24	linear	8	48	1.3	2.8	8.2	11
24	linear	16	48	1.8	3.6	10.9	14.6
24	linear	44.1	48	2.6	5	19.8	24.6
24	cubic	44.1	48	11.3	13	31	36.6
24	linear	48	48	0.1	0.1	0.3	0.4
24	linear	96	48	2.7	5.7	17.1	22.7
24	linear	192	48	3.7	8.1	24.1	32.2
24	linear	48	96	2.4	5.2	15.4	20.6
24	cubic	48	96	2.4	5.2	15.4	20.6
24	linear	96	96	0.1	0.2	0.6	0.8
24	cubic	96	96	0.1	0.2	0.6	0.8
24	linear	192	96	5.3	11.4	34.1	45.5
24	cubic	192	96	5.3	11.4	34.1	45.5
24	linear	48	192	4.1	8.5	25.5	34.1
24	cubic	48	192	4.1	8.5	25.5	34.1
24	linear	96	192	9.1	18.8	56.3	75.1
24	cubic	96	192	9.1	18.8	56.3	75.1
24	linear	192	192	0.3	0.4	1.2	1.6
24	cubic	192	192	0.3	0.4	1.2	1.6

Table 1-2 SRC Timings for HiFi 3

PCM Width	Interpolation	Rate (kHz)		Average CPU Load (MHz)			
		In	Out	1ch	2ch	6ch	8ch
16	linear	8	44.1	1.3	2.1	6.5	8.1
16	linear	16	44.1	2.4	3.7	11.3	13.9
16	linear	44.1	44.1	0	0.1	0.1	0.2
16	linear	48	44.1	5.8	9.2	28.6	35.9
16	cubic	48	44.1	12.8	15.5	37.7	44.9
24	linear	8	48	0.5	1.1	3.4	4.5
24	linear	16	48	0.6	1.4	4.1	5.5
24	linear	44.1	48	3.8	5.1	16.3	19.2

PCM Width	Interpolation	Rate (kHz)		Average CPU Load (MHz)			
		In	Out	1ch	2ch	6ch	8ch
24	cubic	44.1	48	11.5	12	26.2	29
24	linear	48	48	0.1	0.1	0.3	0.4
24	linear	96	48	1.9	3.9	11.7	15.6
24	linear	192	48	2.9	6	18	24
24	linear	48	96	1.6	3.4	10.1	13.4
24	cubic	48	96	1.6	3.4	10.1	13.4
24	linear	96	96	0.1	0.2	0.6	0.8
24	cubic	96	96	0.1	0.2	0.6	0.8
24	linear	192	96	3.7	7.8	23.4	31.2
24	cubic	192	96	3.7	7.8	23.4	31.2
24	linear	48	192	2.2	4.6	13.8	18.5
24	cubic	48	192	2.2	4.6	13.8	18.5
24	linear	96	192	3.2	6.7	20.1	26.8
24	cubic	96	192	3.2	6.7	20.1	26.8
24	linear	192	192	0.3	0.4	1.2	1.6
24	cubic	192	192	0.3	0.4	1.2	1.6

Table 1-3 SRC Timings for HiFi 3z

PCM Width	Interpolation	Rate (kHz)		Average CPU Load (MHz)			
		In	Out	1ch	2ch	6ch	8ch
16	linear	8	44.1	1.1	1.9	6	7.6
16	linear	16	44.1	1.9	3.2	10.4	13
16	linear	44.1	44.1	0	0.1	0.1	0.2
16	linear	48	44.1	4.1	7.2	23.3	29.5
16	cubic	48	44.1	10.4	12.7	31.4	37.6
24	linear	8	48	0.5	1.1	3.3	4.4
24	linear	16	48	0.6	1.3	4	5.4
24	linear	44.1	48	2.7	4.3	14.7	17.8
24	cubic	44.1	48	9.6	10.3	23.4	26.6
24	linear	48	48	0.1	0.1	0.3	0.4
24	linear	96	48	1.7	3.7	10.9	14.5
24	linear	192	48	2.6	5.6	16.6	22.1
24	linear	48	96	1.6	3.3	9.8	13.1
24	cubic	48	96	1.6	3.3	9.8	13.1
24	linear	96	96	0.1	0.2	0.6	0.8
24	cubic	96	96	0.1	0.2	0.6	0.8
24	linear	192	96	3.5	7.3	21.8	29.1
24	cubic	192	96	3.5	7.3	21.8	29.1
24	linear	48	192	2.1	4.5	13.6	18.1
24	cubic	48	192	2.1	4.5	13.6	18.1

PCM Width	Interpolation	Rate (kHz)		Average CPU Load (MHz)			
		In	Out	1ch	2ch	6ch	8ch
24	linear	96	192	3.1	6.6	19.6	26.2
24	cubic	96	192	3.1	6.6	19.6	26.2
24	linear	192	192	0.3	0.4	1.2	1.6
24	cubic	192	192	0.3	0.4	1.2	1.6

Table 1-4 SRC Timings for HiFi 4

PCM Width	Interpolation	Rate (kHz)		Average CPU Load (MHz)			
		In	Out	1ch	2ch	6ch	8ch
16	linear	8	44.1	1	1.7	5.4	7
16	linear	16	44.1	1.8	3	9.4	12
16	linear	44.1	44.1	0	0.1	0.1	0.2
16	linear	48	44.1	4.7	7.9	24.9	32.2
16	cubic	48	44.1	9.7	12.3	31.3	38.4
24	linear	8	48	0.5	1	3.1	4.2
24	linear	16	48	0.6	1.2	3.6	4.8
24	linear	44.1	48	2.7	3.8	12.6	15.8
24	cubic	44.1	48	8.2	8.6	19.7	22.6
24	linear	48	48	0.1	0.1	0.3	0.4
24	linear	96	48	1.7	3.6	10.8	14.4
24	linear	192	48	2.6	5.4	16.2	21.6
24	linear	48	96	1.5	3.2	9.5	12.7
24	cubic	48	96	1.5	3.2	9.5	12.7
24	linear	96	96	0.1	0.2	0.6	0.8
24	cubic	96	96	0.1	0.2	0.6	0.8
24	linear	192	96	3.4	7.2	21.6	28.7
24	cubic	192	96	3.4	7.2	21.6	28.7
24	linear	48	192	2	4.3	12.7	17
24	cubic	48	192	2	4.3	12.7	17
24	linear	96	192	3	6.4	19.1	25.4
24	cubic	96	192	3	6.4	19.1	25.4
24	linear	192	192	0.3	0.4	1.2	1.6
24	cubic	192	192	0.3	0.4	1.2	1.6

Table 1-5 SRC Timings for HiFi 5

PCM Width	Interpolation	Rate (kHz)		Average CPU Load (MHz)			
		In	Out	1ch	2ch	6ch	8ch
16	linear	8	44.1	0.9	1.5	4.2	5.1
16	linear	16	44.1	1.6	2.7	7.6	8.9
16	linear	44.1	44.1	0	0.1	0.1	0.2
16	linear	48	44.1	2.6	4.6	17.5	21.6

PCM Width	Interpolation	Rate (kHz)		Average CPU Load (MHz)			
		In	Out	1ch	2ch	6ch	8ch
16	cubic	48	44.1	6.1	7.5	19.4	24.2
24	linear	8	48	0.4	0.8	2.4	3.2
24	linear	16	48	0.4	0.9	2.5	3.4
24	linear	44.1	48	1.7	2.9	12.5	14.7
24	cubic	44.1	48	5.6	6	14.5	17.5
24	linear	48	48	0.1	0.1	0.3	0.4
24	linear	96	48	1	2.3	6.8	9
24	linear	192	48	1.7	3.7	11	14.5
24	linear	48	96	1	2.3	6.8	9
24	cubic	48	96	1	2.3	6.8	9
24	linear	96	96	0.1	0.2	0.6	0.8
24	cubic	96	96	0.1	0.2	0.6	0.8
24	linear	192	96	2.1	4.5	13.5	17.9
24	cubic	192	96	2.1	4.5	13.5	17.9
24	linear	48	192	1.4	3	9.1	12.1
24	cubic	48	192	1.4	3	9.1	12.1
24	linear	96	192	2	4.4	13	17.3
24	cubic	96	192	2	4.4	13	17.3
24	linear	192	192	0.3	0.4	1.2	1.6
24	cubic	192	192	0.3	0.4	1.2	1.6

ASRC Timings

Table 1-6 ASRC Timings for HiFi 1

PCM Width	Interpolation	Rate (kHz)		Average CPU Load (MHz)			
		In	Out	1ch	2ch	6ch	8ch
16	linear	8	44.1	2.7	5.6	16.7	22.2
16	linear	16	44.1	4.7	9.5	28.5	38
16	linear	44.1	44.1	2.8	5.8	17.2	23
16	linear	48	44.1	6.6	13.5	40.3	53.8
16	cubic	48	44.1	11.8	23.7	71.1	94.8
24	linear	8	48	1.9	4	11.9	15.8
24	linear	16	48	3.4	6.8	20.5	27.3
24	linear	44.1	48	4	8.3	24.8	33.1
24	cubic	44.1	48	9.6	19.5	58.4	77.7
24	linear	48	48	3.1	6.3	19	25.3
24	linear	96	48	5.6	11.6	34.7	46.4
24	linear	192	48	6.7	14	41.8	56.1
24	linear	48	96	5.5	11.2	33.5	44.6
24	cubic	48	96	11.1	22.4	67.3	89.7
24	linear	96	96	6.1	12.7	38	50.5
24	cubic	96	96	17.4	35.2	105.6	140.8

PCM Width	Interpolation	Rate (kHz)		Average CPU Load (MHz)			
		In	Out	1ch	2ch	6ch	8ch
24	linear	192	96	11.2	23.2	69.5	92.7
24	cubic	192	96	22.5	45.8	136.9	183
24	linear	48	192	6.4	13.2	39.5	52.6
24	cubic	48	192	12	24.4	73.3	97.7
24	linear	96	192	10.9	22.3	67	89.2
24	cubic	96	192	22.2	44.9	134.7	179.4
24	linear	192	192	12.2	25.3	75.9	101.1
24	cubic	192	192	34.8	70.4	211.2	281.5

Table 1-7 ASRC Timings for HiFi 3

PCM Width	Interpolation	Rate (kHz)		Average CPU Load (MHz)			
		In	Out	1ch	2ch	6ch	8ch
16	linear	8	44.1	2.2	4.5	13.4	17.9
16	linear	16	44.1	4.1	8.4	25.1	33.4
16	linear	44.1	44.1	4.1	8.3	24.8	33
16	linear	48	44.1	9.2	18.5	55.3	73.8
16	cubic	48	44.1	13	26.1	78.2	104.3
24	linear	8	48	1.3	2.6	7.8	10.4
24	linear	16	48	2.1	4.3	12.9	17.1
24	linear	44.1	48	7.5	15.2	45.5	60.6
24	cubic	44.1	48	11.7	23.5	70.5	93.9
24	linear	48	48	4.4	9	27	36
24	linear	96	48	6.2	12.6	37.7	50.3
24	linear	192	48	7.2	14.8	44	59.1
24	linear	48	96	6	12.1	36.3	48.3
24	cubic	48	96	10.1	20.3	61	81.2
24	linear	96	96	8.9	18	54.1	72
24	cubic	96	96	17.1	34.5	103.4	137.7
24	linear	192	96	12.4	25.2	75.4	100.7
24	cubic	192	96	20.7	41.7	124.6	166.6
24	linear	48	192	6.6	13.4	40.2	53.5
24	cubic	48	192	10.7	21.6	64.9	86.4
24	linear	96	192	11.9	24.2	72.6	96.7
24	cubic	96	192	20.2	40.6	122	162.4
24	linear	192	192	17.8	36	108.1	144
24	cubic	192	192	34.3	68.9	206.8	275.5

Table 1-8 ASRC Timings for HiFi 3z

PCM Width	Interpolation	Rate (kHz)		Average CPU Load (MHz)			
		In	Out	1ch	2ch	6ch	8ch
16	linear	8	44.1	1.5	3.1	9.3	12.4

PCM Width	Interpolation	Rate (kHz)		Average CPU Load (MHz)			
		In	Out	1ch	2ch	6ch	8ch
16	linear	16	44.1	2.8	5.6	16.8	22.5
16	linear	44.1	44.1	2.9	6	17.8	23.8
16	linear	48	44.1	5.9	11.9	35.6	47.4
16	cubic	48	44.1	9.5	19.2	57.5	76.7
24	linear	8	48	1	2.2	6.5	8.6
24	linear	16	48	1.7	3.4	10.3	13.7
24	linear	44.1	48	4.6	9.4	28.2	37.5
24	cubic	44.1	48	8.6	17.4	52.1	69.4
24	linear	48	48	3.2	6.5	19.7	26.2
24	linear	96	48	4.8	9.8	29.5	39.4
24	linear	192	48	5.8	11.8	35.2	47.3
24	linear	48	96	4.7	9.6	28.7	38.2
24	cubic	48	96	8.4	17	50.9	67.8
24	linear	96	96	6.4	13.1	39.3	52.3
24	cubic	96	96	13.9	27.9	83.7	111.6
24	linear	192	96	9.7	19.7	59	78.7
24	cubic	192	96	17.2	34.6	103.3	138
24	linear	48	192	5.3	10.8	32.6	43.4
24	cubic	48	192	9	18.3	54.8	73
24	linear	96	192	9.4	19.1	57.4	76.4
24	cubic	96	192	16.8	33.9	101.8	135.6
24	linear	192	192	12.9	26.2	78.6	104.7
24	cubic	192	192	27.7	55.8	167.5	223.1

Table 1-9 ASRC Timings for HiFi 4

PCM Width	Interpolation	Rate (kHz)		Average CPU Load (MHz)			
		In	Out	1ch	2ch	6ch	8ch
16	linear	8	44.1	1.5	3.1	9.4	12.6
16	linear	16	44.1	2.9	5.8	17.4	23.1
16	linear	44.1	44.1	2.8	5.7	17	22.7
16	linear	48	44.1	6.6	13.4	40.2	53.2
16	cubic	48	44.1	8.7	17.4	52.2	69.6
24	linear	8	48	1	2	6.1	8.2
24	linear	16	48	1.6	3.2	9.6	12.8
24	linear	44.1	48	4.9	9.8	29.1	38.9
24	cubic	44.1	48	7	14.2	42.6	56.7
24	linear	48	48	3.1	6.2	18.7	24.9
24	linear	96	48	4.7	9.5	28.4	37.9
24	linear	192	48	5.6	11.4	33.9	45.5
24	linear	48	96	4.5	9.1	27.4	36.5
24	cubic	48	96	6.7	13.5	40.5	54
24	linear	96	96	6.1	12.5	37.4	49.8

PCM Width	Interpolation	Rate (kHz)		Average CPU Load (MHz)			
		In	Out	1ch	2ch	6ch	8ch
24	cubic	96	96	10.5	21.2	63.5	84.6
24	linear	192	96	9.3	19	56.9	75.9
24	cubic	192	96	13.7	27.7	82.9	110.8
24	linear	48	192	5	10.2	30.8	40.9
24	cubic	48	192	7.2	14.6	43.8	58.4
24	linear	96	192	9	18.3	54.9	73.1
24	cubic	96	192	13.3	27	81	107.9
24	linear	192	192	12.2	24.9	74.8	99.5
24	cubic	192	192	21	42.4	127.1	169.3

Table 1-10 ASRC Timings for HiFi 5

PCM Width	Interpolation	Rate (kHz)		Average CPU Load (MHz)			
		In	Out	1ch	2ch	6ch	8ch
16	linear	8	44.1	0.9	1.8	5.5	7.4
16	linear	16	44.1	1.6	3.3	10	13.4
16	linear	44.1	44.1	1.6	3.3	9.8	13.1
16	linear	48	44.1	3	6.1	18.3	24.5
16	cubic	48	44.1	5	10.1	30.2	40.2
24	linear	8	48	0.7	1.4	4.2	5.7
24	linear	16	48	1	2.1	6.3	8.3
24	linear	44.1	48	2.2	4.5	13.6	18
24	cubic	44.1	48	4.4	8.8	26.5	35.2
24	linear	48	48	1.8	3.6	10.9	14.5
24	linear	96	48	2.7	5.6	16.7	22.2
24	linear	192	48	3.4	7	21	28.1
24	linear	48	96	2.7	5.6	16.8	22.3
24	cubic	48	96	5.8	11.8	35.4	47.2
24	linear	96	96	3.5	7.3	21.8	29
24	cubic	96	96	9.7	19.7	59.1	78.7
24	linear	192	96	5.4	11.1	33.4	44.4
24	cubic	192	96	11.6	23.6	70.5	94.1
24	linear	48	192	3	6.2	18.8	25
24	cubic	48	192	6.1	12.5	37.4	49.8
24	linear	96	192	5.4	11.2	33.6	44.7
24	cubic	96	192	11.6	23.6	70.8	94.3
24	linear	192	192	7	14.5	43.6	58
24	cubic	192	192	19.5	39.4	118.1	157.3

Note The above table lists the MCPS numbers captured for few typical conversion rates, the input chunk size is 512, the drift value is 0.03 for ASRC timings.

Note Performance specification measurements are carried on a cycle-accurate simulator assuming an ideal memory system, *i.e.*, one with zero memory wait states. This is equivalent to running with all code and data in local memories or using an infinite-size, pre-filled cache model.

Note All the memory and MCPS numbers are captured with the XT-CLANG compiler and RI-2022.10 tools.

Note The MCPS numbers for HiFi 3/HiFi 3z/HiFi 4/HiFi 5 are obtained by running the test with binaries that are based on 24-bit optimized source code.

Note Optimization is done for HiFi 1. No Specific optimization is done for HiFi 3/ HiFi 3Z/ HiFi 4/ HiFi5.

Following is a summary of the impact on MCPS for various parameters.

- **Number of channels:** For non-fractional ratios, the MCPS numbers are approximately proportional to the channel count; for fractional ratios, the MHz/channel number decreases with the increasing channel count.
- **PCM width:** For a 16-bit input PCM signal, the MCPS numbers are close to or slightly lower than the MCPS numbers required for a 24-bit signal.
- **Chunk size:** The MCPS numbers are slightly higher for smaller chunk sizes.
- Cubic interpolation:
  - For a single channel, the MCPS numbers are higher than the MCPS numbers required for linear interpolation.
  - For a higher number of channels, due to multichannel processing, the MHz/channel number decrease with the increasing channel count.
- ASRC mode:
  - The MCPS numbers are proportional to the number of channels.
  - For different values of drift, there is a small change (less than +/- 5%) in MCPS.
  - The high quality ASRC (with cubic interpolation) MCPS numbers are higher than that of the medium quality ASRC (with linear interpolation).

2. Generic HiFi Audio Codec API

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

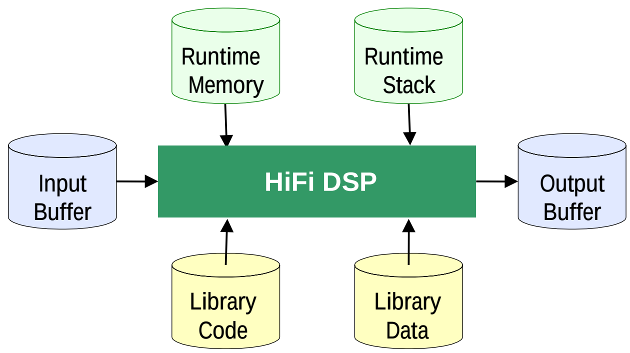


Figure 1 HiFi Audio Codec Interfaces

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

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

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

2.1 Memory Management

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

The 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

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

Output Buffer

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

2.2 C Language API

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

Table 2-1 Codec API

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

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

Parameters	▪ `p_xa_module_obj` Pointer to the opaque API structure. ▪ `i_cmd` Command. ▪ `i_idx` Command subtype or index. ▪ `pv_value` Pointer to the variable used to pass in, or get out properties from, the state structure.
Returns	Error code based on the success or failure of API command.

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

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

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

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

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

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

2.3 Generic API Errors

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

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

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

The class of an error can be API, Config, or Execution. The API errors are caused by 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 flowchart in Figure 2. For each stage of the flowchart 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.

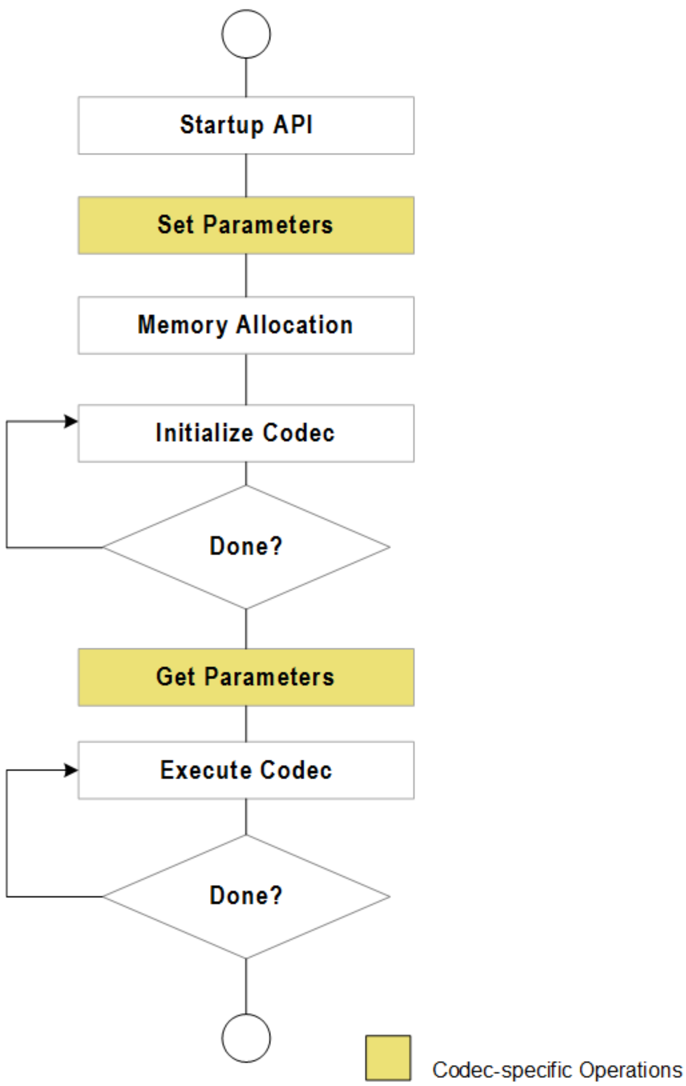


Figure 2 API Command Sequence Overview

2.4.1 Start-up API Stage

All the following commands should be run once 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 run 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 run in a loop to allocate the memory. The application first requests all the attributes of the memory block and then allocates it. Note that 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.

2.4.4Initialize Codec Stage

The following commands should be run in a loop during initialization. These 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, performs 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.

2.4.5Get Codec-Specific Parameters Stage

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

Table 2-7 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 run 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 was 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`	Run 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.

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 the block allocation

- `xa_type_def.h`

  All the types required for the API calls

2.6 HiFi API Command Reference

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

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

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

The errors returned by the API are detailed after each of the command definitions. However, there are a few errors that are common to all the API commands, which are listed in Section 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.

Since SRC is a post-processing module, the standard APIs related to configuration parameters `XA_CONFIG_PARAM_CUR_INPUT_STREAM_POS` and `XA_CONFIG_PARAM_GEN_INPUT_STREAM_POS` are not supported by the library.

2.6.1Common 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`)

2.6.2XA_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

2.6.3XA_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 to allocate memory for the API structure. The pointer to the API size variable is passed and the API returns the size of the structure in bytes. The API structure is used for the interface and is persistent.
Actual Parameters	▪ `p_xa_module_obj` **`NULL`** ▪ `i_cmd` `XA_API_CMD_GET_API_SIZE` ▪ `i_idx` **`NULL`** ▪ `pv_value` `&api_size` – Pointer to the API size variable.
Restrictions	The application shall allocate memory with an alignment of 4 bytes.

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

Example

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

Errors

- XA_API_FATAL_MEM_ALLOC

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

- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.4XA_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 the 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 the 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 the 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 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 the API structure.
	▪ `i_cmd`
	`XA_API_CMD_INIT`
	▪ `i_idx`
	`XA_CMD_TYPE_INIT_DONE_QUERY`
	▪ `pv_value`
	`&init_done` – Pointer to a flag that indicates the completion of initialization process.

Restrictions	None

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.5XA_API_CMD_GET_MEMTABS_SIZE

Table 2-17 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 the API structure. ▪ `i_cmd` `XA_API_CMD_GET_MEMTABS_SIZE` ▪ `i_idx` **`NULL`** ▪ `pv_value` `&proc_mem_tabs_size` – Pointer to the memory size variable.
Restrictions	The application shall allocate memory with an alignment of 4 bytes.

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.6 XA_API_CMD_SET_MEMTABS_PTR

Table 2-18 XA_API_CMD_SET_MEMTABS_PTR command

Subcommand	None
Description	This command is used to set the memory structure pointer in the library to the allocated value.
Actual Parameters	▪ `p_xa_module_obj` `api_obj` – Pointer to the 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 the 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 the API structure. ▪ `i_cmd` `XA_API_CMD_GET_MEM_INFO_SIZE` ▪ `i_idx` Index of the memory. ▪ `pv_value` `&size` – Pointer to the memory size.
Restrictions	None

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_API_FATAL_INVALID_CMD_TYPE

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

2.6.9 XA_API_CMD_GET_MEM_INFO_ALIGNMENT

Table 2-21 XA_API_CMD_GET_MEM_INFO_ALIGNMENT command

Subcommand	Memory index
Description	This command gets the alignment information of the memory-type being referred to by the index. The alignment required in bytes is returned to the application.
Actual Parameters	▪ `p_xa_module_obj` `api_obj` – Pointer to the 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 the API structure. ▪ `i_cmd` `XA_API_CMD_GET_MEM_INFO_TYPE` ▪ `i_idx` Index of the memory. ▪ `pv_value` `&type` – Pointer to the memory type variable.
Restrictions	None

Example

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

Table 2-23 Memory-Type Indices

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`
- XA_API_FATAL_INVALID_CMD_TYPE

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

2.6.11 XA_API_CMD_GET_MEM_INFO_PRIORITY

Table 2-24 XA_API_CMD_GET_MEM_INFO_PRIORITY command

Subcommand	Memory index
Description	This command gets the allocation priority of memory being referred to by the index. (The meaning of the levels is defined on a codec-specific basis. This command returns a fixed dummy value, unless the codec defines it otherwise.)
Actual Parameters	▪ `p_xa_module_obj` `api_obj` – Pointer to the 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 run this command during the main codec loop.
Actual Parameters	▪ `p_xa_module_obj` `api_obj` – Pointer to the 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 the 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 in both 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 the 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 the API structure. ▪ `i_cmd` `XA_API_CMD_GET_CURIDX_INPUT_BUF` ▪ `i_idx` **`NULL`** ▪ `pv_value` `&bytes_consumed` – Pointer to the bytes consumed variable.
Restrictions	None

Example

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

Errors

- Common API Errors
- XA_API_FATAL_MEM_ALLOC

  `pv_value` is `NULL`

2.6.16 XA_API_CMD_EXECUTE

Table 2-30 XA_CMD_TYPE_DO_EXECUTE subcommand

Subcommand	`XA_CMD_TYPE_DO_EXECUTE`
Description	This command runs the codec.

Actual Parameters	▪ `p_xa_module_obj`
	`api_obj` – Pointer to the 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 the 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 the API Structure.
	▪ `i_cmd`
	`XA_API_CMD_EXECUTE`
	▪ `i_idx`
	`XA_CMD_TYPE_DO_RUNTIME_INIT`
	▪ `pv_value`
	`NULL`

Restrictions	None

Example

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

Errors

- Common API Errors

2.6.17 XA_API_CMD_GET_OUTPUT_BYTES

Table 2-33 XA_API_CMD_GET_OUTPUT_BYTES command

Subcommand	None
Description	This command obtains the number of bytes output by the codec during the last execution.
Actual Parameters	▪ `p_xa_module_obj` `api_obj` – Pointer to the 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`

3. HiFi Audio Sample Rate Converter

The HiFi Sample Rate Converter (SRC) conforms to the generic audio codec API.

The flow chart of the command sequence used in the example testbench is shown in Figure 3.

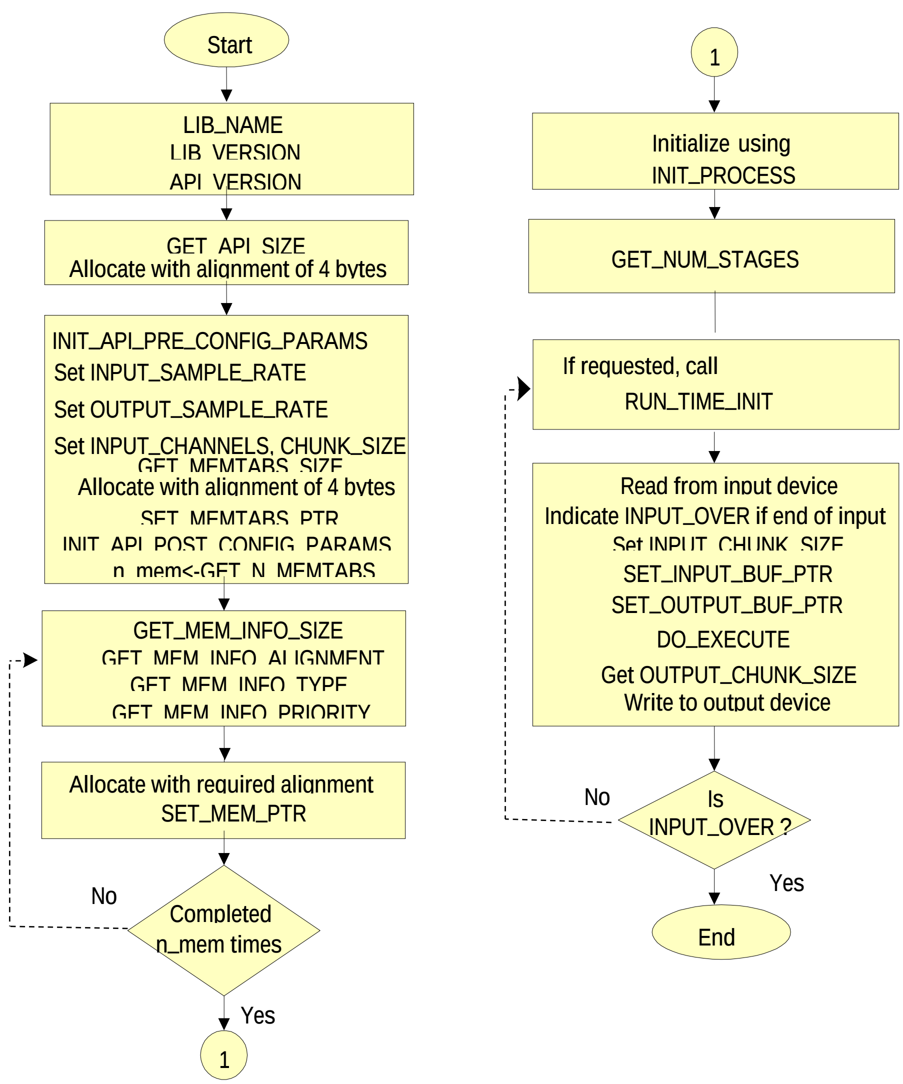


Figure 3 Flow Chart for HiFi SRC Application Wrapper

3.1 Files Specific to the Sample Rate Converter

The SRC parameter header file (`include/src_pp`)

- `xa_src_pp_api.h`

The SRC library (`lib`)

- `xa_asrc_src_pp.a`

The SRC API call is defined as:

```
XA_ERRORCODE xa_src_pp (xa_codec_handle_t p_xa_src_pp_obj,
                              WORD32            i_cmd,
                              WORD32            i_idx,
                              pVOID             pv_value);
```

The files inside `algo/utilities` contain the helper files related to configuring the HiFi SRC library with user-defined customer specifc SRC features. These features are available only to the source package users. For more information, see section 4.3.

3.2 Configuration Parameters

The SRC library accepts the following parameters from the user. The details of the SET CONFIG API for these parameters are described in Section 3.4.2.

- **`ch:`** Number of channels in the input stream.
- **`inrate:`** Input sample rate – samples per second in the input stream.
  - Expected value is a standard audio sample rate from the set {4000, 6000, 8000, 11025, 12000, 16000, 22050, 24000, 32000, 44100, 48000, 64000, 88200, 96000, 128000, 176400, 192000, 352800, 384000} Hz.
- **`insize:`** Maximum number of samples per channel in the input buffer for processing (also referred to as input chunk size in this document).
  - Input chunk size can be any value from 4 to 512 samples.
- **`outrate:`** Output sample rate – desired samples per second in the output stream.
  - Expected value is a standard audio sample rate {4000, 6000, 8000, 11025, 12000, 16000, 22050, 24000, 32000, 44100, 48000, 64000, 88200, 96000, 128000, 176400, 192000, 352800, 384000} Hz
- **`pcmwidth:`** Width of PCM sample in bytes.
  - Default value is 3. Valid values are 2 or 3, which you need to provide for raw PCM input files. Note that this value cannot be changed at runtime.
- **`reset:`** Reset SRC module at a specified frame number.

- **`enable_asrc:`** Enable ASRC mode when the library is built with the preprocessor flag ASRC_ENABLE (the library is built with the flag enabled).
  - Default value is 0 (disabled). Valid values are 0 or 1.
- **`drift_asrc:`** Drift applied to one output sample in ASRC mode.
- **`enable_cubic:`** Enable cubic interpolation for fractional ratios when the library is built with the preprocessor flag **`POLYPHASE_CUBIC_INTERPOLATION`** (the library is built with the flag enabled).
  - Default value is 0 (disabled). Valid values are 0 or 1.
  - Currently, cubic interpolation is init-time configurable only.

3.3 Usage Notes

Although HiFi SRC conforms to the generic codec API described in section 2, take the following into account to ensure correct SRC operation:

- For 24-bit input signals, the HiFi SRC library supports 24-bit PCM data stored as a 32-bit word and is aligned to the most significant 24 bits.
- For 16- bit input signals, the HiFi SRC library supports 16-bit PCM data stored in a 16-bit word.
- For a multichannel input, the PCM data from different channels is to be provided in an interleaved method to the library. The library generates an interleaved output.
- When the input and output sample rates are the same, the converter is set to the bypass mode and copies the samples in the input buffer to the output buffer. The bypass only applies to non-ASRC modes.
- If the input or output sample rate is set as 4 kHz or 6 kHz then the output or input sample rate should be 16 or 48kHz respectively. Other sampling conversion ratios for 4 and 6kHz sample rates are not supported.
- If the output sample rate is 384 kHz; the supported input sample rates are 32, 44.1, 48, 96, and 192kHz, but if the input sample rate is 384kHz, the output sample rate should be 48kHz. Similarly, if the input or output sample rate is set to 352.8 kHz then the output or input sample rate should be 48kHz respectively.
- The input chunk size can be any value from 4 to 512, and must be an integer multiple of 4. The default value is 512.
- Note that with smallest input chunk size (starting with 4 samples/channel), there will not be output every frame for some conversion ratios (i.e., zero output-chunk size). Output chunk size for each frame is determined based on input chunk size, constrained to keep the conversion ratio. Larger the input chunk size, less frequent zero output-chunk size, and better the cycle performance. Here are general guidelines on required minimum input chunk sizes which guarantees valid output for every input frame:
  - Upsample integer ratios : 4 samples
    - Upsample by 3 : 4 samples
    - Upsample by 2 : 4 samples
  - Downsample integer ratios : ratio*4 samples
    - Downsample by 3 : 12 samples

        - Downsample by 2 : 8 samples
  - Fractional ratios which don't use polyphase implementation:
    - Downsample ratio 3:2 : 12 samples
    - Downsample ratio 4:3 : 8 samples
    - Upsample ratio 2:3 : 8 samples
    - Upsample ratio 3:4 : 6 samples
  - Fractional ratios which use polyphase implementation, e.g,
    - 44.1 kHz to 32 kHz conversion: 4 samples
    - 48 kHz to 44.1 kHz conversion: 4 samples
    - 64 kHz to 44.1 kHz conversion: 8 samples
    - 44.1 kHz to 48 kHz conversion: 4 samples

The above listed cases represent unique ratios. Scaling of sample rate conversion ratio would result in scaling of minimum input chunk size by same factor. Recommend using SRC test bench application in the release package to find such minimum input chunk size required for specific conversion ratios of interest.

3.4 HiFi SRC Specific Commands and Errors

This section lists the commands and errors unique to the HiFi SRC. They are listed in sections based on their primary commands type (`i_cmd`). Each section contains a table for every subcommand. In the case of no subcommands, the primary command is presented.

3.4.1Initialization and Execution Errors

These errors can result from the initialization or execution of the API calls, and may be encountered during SET_CONFIG_PARAM API calls:

- XA_SRC_PP_CONFIG_FATAL_INVALID_INPUT_RATE
  - Unsupported input sample rate
- XA_SRC_PP_CONFIG_FATAL_INVALID_OUTPUT_RATE
  - Unsupported output sample rate
- XA_SRC_PP_CONFIG_FATAL_INVALID_INPUT_CHUNK_SIZE
  - During init time, this error occurs when the parameter input chunk size (number of samples per channel) value is out of range [4, 512] or the value is not an integer multiple of 4.
  - During runtime, this error occurs when the parameter input chunk size specified (number of samples per channel) is more than the one specified at init time, or the value is not an integer multiple of 4.
- XA_SRC_PP_CONFIG_FATAL_INVALID_INPUT_CHANNELS
  - Number of input channels value is more than 24 or less than 1.

- XA_SRC_PP_CONFIG_FATAL_INVALID_BYTES_PER_SAMPLE
  - PCM width should be either 2 or 3.
- XA_SRC_PP_CONFIG_NON_FATAL_INVALID_ENABLE_ASRC
  - Enable ASRC flag must be either 0 or 1.
- XA_SRC_PP_CONFIG_NON_FATAL_INVALID_DRIFT_ASRC
  - Drift ASRC must be between the ranges -0.04 to 0.04.
  - Drift should be applied only when ASRC is enabled.
- XA_SRC_PP_CONFIG_NON_FATAL_INVALID_ENABLE_CUBIC
  - Enable cubic interpolation flag must be either 0 or 1.
- XA_SRC_PP_CONFIG_NON_FATAL_INVALID_CONFIG_TYPE
  - This is typically caused due to unsupported parameter combinations in SET_CONFIG or GET_CONFIG calls.

The following errors may be encountered during initialization or execution API calls.

- XA_SRC_PP_EXECUTE_FATAL_ERR_POST_CONFIG_INIT

  Fatal error encountered during post configuration initialization. This is typically caused by unsupported SET_CONFIG parameter combinations.

- XA_SRC_PP_EXECUTE_FATAL_ERR_INIT

  Fatal error encountered during SRC initialization.

- XA_SRC_PP_EXECUTE_FATAL_ERR_EXECUTE

  Fatal error encountered during SRC execution. This may be due to wrong API sequence, i.e., EXEC API is called before successful INIT.

- XA_SRC_PP_EXECUTE_NON_FATAL_INVALID_CONFIG_SEQ

  This error is returned when the application tries to change a configuration parameter that is not allowed to be changed during the processing loop. The new value is rejected by the SRC library. Such configuration parameter changes can only be done by following the correct initialization sequence starting from SET_CONFIG followed by POST_CONFIG_INIT API.

- XA_SRC_PP_EXECUTE_NON_FATAL_INVALID_API_SEQ

  This error is returned when the application wrapper calls an out of sequence API. The API call is ignored by the SRC library.

3.4.2 XA_API_CMD_SET_CONFIG_PARAM

Table 3-33 XA_SRC_PP_CONFIG_PARAM_INPUT_SAMPLE_RATE subcommand

Subcommand	XA_SRC_PP_CONFIG_PARAM_INPUT_SAMPLE_RATE
Description	This command sets the input sampling frequency parameter. This parameter cannot be changed after initialization.
Actual Parameters	▪ p_xa_module_obj api_obj – Pointer to the API structure. ▪ i_cmd XA_API_CMD_SET_CONFIG_PARAM ▪ i_idx XA_SRC_PP_CONFIG_PARAM_INPUT_SAMPLE_RATE ▪ pv_value &fs_in – Pointer to the input sample rate variable.
Restrictions	Valid values: Standard audio sample rate from the set {8000, 11025, 12000, 16000, 22050, 24000, 32000, 44100, 48000, 64000, 88200, 96000, 128000, 176400, 192000} Hz. Default value is 48000 Hz.

Example

```
int fs_in = 48000;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_SRC_PP_CONFIG_PARAM_INPUT_SAMPLE_RATE,
                  (pVOID) &fs_in);
```

Errors

- Common API Errors
- XA_SRC_PP_CONFIG_FATAL_INVALID_INPUT_RATE

  Value is not a standard audio sample rate.

Table 3-34 XA_SRC_PP_CONFIG_PARAM_OUTPUT_SAMPLE_RATE subcommand

Subcommand	`XA_SRC_PP_CONFIG_PARAM_OUTPUT_SAMPLE_RATE`
Description	This command sets the output sampling frequency parameter. This parameter cannot be changed after initialization.

Actual Parameters	▪ `p_xa_module_obj`
	`api_obj` – Pointer to the API structure.
	▪ `i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	▪ `i_idx`
	`XA_SRC_PP_CONFIG_PARAM_OUTPUT_SAMPLE_RATE`
	▪ `pv_value`
	`&fs_out` – Pointer to the output sample rate variable.

Restrictions	Valid values: Standard audio sample rates from the set {8000, 11025, 12000, 16000, 22050, 24000, 32000, 44100, 48000, 64000, 88200, 96000, 128000, 176400, 192000} Hz. The default value is 48000 Hz.

Example

```
int fs_out = 44100;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_SRC_PP_CONFIG_PARAM_OUTPUT_SAMPLE_RATE,
                  (pVOID) &fs_out);
```

Errors

- Common API Errors
- XA_SRC_PP_CONFIG_FATAL_INVALID_OUTPUT_RATE

  Value is not a standard audio sample rate.

Table 3-345 XA_SRC_PP_CONFIG_PARAM_INPUT_CHUNK_SIZE subcommand

Subcommand	`XA_SRC_PP_CONFIG_PARAM_INPUT_CHUNK_SIZE`
Description	This command sets the input chunk size parameter. This parameter can change during runtime.

Actual Parameters	▪ `p_xa_module_obj`
	`api_obj` – Pointer to the API structure.
	▪ `i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	▪ `i_idx`
	`XA_SRC_PP_CONFIG_PARAM_INPUT_CHUNK_SIZE`
	▪ `pv_value`
	`&insize_chunk` – Pointer to the input chunk size variable.

Restrictions	Valid value: between 4 and 512 and integer multiple of 4. During the processing loop, the value should be no larger than the INPUT_CHUNK_SIZE value set during the INIT process.

Example

```
int insize_chunk = 256;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_SRC_PP_CONFIG_PARAM_INPUT_CHUNK_SIZE,
                  (pVOID) &insize_chunk);
```

Errors

- Common API Errors
- XA_SRC_PP_CONFIG_FATAL_INVALID_INPUT_CHUNK_SIZE
  - Value is not between 4 and 512 or an integer multiple of 4.
  - In EXECUTE process, the value is larger than the value set during INIT process.

Table 3-36 XA_SRC_PP_CONFIG_PARAM_INPUT_CHANNELS subcommand

Subcommand	XA_SRC_PP_CONFIG_PARAM_INPUT_CHANNELS
Description	This command sets the number of channels in the input stream. This parameter cannot be changed after initialization.
Actual Parameters	▪ p_xa_module_obj api_obj – Pointer to the API structure. ▪ i_cmd XA_API_CMD_SET_CONFIG_PARAM ▪ i_idx XA_SRC_PP_CONFIG_PARAM_INPUT_CHANNELS ▪ pv_value &nch – Pointer to the channel count variable.
Restrictions	Valid values: Between 1 to 24. Default is 2. This parameter cannot be changed after initialization.

Example

```
int nch = 6;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_SRC_PP_CONFIG_PARAM_INPUT_CHANNELS,
                  (pVOID) &nch);
```

Errors

- Common API Errors
- XA_SRC_PP_CONFIG_FATAL_INVALID_INPUT_CHANNELS

  Value is not valid

Table 3-37 XA_SRC_PP_CONFIG_PARAM_BYTES_PER_SAMPLE subcommand

Subcommand	XA_SRC_PP_CONFIG_PARAM_BYTES_PER_SAMPLE
Description	This command sets the number of bytes per PCM sample. This parameter cannot be changed after initialization.
Actual Parameters	▪ p_xa_module_obj api_obj – Pointer to the API structure. ▪ i_cmd XA_API_CMD_SET_CONFIG_PARAM ▪ i_idx XA_SRC_PP_CONFIG_PARAM_BYTES_PER_SAMPLE ▪ pv_value &bytes_per_sample – Pointer to the PCM width variable.
Restrictions	Valid values: 2 or 3. Default is 3. This parameter cannot be changed after initialization.

Example

```
int bytes_per_sample = 2;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_SRC_PP_CONFIG_PARAM_BYTES_PER_SAMPLE,
                  (pVOID) &bytes_per_sample);
```

Errors

- Common API Errors
- XA_SRC_PP_CONFIG_FATAL_INVALID_BYTES_PER_SAMPLE

  Value is not valid

Table 3-38 XA_SRC_PP_CONFIG_PARAM_SET_INPUT_BUF_PTR subcommand

Subcommand	`XA_SRC_PP_CONFIG_PARAM_SET_INPUT_BUF_PTR`
Description	This command sets the input PCM buffer pointers, which can change after initialization. The library assumes the input is available in an interleaved format.

Actual Parameters	▪ `p_xa_module_obj`
	`api_obj` – Pointer to the API structure.
	▪ `i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	▪ `i_idx`
	`XA_SRC_PP_CONFIG_PARAM_SET_INPUT_BUF_PTR`
	▪ `pv_value`
	`pin` – Pointer to the array of input buffer pointers.

Restrictions	Valid pointer to input PCM pointers array. The input PCM pointers should point to an address having an alignment of 4 bytes for both 24-bit and 16-bit input signals.

Example

```
int *pin;
pin = &inbuf[0];
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_SRC_PP_CONFIG_PARAM_SET_INPUT_BUF_PTR,
                  (pVOID) pin);
```

Errors

- Common API Errors

  XA_SRC_PP_CONFIG_FATAL_INVALID_INPUT_PTR

Table 3-39 XA_SRC_PP_CONFIG_PARAM_SET_OUTPUT_BUF_PTR subcommand

Subcommand	XA_SRC_PP_CONFIG_PARAM_SET_OUTPUT_BUF_PTR
Description	This command sets the output PCM buffer pointers, which can change after initialization. The library generates output in an interleaved format.
Actual Parameters	▪ p_xa_module_obj api_obj – Pointer to the API structure. ▪ i_cmd XA_API_CMD_SET_CONFIG_PARAM ▪ i_idx XA_SRC_PP_CONFIG_PARAM_SET_OUTPUT_BUF_PTR ▪ pv_value pout – Pointer to the output buffer.
Restrictions	Valid pointer to output PCM array. The output PCM pointers should point to an address having an alignment of 4 bytes.

Example

```
int *pout;
pout = &outbuf[0];
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_SRC_PP_CONFIG_PARAM_SET_OUTPUT_BUF_PTR,
                  (pVOID) pout);
```

Errors

- Common API Errors

  XA_SRC_PP_CONFIG_FATAL_INVALID_OUTPUT_PTR

Table 3-40 XA_SRC_PP_CONFIG_PARAM_ENABLE_ASRC subcommand

Subcommand	`XA_SRC_PP_CONFIG_PARAM_ENABLE_ASRC`
Description	This command enables or disables ASRC mode. The library must be built with the pre-processor flag `ASRC_ENABLE`.

Actual Parameters	▪ `p_xa_module_obj`
	`api_obj` – Pointer to the API structure.
	▪ `i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	▪ `i_idx`
	`XA_SRC_PP_CONFIG_PARAM_ENABLE_ASRC`
	▪ `pv_value`
	`&enable_asrc` – Pointer to the ASRC enable flag variable.

Restrictions	Value must be 0 or 1. This command is not allowed at runtime.

Example

```
int enable_asrc;
enable_asrc = 1;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_SRC_PP_CONFIG_PARAM_ENABLE_ASRC,
                  &enable_asrc);
```

Errors

- Common API Errors
- XA_SRC_PP_CONFIG_NON_FATAL_INVALID_ENABLE_ASRC

  Invalid config for `enable_asrc`. Value must be either 0 or 1.

Table 3-41 XA_SRC_PP_CONFIG_PARAM_DRIFT_ASRC subcommand

Subcommand	`XA_SRC_PP_CONFIG_PARAM_DRIFT_ASRC`
Description	Drift applied to one output sample in ASRC mode. This parameter can be modified at runtime.

Actual Parameters	▪ `p_xa_module_obj`
	`api_obj` – Pointer to the API structure.
	▪ `i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	▪ `i_idx`
	`XA_SRC_PP_CONFIG_PARAM_DRIFT_ASRC`
	▪ `pv_value`
	`&drift_asrc` – Pointer to the drift amount variable.

Restrictions	Value must be between -0.04 and +0.04 in Q31. It can be set only when ENABLE_ASRC is 1.

Example

```
int drift_asrc;
drift_asrc = ((int)((-0.04)*( (long long)1 << 31) ));
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_SRC_PP_CONFIG_PARAM_DRIFT_ASRC,
                  &drift_asrc);
```

Errors

- Common API Errors
- XA_SRC_PP_CONFIG_NON_FATAL_INVALID_DRIFT_ASRC
  - Invalid value. Value must be between -0.04 and +0.04 in Q31.
  - Drift can be set only when ENABLE_ASRC is 1.

Table 3-42 XA_SRC_PP_CONFIG_PARAM_ENABLE_CUBIC subcommand

Subcommand	`XA_SRC_PP_CONFIG_PARAM_ENABLE_CUBIC`
Description	This command enables or disables cubic interpolation for fractional ratios. The library must be built with the pre-processor flag `POLYPHASE_CUBIC_INTERPOLATION`.

Actual Parameters	▪ `p_xa_module_obj`
	`api_obj` – Pointer to the API structure.
	▪ `i_cmd`
	`XA_API_CMD_SET_CONFIG_PARAM`
	▪ `i_idx`
	`XA_SRC_PP_CONFIG_PARAM_ENABLE_CUBIC`
	▪ `pv_value`
	`&enable_cubic` – Pointer to the cubic interpolation flag variable.

Restrictions	Value must be 0 or 1. This command is not allowed at runtime.

Example

```
int enable_cubic;
enable_cubic= 1;
res = (*api_func)(api_obj,
                  XA_API_CMD_SET_CONFIG_PARAM,
                  XA_SRC_PP_CONFIG_PARAM_ENABLE_CUBIC,
                  &enable_cubic);
```

Errors

- Common API Errors
- XA_SRC_PP_CONFIG_NON_FATAL_INVALID_ENABLE_CUBIC
  - Invalid config for enable cubic. Value must be either 0 or 1.

3.4.3XA_API_CMD_GET_CONFIG_PARAM

Table 3-43 XA_SRC_PP_CONFIG_PARAM_OUTPUT_CHUNK_SIZE subcommand

Subcommand	XA_SRC_PP_CONFIG_PARAM_OUTPUT_CHUNK_SIZE
Description	This command gets the number of samples available in the output buffer after SRC execution.
Actual Parameters	▪ p_xa_module_obj api_obj – Pointer to the API structure. ▪ i_cmd XA_API_CMD_GET_CONFIG_PARAM ▪ i_idx XA_SRC_PP_CONFIG_PARAM_OUTPUT_CHUNK_SIZE ▪ pv_value &outsize – Pointer to the output sample count variable.
Restrictions	None

Example

```
int outsize;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_SRC_PP_CONFIG_PARAM_OUTPUT_CHUNK_SIZE,
                  (pVOID) &outsize);
```

Errors

- Common API Errors

Table 3-44 XA_SRC_PP_CONFIG_PARAM_GET_NUM_STAGES subcommand

Subcommand	XA_SRC_PP_CONFIG_PARAM_GET_NUM_STAGES
Description	This command gets the number of stages in which the SRC achieves the conversion, wherever applicable. The sample rate conversion is performed by factoring the conversion factor into multiples of smaller conversion rates, and the conversion is achieved via cascade of these smaller conversions performed in multiple stages. Maximum number of stages is 6.
Actual Parameters	▪ p_xa_module_obj api_obj – Pointer to the API structure. ▪ i_cmd XA_API_CMD_GET_CONFIG_PARAM ▪ i_idx XA_SRC_PP_CONFIG_PARAM_GET_NUM_STAGES ▪ pv_value &num_stages – Pointer to the stage count variable.
Restrictions	Value returned is between 1 and 6. If the value returned is <1, it indicates bad initialization.

Example

```
int num_stages;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_SRC_PP_CONFIG_PARAM_GET_NUM_STAGES,
                  (pVOID) &num_stages);
```

Errors

- Common API Errors

Table 3-45 XA_SRC_PP_CONFIG_PARAM_GET_DRIFT_FRACT_ASRC subcommand

Subcommand	`XA_SRC_PP_CONFIG_PARAM_GET_DRIFT_FRACT_ASRC`
Description	This command gets the remaining fraction of input samples in Q31 formats.

Actual Parameters	▪ `p_xa_module_obj`
	`api_obj` – Pointer to the API structure.
	▪ `i_cmd`
	`XA_API_CMD_GET_CONFIG_PARAM`
	▪ `i_idx`
	`XA_SRC_PP_CONFIG_PARAM_GET_DRIFT_FRACT_ASRC`
	▪ `pv_value`
	`&pre_Fm` – Pointer to the 64-bit remaining fraction variable.

Restrictions	Value returned is in Q31. Divide this value by 1<<31 to represent it in float.

Example

```
WORD64 pre_Fm;
res = (*api_func)(api_obj,
                  XA_API_CMD_GET_CONFIG_PARAM,
                  XA_SRC_PP_CONFIG_PARAM_GET_DRIFT_FRACT_ASRC,
                  (pVOID) & pre_Fm);
```

Errors

- Common API Errors

4. Introduction to the Example Test Bench

The SRC library is released as a .tgz file for Linux/makefile based usage, and an .xws file for Xtensa Xplorer based usage.

The supplied test bench includes the following files:

- Test bench source files (`test/src`)
  - `xa_src_pp_sample_testbench_hifi3.c`
  - `xa_src_pp_error_handler.c`
  - `xa_src_pp_waveio.c`
  - `xa_src_pp_waveio.h`
- Makefile to build the executable (`test/build`)
  - `makefile_testbench_sample`
- Sample parameter file to run the test bench(`test/build`)
  - `paramfile_src_pp.txt`

4.1 Making an Executable

Build the Application

1. Go to `test/build`.
2. Type: `xt-make -f makefile_testbench_sample clean all`

This will build the SRC example test bench application xa_asrc_src_pp_test.

Note If you have source code distribution, you must build the SRC library before you can build the test bench.

Build the Library

6. Go to the `build` directory.
7. Type: `xt-make clean all install REDUCE_ROM_SIZE=1`

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

To build and run the application from xws based release package, refer to the readme.html file available in the imported application project.

Note The test bench works only in the least significant byte (LSB) environment.

4.2 Usage

The sample application executable can be run with direct command-line options or with a parameter file. For running the sample application from Xtensa Xplorer workspace, refer to the readme.html file available in the imported project.

The command line usage is as follows:

```
xt-run <testbench>
      -ifile:< input filename> \
      -ofile:< output filename> \
      -inrate:<input sample rate> \
      -outrate:<output sample rate>\
      -insize:<input chunk size>\
      -ch:< number of input channels>\
      -pcmwidth:< width of pcm sample in bytes>\
      -reset:<reset value> \
      -enable_asrc:<enable asrc> \
      -drift_asrc:<asrc drift value>
      -enable_cubic:<enable cubic interpolation>
```

Where:

`<testbench>`	`xa_asrc_src_pp_test`
<input filename>	Complete path for the input file.
<output filename>	Complete path for the output file.
<input sample rate>	Samples per second in the input stream. In case of raw pcm input files, the user must provide this information (default = 48000). Expected value is a standard audio sample rate between 4 kHz to 384kHz*
<output sample rate>	Required samples per second in the output stream (default = 48000). Expected value is a standard audio sample rate between 4 kHz to 384kHz*.
<input chunk size>	Number of pcm samples in the input buffer (default = 512). Input chunk size can be from 4 to 512.

<number of input channels>	In case of raw pcm input files, the user must provide this information Range: 1-24 (default = 2).
<width of pcm sample in bytes>	In case of raw pcm input files, the user must provide this information Pcmwidth should be 2 or 3 bytes(default = 3 bytes).
<reset value>	The frame number at which enabling runtime init of the SRC is done (default value is 0). Multiple frame numbers are not allowed
<enable asrc>	Enable or disable asrc (default = 0, asrc is disabled by default) For the current library, any value other than 0 or 1 is invalid.
<asrc drift value>	Drift applied to one output sample. This value must be between the ranges -0.04 to 0.04 (default = 0) with step of 0.000001.
<enable_cubic interpolation>	Enable or disable cubic interpolation (default = 0, cubic interpolation is disabled by default). For the current library, any value other than 0 or 1 is invalid.

Refer to the parameter definitions in section 3.2 for a full description of their usage.

Note To reset/re-init the HiFi SRC, give the reset parameter to the sample test bench as the frame number at which you need to reset the SRC. The reset parameter value should be more than 0.

Note The output file format follows that of the input file. Thus, if the input is a WAV file, the output is also a WAV file; and if the input is a PCM file, the output is a PCM file. When the input is a PCM file, the input sample rate, the number of channels, and the PCM width (bytes per sample) are required; when the input is a WAV file, the information can be obtained from the WAV file header.

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

Following is the syntax for writing the `paramfile_src_pp.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 SRC 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 test bench program.

Note All the `@<command>`s should be at the first column of a line except the `@New_line` command.

Note All the `@<command>`s are case sensitive. If the command line in the parameter file must be separated on two different lines, use the `@New_line` command.
For example:
```
<command line part 1> @New_line
<command line part 2>
```

Note Blank lines will be ignored.

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

4.3 Customizing the Library

In addition to source code and relative makefile to build the standard SRC library, the source package also contains modules and utilities that help in modifying and customizing the standard SRC library. All the code and scripts related to this are present in the `./algo/utilities` directory.

You can build a library with different modifications.

4.3.1 The Custom_Library Folder

This sub-folder contains utilities and scripts required to generate modular custom SRC library with the required features and conversion ratios. This method results in saving of static codesize and ROM memory when use-cases are known in advance. Multiple config parameters are provided to enable/disable different features like multichannel support, ASRC, Cubic interpolation, etc. to achieve static memory saving.

The Readme.txt in this sub-folder describes the steps to update the .csv file and run the python script to generate custom library. An example of csv file to generate the SRC library with one upsampling (48 kHz -> 96 kHz) and one downsampling conversion (96 kHz -> 16 kHz) is provided as a part of source package.

Note This support is available for customers with source code license.

5. Reference

[1] *Digital Signal Processing – Principles, Algorithms and Applications (4th Edition):* John G Proakis, Dimitris G Manolakis

[2] AES 119 Convention Paper, 2005 by P Beckmann at el "An efficient asynchronous sampling rate conversion algorithm for multi-channel audio applications"

[3] IEEE tran signal processing Dec 1996 by See May Phoong at el, "*Time-varying Filter and filter banks: Some basic principles*"