SAMPLE RATE CONVERSION - Programming Guide

Release: 2.4.0
Publication Date: 2024/02/10

Table of Contents

1 Sample Rate Conversion Library

1.1 Overview

The XMOS Sample Rate Conversion (SRC) library provides both synchronous and asynchronous audio sample rate conversion functions for use on xCORE multicore micro-controllers.

In systems where the rate change is exactly equal to the ratio of nominal rates, synchronous sample rate conversion (SSRC) provides efficient and high performance rate conversion. Where the input and output rates are not locked by a common clock or clocked by an exact rational frequency ratio, the Asynchronous Sample Rate Converter (ASRC) provides a way of streaming high quality audio between the two different clock domains, at the cost of higher processing resource usage. ASRC can ease interfacing in cases where there are multiple digital audio inputs or allow cost saving by removing the need for physical clock recovery using a PLL.

1.1.1 Features

Multi-rate Hi-Fi functionality:

- Conversion between 44.1, 48, 88.2, 96, 176.4 and 192 KHz input and output sample rates.
- 32 bit PCM input and output data in Q1.31 signed format.
- Optional output dithering to 24 bit using Triangular Probability Density Function (TPDF).
- Optimized for xCORE-200 instruction set with dual-issue and compatible with XCORE-AI.
- Block based processing - Minimum 4 samples input per call, must be power of 2.
- Up to 10000 ppm sample rate ratio deviation from nominal rate (ASRC only).
- Very high quality - SNR greater than 135 dB (ASRC) or 140 dB (SSRC), with THD of less than 0.0001% (reference 1KHz).
- Configurable number of audio channels per SRC instance.
- Reentrant library permitting multiple instances with differing configurations and channel count.
- No external components (PLL or memory) required.

Fixed factor functionality:

- Synchronous fixed factor of 3 downsample and oversample functions supporting either HiFi quality or reduced resource requirements for voice applications.
- Synchronous fixed factor of 3 and 3/2 downsample and oversample functions for voice applications optimized for the XS3 Vector Processing Unit.

1.1.2 Building

The library can be built under *cmake* or *xcommon* via *xmake* offering backwards compatibility for legacy applications. It is recommended to use *cmake* where the library name *lib_src* is included in the cmake files. See *Related application notes* for example usage. The library has no dependencies when building under *cmake* although does require *lib_logging* and *lib_xassert* when using *xcommon*.

1.1.3 Components

- Synchronous Sample Rate Converter function (ssrc)
- Asynchronous Sample Rate Converter function (asrc)
- Synchronous factor of 3 downsample function (ds3)
- Synchronous factor of 3 oversample function (os3)
- Synchronous factor of 3 downsample function optimized for use with voice (src_ds3_voice)
- Synchronous factor of 3 oversample function optimised for use with voice (src_us3_voice)
- Synchronous factor of 3 downsample function for use with voice optimized for XS3 (ff3_96t_ds)
- Synchronous factor of 3 oversample function for use with voice optimized for XS3 (ff3_96t_us)
- Synchronous factor of 3/2 downsample function for use with voice optimized for XS3 (rat_2_3_96t_ds)
- Synchronous factor of 3/2 oversample function for use with voice optimized for XS3 (rat_3_2_96t_us)

There are three different component options that support fixed factor of 3 up/downsampling. To help choose which one to use follow these steps:

1. If HiFi quality (130 dB SNR) up/downsampling is required, use ds3 or os3.
2. If voice quality (65 dB SNR) is required running on xCORE-200, use ds3_voice or us3_voice.
3. If voice quality (75 dB SNR) is required running xcore-ai, use ff3_96t_ds or ff3_96t_us.

Warning: Synchronous fixed factor of 3 and 3/2 downsample and oversample functions for voice applications optimized for the XS3 Vector Processing Unit currently overflow rather than saturate in cases where a full scale input causes a perturbation above full scale at the output. To avoid this scenario, please ensure that the input amplitude is always 3.5 dB below full scale. The overflow behavior of these SRC components will be replaced by saturating behavior (to match all other SRC components) in a future release.

1.1.4 Related Application Notes

An adaptive USB Audio ASRC example can be found in https://github.com/xmos/sln_voice/tree/develop/examples.

Simple file-based test applications may be found in this repo under *tests/xxxx_test* where xxxx is either *asrc*, *ssrc*, *ds3*, *os3*, *ds3_voice*, *os3_voice*, *vpu_ff3* or *vpu_rat*. These test applications may be used as basic examples to show how these components are used in a minimal application.

2 Multi-rate HiFi Sample Rate Conversion

2.1 Usage

Both SSRC and ASRC functions are accessed via standard function calls, making them accessible from C or XC. Both SSRC and ASRC functions are passed an external state structure which provides re-entrancy. The functions may be called in-line with other signal processing or placed on a logical core within its own task to provide guaranteed performance. By placing the calls to SRC functions on separate logical cores, multiple instances can be processed concurrently.

The API is designed to be simple and intuitive with just two public functions per sample rate converter type.

2.1.1 Initialization

All public ASRC and SSRC functions are declared within the `src.h` header:

```
#include "src.h"
```

There are a number of arrays of structures that must be declared from the application which contain the buffers between the FIR stages, state and adapted coefficients (ASRC only). There must be one element of each structure declared for each channel handled by the SRC instance. The structures are then all linked into a single control structure, allowing a single reference to be passed each time a call to the SRC is made.

For SSRC, the following state structures are required:

```
//State of SSRC module
ssrc_state_t     ssrc_state[SSRC_CHANNELS_PER_INSTANCE];
//Buffers between processing stages
int              ssrc_stack[SSRC_CHANNELS_PER_INSTANCE][SSRC_STACK_LENGTH_MULT * SSRC_N_IN_SAMPLES];
//SSRC Control structure
ssrc_ctrl_t      ssrc_ctrl[SSRC_CHANNELS_PER_INSTANCE];
```

For ASRC, the following state structures are required. Note that only one instance of the filter coefficients need be declared because these are shared amongst channels within the instance:

```
//ASRC state
asrc_state_t       asrc_state[ASRC_CHANNELS_PER_INSTANCE];
int                asrc_stack[ASRC_CHANNELS_PER_INSTANCE][ASRC_STACK_LENGTH_MULT * ASRC_N_IN_
↪SAMPLES];
//Control structure
asrc_ctrl_t        asrc_ctrl[ASRC_CHANNELS_PER_INSTANCE];
//Adaptive filter coefficients
asrc_adfir_coefs_t asrc_adfir_coefs;
```

There is an initialization call which sets up the variables within the structures associated with the SRC instance and clears the inter-stage buffers. Initialization ensures the correct selection, ordering and configuration of the filtering stages, be they decimators, interpolators or pass-through blocks. This initialization call contains arguments defining selected input and output nominal sample rates as well as settings for the sample rate converter:

`ssrc_init()`

The initialization call is the same for ASRC:

`asrc_init()`

The input block size must be a power of 2 and is set by the `n_in_samples` argument. In the case where more than one channel is to be processed per SRC instance, the total number of input samples expected for each processing call is `n_in_samples * n_channels_per_instance`.

Please ensure the settings within `src_config.h` are correct for the application. The default settings allow for any input/output ratio between 44.1 kHz and 192 kHz.

2.1.2 Processing

Following initialization, the processing API is called for each block of input samples which will then produce a block of output samples as shown in Fig. 2.1.

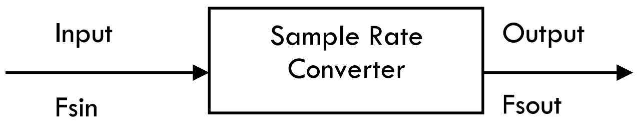

Fig. 2.1: SRC Operation

The logic is designed so that the final filtering stage always receives a sample to process. The sample rate converters have been designed to handle a maximum decimation of factor four from the first two stages. This architecture requires a minimum input block size of 4 to operate.

The processing function call is passed the input and output buffers and a reference to the control structure:

`ssrc_process()`

In the case of ASRC a fractional frequency ratio argument is also supplied:

`asrc_process()`

The SRC processing call always returns a whole number of output samples produced by the sample rate conversion. Depending on the sample ratios selected, this number may be between zero and `(n_in_samples * n_channels_per_instance * SRC_N_OUT_IN_RATIO_MAX)`. `SRC_N_OUT_IN_RATIO_MAX` is the maximum number of output samples for a single input sample. For example, if the input frequency is 44.1 kHz and the output rate is 192 kHz then a sample rate conversion of one sample input may produce up to 5 output samples.

The fractional number of samples produced to be carried to the next operation is stored inside the control structure, and additional whole samples are added during subsequent calls to the sample rate converter as necessary.

For example, a sample rate conversion from 44.1 kHz to 48 kHz with a input block size of 4 will produce a 4 sample result with a 5 sample result approximately every third call.

Each SRC processing call returns the integer number of samples produced during the sample rate conversion.

The SSRC is synchronous in nature and assumes that the ratio is equal to the nominal sample rate ratio. For example, to convert from 44.1 kHz to 48 kHz, it is assumed that the word clocks of the input and output stream are derived from the same master clock and have an exact ratio of 147:160.

If the word clocks are derived from separate oscillators, or are not synchronous (for example are derived from each other using a fractional PLL), the ASRC must be used.

2.1.3 Buffer Formats

The format of the sample buffers sent and received from each SRC instance is time domain interleaved. How this looks in practice depends on the number of channels and SRC instances. Three examples are shown below, each showing `n_in_samples = 4`. The ordering of sample indicies is 0 representing the oldest sample and `n - 1`, where n is the buffer size, representing the newest sample.

In the case where two channels are handled by a single SRC instance Fig. 2.2 shows that the samples are interleaved into a single buffer of size 8.

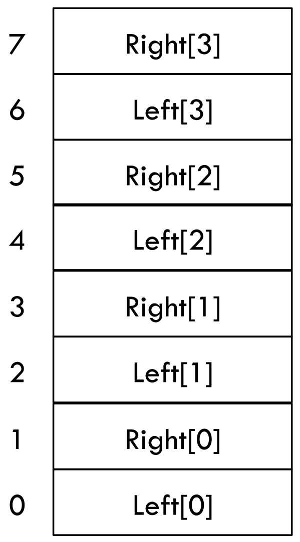

Fig. 2.2: Buffer Format for Single Stereo SRC instance

Where a single audio channel is mapped to a single instance, the buffers are simply an array of samples starting with the oldest sample and ending with the newest sample as shown in Fig. 2.3.

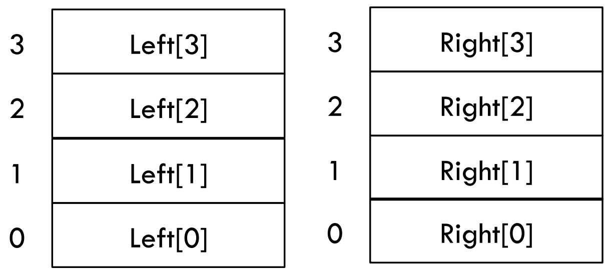

Fig. 2.3: Buffer Format for Dual Mono SRC instances

In the case where four channels are processed by two instances, channels 0 & 1 are processed by SRC instance 0 and channels 2 & 3 are processed by SRC instance 1 as shown in Fig. 2.4. For each instance, four pairs of samples are passed into the SRC processing function and n pairs of samples are returned, where n depends on the input and output sample rate ratio.

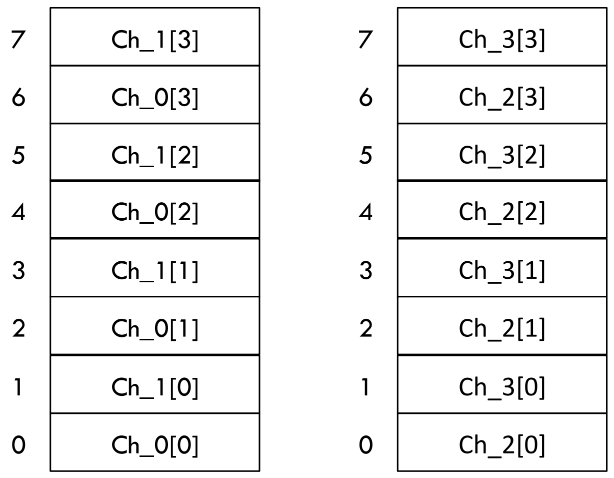

Fig. 2.4: Buffer Format for Dual Stereo SRC instances (4 channels total)

In addition to the above arguments the `asrc_process()` call also requires an unsigned Q4.28 fixed point ratio value specifying the actual input to output ratio for the next calculated block of samples. This allows the input and output rates to be fully asynchronous by allowing rate changes on each call to the ASRC. The converter dynamically computes coefficients using a spline interpolation within the last filter stage. It is up to the callee to maintain the input and output sample rate ratio difference.

Further details about these function arguments are contained here: *SSRC API*.

2.2 Performance and resource utilization

2.2.1 Audio Performance

The performance of the SSRC library is as follows:

- THD+N (1 kHz, 0 dBFs): better than -130 dB, depending on the accuracy of the ratio estimation
- SNR: 140 dB (or better). Note that when dither is not used, SNR is infinite as output from a zero input signal is zero.

To see frequency plots illustrating the noise floor with respect to a sample rate converted tone please refer to the performance-plots section of this document.

2.2.2 SSRC Resource utilization

The SSRC algorithm runs a series of cascaded FIR filters to perform the rate conversion. This includes interpolation, decimation and bandwidth limiting filters with a final polyphase FIR filter. The last stage supports the rational rate change of 147:160 or 160:147 allowing conversion between 44.1 kHz family of sample rates to the 48 kHz family of sample rates.

Tip: Table 2.1 shows the worst case MHz consumption at a given sample rate using the minimum block size of 4 input samples with dithering disabled. The MHz requirement can be reduced by around 8-12%, depending on sample rate, by increasing the input block size to 16. It is not usefully reduced by increasing block size beyond 16.

Table 2.1: SSRC Processor Usage per Channel (MHz)

		Output rate					
		44.1 kHz	**48 kHz**	**88.2 kHz**	**96 kHz**	**176.4 kHz**	**192 kHz**
Input rate	44.1 kHz	1 MHz	23 MHz	16 MHz	26 MHz	26 MHz	46 MHz
	48 kHz	26 MHz	1 MHz	28 MHz	17 MHz	48 MHz	29 MHz
	88.2 kHz	18 MHz	43 MHz	1 MHz	46 MHz	32 MHz	53 MHz
	96 kHz	48 MHz	20 MHz	52 MHz	2 MHz	56 MHz	35 MHz
	176.4 kHz	33 MHz	61 MHz	37 MHz	67 MHz	3 MHz	76 MHz
	192 kHz	66 MHz	36 MHz	70 MHz	40 MHz	80 MHz	4 MHz

2.2.3 ASRC Performance

The performance of the ASRC library is as follows:

- THD+N: (1 kHz, 0 dBFs): better than -130 dB
- SNR: 135 dB (or better). Note that when dither is not used, SNR is infinite as output from a zero input signal is zero.

To see frequency plots illustrating the noise floor with respect to a sample rate converted tone please refer to the performance-plots section of this document.

2.2.4 ASRC Resource utilization

The ASRC algorithm also runs a series of cascaded FIR filters to perform the rate conversion. The final filter is different because it uses adaptive coefficients to handle the varying rate change between the input and the output. The adaptive coefficients must be computed for each output sample period, but can be shared amongst all channels within the ASRC instance. Consequently, the MHz usage of the ASRC is expressed as two tables; Table 2.2 quantifies the MHz required for the first channel with adaptive coefficients calculation and Table 2.3 specifies the MHz required for filtering of each additional channel processed by the ASRC instance.

Tip: The below tables show the worst case MHz consumption per sample, using the minimum block size of 4 input samples. The MHz requirement can be reduced by around 8-12% by increasing the input block size to 16.

Tip: Typically some performance headroom is needed for buffering (especially if the system is sample orientated rather than block orientated) and inter-task communication.

Table 2.2: ASRC Processor Usage (MHz) for the First Channel in the ASRC Instance

		Output rate					
		44.1 kHz	48 kHz	88.2 kHz	96 kHz	176.4 kHz	192 kHz
Input rate	44.1 kHz	29 MHz	30 MHz	40 MHz	42 MHz	62 MHz	66 MHz
	48 kHz	33 MHz	32 MHz	42 MHz	43 MHz	63 MHz	66 MHz
	88.2 kHz	47 MHz	50 MHz	58 MHz	61 MHz	80 MHz	85 MHz
	96 kHz	55 MHz	51 MHz	67 MHz	64 MHz	84 MHz	87 MHz
	176.4 kHz	60 MHz	66 MHz	76 MHz	81 MHz	105 MHz	106 MHz
	192 kHz	69 MHz	66 MHz	82 MHz	82 MHz	109 MHz	115 MHz

Caution: Configurations requiring more than 100 MHz may not be able run in real time on a single logical core. The performance limit for a single core on a 500 MHz xCORE-200 device is 100 MHz (500/5) however an XCORE-AI device running at 600 MHz can provide 120 MHz logical cores.

Table 2.3: ASRC Processor Usage (MHz) for Subsequent Channels in the ASRC Instance

		Output rate					
		44.1 kHz	48 kHz	88.2 kHz	96 kHz	176.4 kHz	192 kHz
Input rate	44.1 kHz	28 MHz	28 MHz	32 MHz	30 MHz	40 MHz	40 MHz
	48 kHz	39 MHz	31 MHz	33 MHz	36 MHz	40 MHz	45 MHz
	88.2 kHz	51 MHz	49 MHz	57 MHz	55 MHz	65 MHz	60 MHz
	96 kHz	51 MHz	56 MHz	57 MHz	62 MHz	66 MHz	71 MHz
	176.4 kHz	60 MHz	66 MHz	76 MHz	79 MHz	92 MHz	91 MHz
	192 kHz	69 MHz	66 MHz	76 MHz	82 MHz	90 MHz	100 MHz

2.3 SRC Implementation

The SSRC and ASRC implementations are closely related to each other and share the majority of the system building blocks. The key difference between them is that SSRC uses fixed polyphase 160:147 and 147:160 final rate change filters whereas the ASRC uses an adaptive polyphase filter. The ASRC adaptive polyphase coefficients are computed for every sample using second order spline based interpolation.

2.3.1 SSRC Structure

The SSRC algorithm is based on three cascaded FIR filter stages (F1, F2 and F3). These stages are configured differently depending on rate change and only part of them is used in certain cases. Fig. 2.5 shows an overall view of the SSRC algorithm:

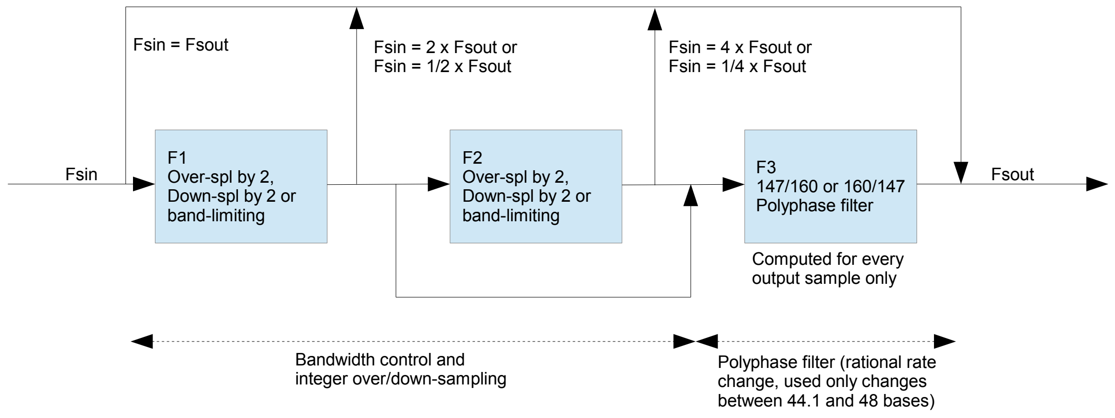

Fig. 2.5: SSRC Algorithm Structure

The SSRC algorithm is implemented as a two stage structure:

- The bandwidth control stage which includes filters F1 and F2 is responsible for limiting the bandwidth of the input signal and for providing integer rate Sample Rate Conversion. It is also used for signal conditioning in the case of rational non-integer Sample Rate Conversion.
- The polyphase filter stage which converts between the 44.1 kHz and the 48 kHz families of sample rates.

2.3.2 ASRC Structure

Similar to the SSRC, the ASRC algorithm is based on three cascaded FIR filters (F1, F2 and F3). These are configured differently depending on rate change and F2 is not used in certain rate changes. Fig. 2.6 shows an overall view of the ASRC algorithm:

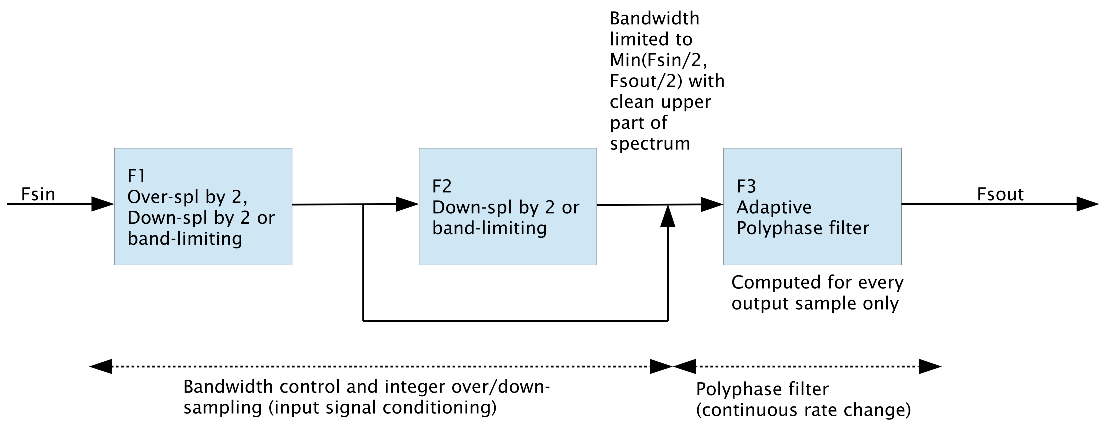

Fig. 2.6: ASRC Algorithm Structure

The ASRC algorithm is implemented as a two stage structure:

- The bandwidth control stage includes filters F1 and F2 which are responsible for limiting the bandwidth of the input signal and for providing integer rate sample rate conversion to condition the input signal for the adaptive polyphase stage (F3).
- The polyphase filter stage consists of the adaptive polyphase filter F3, which effectively provides the asynchronous connection between the input and output clock domains.

2.3.3 SRC Filter list

A complete list of the filters supported by the SRC library, both SSRC and ASRC, is shown in Table 2.4. The filters are implemented in C within the `FilterDefs.c` function and the coefficients can be found in the `/FilterData` folder. The particular combination of filters cascaded together for a given sample rate change is specified in `ssrc.c` and `asrc.c`.

Table 2.4: SRC Filter Specifications

Filter	Fs (norm)	Pass-band	Stop-band	Ripple	Attenua-tion	Taps	Notes
BL	**2**	**0.454**	**0.546**	**0.01 dB**	**155 dB**	**144**	**Down-sampler by two, steep**
BL9644	2	0.417	0.501	0.01 dB	155 dB	160	Low-pass filter, steep for 96 to 44.1
BL8848	2	0.494	0.594	0.01 dB	155 dB	144	Low-pass, steep for 88.2 to 48
BLF	2	0.41	0.546	0.01 dB	155 dB	96	Low-pass at half band
BL19288	2	0.365	0.501	0.01 dB	155 dB	96	Low pass, steep for 192 to 88.2
BL17696	2	0.455	0.594	0.01 dB	155 dB	96	Low-pass, steep for 176.4 to 96
UP	2	0.454	0.546	0.01 dB	155 dB	144	Over sample by 2, steep
UP4844	2	0.417	0.501	0.01 dB	155 dB	160	Over sample by 2, steep for 48 to 44.1
UPF	2	0.41	0.546	0.01 dB	155 dB	96	Over sample by 2, steep for 176.4 to 192
UP192176	2	0.365	0.501	0.01 dB	155 dB	96	Over sample by 2, steep for 192 to 176.4
DS	4	0.57	1.39	0.01 dB	160 dB	32	Down sample by 2, relaxed
OS	2	0.57	1.39	0.01 dB	160 dB	32	Over sample by 2, relaxed
HS294	284	0.55	1.39	0.01 dB	155 dB	2352	Polyphase 147/160 rate change
HS320	320	0.55	1.40	0.01 dB	151 dB	2560	Polyphase 160/147 rate change
ADFIR	256	0.45	1.45	0.012 dB	170 dB	1920	Adaptive polyphase prototype filter

2.4 SRC File Structure and Overview

All source files for the SSRC and ASRC are located within the `multirate_hifi` subdirectory.

- src_mrhf_ssrc_wrapper.c / src_mrhf_ssrc_wrapper.h

  These wrapper files provide a simplified public API to the SSRC initialization and processing functions.

- src_mrhf_asrc_wrapper.c / src_mrhf_asrc_wrapper.h

  These wrapper files provide a simplified public API to the ASRC initialization and processing functions.

- src_mrhf_ssrc.c / src_mrhf_ssrc.h

  These files contain the core of the SSRC algorithm. They set up the correct filtering chains depending on rate change and apply in the processing calls. The table sFiltersIDs declared in SSRC.c contains definitions of the filter chains for all supported rate changes. The files also integrate the code for the optional dithering function.

- src_mrhf_asrc.c / src_mrhf_asrc.h

  These files contain the core of the ASRC algorithm. They setup the correct filtering chains depending on rate change and apply them for the corresponding processing calls. Note that filters F1, F2 and dithering are implemented using a block based approach similar to SSRC. The adaptive polyphase filter (ADFIR) is implemented on a sample by sample basis. These files also contain functions to compute the adaptive polyphase filter coefficients.

- src_mrhf_fir.c / src_mrhf_fir.h

  These files provide Finite Impulse Response (FIR) filtering setup, with calls to the assembler-optimized inner loops. They provide functions for handling down-sampling by 2, synchronous or over-sampling by 2 FIRs. They also provides functions for handling polyphase filters used for rational ratio rate change in the SSRC and adaptive FIR filters used in the asynchronous section of the ASRC.

- src_mrhf_filter_defs.c / src_mrhf_filter_defs.h

  These files define the size and coefficient sources for all the filters used by the SRC algorithms.

- /FilterData directory (various files)

  This directory contains the pre-computed coefficients for all of the fixed FIR filters. The numbers are stored as signed Q1.31 format and are directly included in the source of FilterDefs.c. Both the .dat files used by the C compiler and the .sfp ScopeFIR (http://iowegian.com/scopefir/) design source files, used to originally create the filters, are included.

- src_mrhf_fir_inner_loop_asm.S / src_mrhf_fir_inner_loop_asm.h

  Inner loop for the standard FIR function optimized for double-word load and store, 32 bit * 32 bit -> 64 bit MACC and saturation instructions. Even and odd sample long word alignment versions are provided.

- src_mrhf_fir_os_inner_loop_asm.S / scr_mrhf_fir_os_inner_loop_asm.h

  Inner loop for the oversampling FIR function optimized for double-word load and store, 32 bit * 32 bit -> 64 bit MACC and saturation instructions. Both (long word) even and odd sample input versions are provided.

- src_mrhf_spline_coeff_gen_inner_loop_asm.S / src_mrhf_spline_coeff_gen_inner_loop_asm.h

  Inner loop for generating the spline interpolated coefficients. This assembler function is optimized for double-word load and store, 32 bit * 32 bit -> 64 bit MACC and saturation instructions.

- src_mrhf_adfir_inner_loop_asm.S / src_mrhf_adfir_inner_loop_asm.h

  Inner loop for the adaptive FIR function using the previously computed spline interpolated coefficients. It is optimized for double-word load and store, 32 bit * 32 bit -> 64 bit MACC and saturation instructions. Both (long word) even and odd sample input versions are provided.

- src_mrhf_int_arithmetic.c / src_mrhf_int_arithmetic.h

  These files contain simulation implementations of XMOS ISA specific assembler instructions. These are only used for dithering functions, and may be eliminated during future optimizations.

2.5 SSRC API

void **`ssrc_init`**(const fs_code_t sr_in, const fs_code_t sr_out, ssrc_ctrl_t ssrc_ctrl[], const unsigned n_channels_per_instance, const unsigned n_in_samples, const dither_flag_t dither_on_off)

initializes synchronous sample rate conversion instance.

Parameters

- **`sr_in`** – Nominal sample rate code of input stream
- **`sr_out`** – Nominal sample rate code of output stream
- **`ssrc_ctrl`** – Reference to array of SSRC control stuctures
- **`n_channels_per_instance`** – Number of channels handled by this instance of SSRC
- **`n_in_samples`** – Number of input samples per SSRC call
- **`dither_on_off`** – Dither to 24b on/off

unsigned **`ssrc_process`**(int in_buff[], int out_buff[], ssrc_ctrl_t ssrc_ctrl[])

Perform synchronous sample rate conversion processing on block of input samples using previously initialized settings.

Parameters

- **`in_buff`** – Reference to input sample buffer array
- **`out_buff`** – Reference to output sample buffer array
- **`ssrc_ctrl`** – Reference to array of SSRC control stuctures

Returns

The number of output samples produced by the SRC operation

2.6 ASRC API

uint64_t **`asrc_init`**(const fs_code_t sr_in, const fs_code_t sr_out, asrc_ctrl_t asrc_ctrl[], const unsigned n_channels_per_instance, const unsigned n_in_samples, const dither_flag_t dither_on_off)

initializes asynchronous sample rate conversion instance.

Parameters

- **`sr_in`** – Nominal sample rate code of input stream
- **`sr_out`** – Nominal sample rate code of output stream
- **`asrc_ctrl`** – Reference to array of ASRC control structures
- **`n_channels_per_instance`** – Number of channels handled by this instance of SSRC
- **`n_in_samples`** – Number of input samples per SSRC call
- **`dither_on_off`** – Dither to 24b on/off

Returns

The nominal sample rate ratio of in to out in Q4.60 format unsigned **`asrc_process`**(int in_buff[], int out_buff[], uint64_t fs_ratio, asrc_ctrl_t asrc_ctrl[])

Perform asynchronous sample rate conversion processing on block of input samples using previously initialized settings.

Parameters

- **`in_buff`** – Reference to input sample buffer array
- **`out_buff`** – Reference to output sample buffer array
- **`fs_ratio`** – Fixed point ratio of in/out sample rates in Q4.60 format
- **`asrc_ctrl`** – Reference to array of ASRC control structures

Returns

The number of output samples produced by the SRC operation.

3 Fixed factor of 3 HiFi functions

3.1 Overview

The SRC library includes synchronous sample rate conversion functions to downsample (decimate) and oversample (upsample or interpolate) by a fixed factor of 3.

These components offer a high quality conversion with an SNR of 130 dB.

In each case, the processing is carried out each time a single output sample is required. In the case of the decimator, three input samples are passed to the filter with a resulting one sample output on calling the processing function. The interpolator produces an output sample each time the processing function is called but will require a single sample to be pushed into the filter every third cycle. All samples use Q1.31 format (left justified signed 32b integer).

Both sample rate converters are based on a 144 tap FIR filter with two sets of coefficients available, depending on application requirements:

- firos3_b_144.dat / firds3_b_144.dat - These filters have 20 dB of attenuation at the Nyquist frequency and a higher cutoff frequency
- firos3_144.dat / firds3_144.dat - These filters have 60 dB of attenuation at the Nyquist frequency but trade this off with a lower cutoff frequency

The default setting is to use the coefficients that provide 60 dB of attenuation at the Nyquist frequency.

The filter coefficients may be selected by adjusting the line:

```
#define   FIROS3_COEFS_FILE
```

and:

```
#define   FIRDS3_COEFS_FILE
```

in the files `src_ff3_os3.h` (API for oversampling) and `src_ff3_ds3.h` (API for downsampling) respectively.

The OS3 processing takes up to 153 core cycles to compute a sample which translates to 1.53 μs at 100 MHz or 2.448 μs at 62.5 MHz core speed. This permits up to 8 channels of 16 kHz -> 48 kHz sample rate conversion in a single 62.5MHz core.

The DS3 processing takes up to 389 core cycles to compute a sample which translates to 3.89 μs at 100 MHz or 6.224 μs at 62.5 MHz core speed. This permits up to 9 channels of 48 kHz -> 16 kHz sample rate conversion in a single 62.5MHz core.

Both downsample and oversample functions return `ERROR` or `NO_ERROR` status codes as defined in the return code enums listed below. The only way these functions can error is if the passed *delay_base* structure member is uninitialized (NULL).

The downsampling functions return the following error codes

```
FIRDS3_NO_ERROR
FIRDS3_ERROR
```

The upsampling functions return the following error codes

```
FIROS3_NO_ERROR
FIROS3_ERROR
```

3.2 API

enum **`src_ff3_return_code_t`**

Fixed factor of 3 return codes

This type describes the possible error status states from calls to the DS3 and OS3 API.

Values:

enumerator **`SRC_FF3_NO_ERROR`**

enumerator **`SRC_FF3_ERROR`**

3.3 DS3 API

struct **`src_ds3_ctrl_t`**

Downsample by 3 control structure

src_ff3_return_code_t **`src_ds3_init`**(*src_ds3_ctrl_t* *src_ds3_ctrl)

This function initializes the decimate by 3 function for a given instance

Parameters

- **`src_ds3_ctrl`** – DS3 control structure

Returns

SRC_FF3_NO_ERROR on success, SRC_FF3_ERROR on failure

src_ff3_return_code_t **`src_ds3_sync`**(*src_ds3_ctrl_t* *src_ds3_ctrl)

This function clears the decimate by 3 delay line for a given instance

Parameters

- **`src_ds3_ctrl`** – DS3 control structure

Returns

SRC_FF3_NO_ERROR on success, SRC_FF3_ERROR on failure

src_ff3_return_code_t **`src_ds3_proc`**(*src_ds3_ctrl_t* *src_ds3_ctrl)

This function performs the decimation on three input samples and outputs one sample. The input and output buffers are pointed to by members of the src_ds3_ctrl structure

Parameters

- **`src_ds3_ctrl`** – DS3 control structure

Returns

SRC_FF3_NO_ERROR on success, SRC_FF3_ERROR on failure

3.4 OS3 API

struct **`src_os3_ctrl_t`**

Oversample by 3 control structure

src_ff3_return_code_t **`src_os3_init`**(*src_os3_ctrl_t* *src_os3_ctrl)

This function initializes the oversample by 3 function for a given instance

Parameters

- **`src_os3_ctrl`** – OS3 control structure

Returns

SRC_FF3_NO_ERROR on success, SRC_FF3_ERROR on failure

src_ff3_return_code_t **`src_os3_sync`**(*src_os3_ctrl_t* *src_os3_ctrl)

This function clears the oversample by 3 delay line for a given instance

Parameters

- **`src_os3_ctrl`** – OS3 control structure

Returns

SRC_FF3_NO_ERROR on success, SRC_FF3_ERROR on failure

src_ff3_return_code_t **`src_os3_input`**(*src_os3_ctrl_t* *src_os3_ctrl)

This function pushes a single input sample into the filter. It should be called three times for each FIROS3_proc call

Parameters

- **`src_os3_ctrl`** – OS3 control structure

Returns

SRC_FF3_NO_ERROR on success, SRC_FF3_ERROR on failure

src_ff3_return_code_t **`src_os3_proc`**(*src_os3_ctrl_t* *src_os3_ctrl)

This function performs the oversampling by 3 and outputs one sample. The input and output buffers are pointed to by members of the src_os3_ctrl structure

Parameters

- **`src_os3_ctrl`** – OS3 control structure

Returns

SRC_FF3_NO_ERROR on success, SRC_FF3_ERROR on failure

4 Fixed factor of 3 functions optimized for use with voice

4.1 Overview

A pair of SRC components supporting upconversion and downconversion by a factor of 3 are provided that are suitable for voice applications. They provide voice quality SNR (around 60 dB) and use a 72 tap Remez FIR filter and are optimized for the XS2 instruction set.

Warning: These SRC components have been deprecated. For new designs using XCORE-AI, please use the XS3 optimized components which provide both much better performance and use approximately half of the MIPS. See *ff3_voice_vpu_hdr*

4.2 Voice DS3 API

int64_t **`src_ds3_voice_add_sample`**(int64_t sum, int32_t data[], const int32_t coefs[], int32_t sample)

This function performs the first two iterations of the downsampling process

Parameters

- **`sum`** – Partially accumulated value returned during previous cycle
- **`data`** – Data delay line
- **`coefs`** – FIR filter coefficients
- **`sample`** – The newest sample

Returns

Partially accumulated value, passed as `sum` parameter next cycle

int64_t **`src_ds3_voice_add_final_sample`**(int64_t sum, int32_t data[], const int32_t coefs[], int32_t sample)

This function performs the final iteration of the downsampling process

Parameters

- **`sum`** – Partially accumulated value returned during previous cycle
- **`data`** – Data delay line
- **`coefs`** – FIR filter coefficients
- **`sample`** – The newest sample

Returns

The decimated sample

4.3 Voice US3 API

int32_t **`src_us3_voice_input_sample`**(int32_t data[], const int32_t coefs[], int32_t sample)

This function performs the initial iteration of the upsampling process

Parameters

- **`data`** – Data delay line
- **`coefs`** – FIR filter coefficients
- **`sample`** – The newest sample

Returns

A decimated sample

int32_t **`src_us3_voice_get_next_sample`**(int32_t data[], const int32_t coefs[])

This function performs the final two iterations of the upsampling process

Parameters

- **`data`** – Data delay line
- **`coefs`** – FIR filter coefficients

Returns

A decimated sample

5 Fixed factor of 3 and 3/2 voice functions optimized for XS3

5.1 Overview

A set of SRC components are provided which are optimized for the Vector Processing Unit (VPU) and are suitable for voice applications. The fixed factor of 3 SRC components are designed for conversion between 48 kHz to 16 kHz and the fixed factor of 3/2 are designed for conversion between 48 kHz and 32 kHz.

They have been designed for voice applications and, in particular, conformance to the MS Teams v5 specification.

Note: These filters will only run on XCORE-AI due to the inner dot product calculation employing the XS3 VPU.

Warning: Synchronous fixed factor of 3 and 3/2 downsample and oversample functions for voice applications optimized for the XS3 Vector Processing Unit currently overflow rather than saturate in cases where a full scale input causes a perturbation above full scale at the output. To avoid this scenario, please ensure that the input amplitude is always 3.5 dB below full scale. The overflow behavior of these SRC components will be replaced by saturating behavior (to match all other SRC components) in a future release.

5.2 Fixed factor of 3 VPU

The filters use less than half of the cycles of the previous fixed factor of 3 functions but at the same time offer a much improved filter response thanks to an increased filter length of 96 taps (compared with 72 taps) and use of a Kaiser window with a beta of 4.0. The filter specification is shown in Table 5.1.

Table 5.1: Fixed Factor of 3 Voice VPU SRC characteristics

Filter	CPU cycles	Passband	Stopband	Ripple	Attenuation	Taps
src_ff3_96t_ds	104	0.475	0.525	0.01 dB	70 dB min	96
src_ff3_96t_us	85	0.475	0.525	0.01 dB	70 dB min	96

The fixed factor of 3 components produce three samples for each call passing one sample in the case of upsampling and produce a single sample for each call passing three samples in the case of downsampling. All input and output samples are signed 32 bit integers. The filter characteristics are shown in Fig. 5.1 and Fig. 5.2.

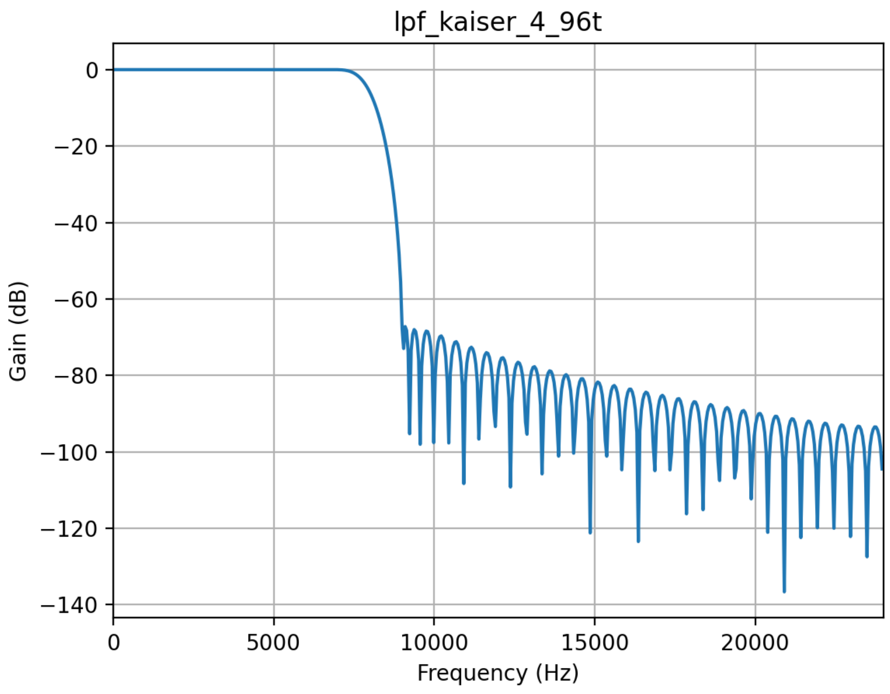

Fig. 5.1: Fixed Factor of 3 Voice VPU SRC filter response

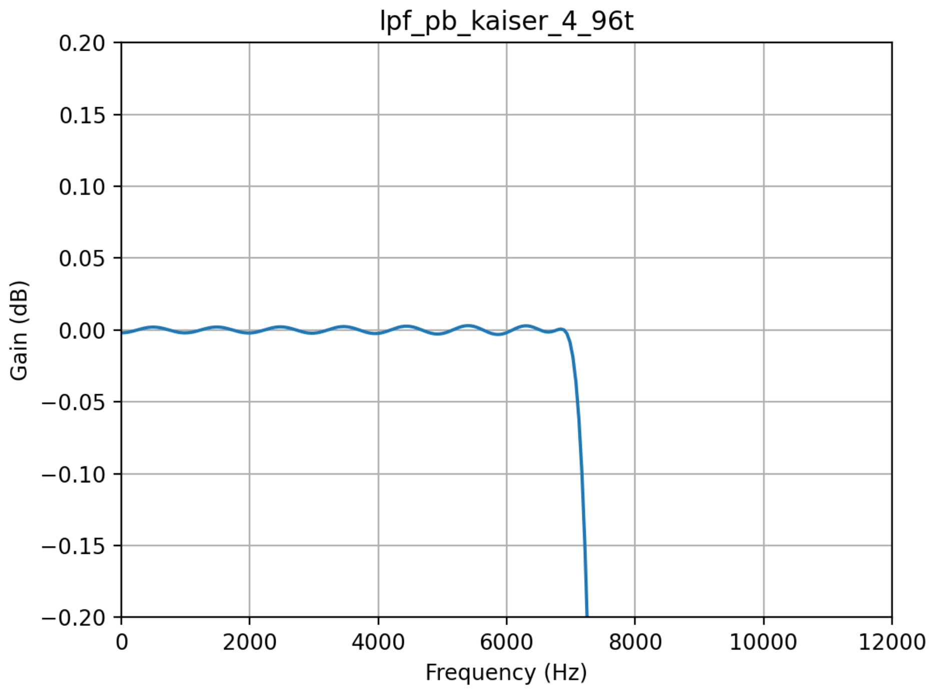

Fig. 5.2: Fixed Factor of 3 Voice VPU SRC passband ripple

5.3 Voice DS3 API

static inline void **`src_ff3_96t_ds`**(int32_t samp_in[3], int32_t samp_out[1], const int32_t coefs_ff3[3][32], int32_t state_ds[3][32])

Performs VPU-optimised 96 taps polyphase fixed-factor-of-3 downsampling.

Parameters

- **`samp_in`** – Values to be downsampled
- **`samp_out`** – Downsampled output
- **`coefs_ff3`** – Three-phase FIR coefficients array with [3][32] dimensions
- **`state_ds`** – Three-phase FIR state array with [3][32] dimensions

5.4 Voice US3 API

static inline void **`src_ff3_96t_us`**(int32_t samp_in[1], int32_t samp_out[3], const int32_t coefs_ff3[3][32], int32_t state_us[32])

Performs VPU-optimised 96 taps polyphase fixed-factor-of-3 upsampling.

Note: samp_in and samp_out have to be different memory locations

Parameters

- **`samp_in`** – Value to be upsampled
- **`samp_out`** – Upsampled output
- **`coefs_ff3`** – Three-phase FIR coefficients array with [3][32] dimensions
- **`state_us`** – FIR state array with 32 elements in it

5.5 Fixed factor of 3/2 VPU

The fixed factor of 3/2 VPU sample rate converts use a rational factor polyphase architecture to achieve the non-integer rate ratio. Downsampling takes two phases while upsampling takes three. The filters have been designed with a Kaiser window with a beta of 3.2. The filter specification is shown in Table 5.2.

Table 5.2: Fixed Factor of 3/2 Voice VPU SRC characteristics

Filter	CPU cycles	Passband	Stopband	Ripple	Attenuation	Taps
src_rat_2_3_96t_ds	112	0.46875	0.53125	0.03 dB	70 dB	96
src_rat_3_2_96t_us	95	0.46875	0.53125	0.03 dB	70 dB	96

The fixed factor of 3/2 components produce three samples for each call passing two samples in the case of upsampling and produce two samples for each call passing three samples in the case of downsampling. All input and output samples are signed 32 bit integers. The filter characteristics are shown in Fig. 5.3 and Fig. 5.4.

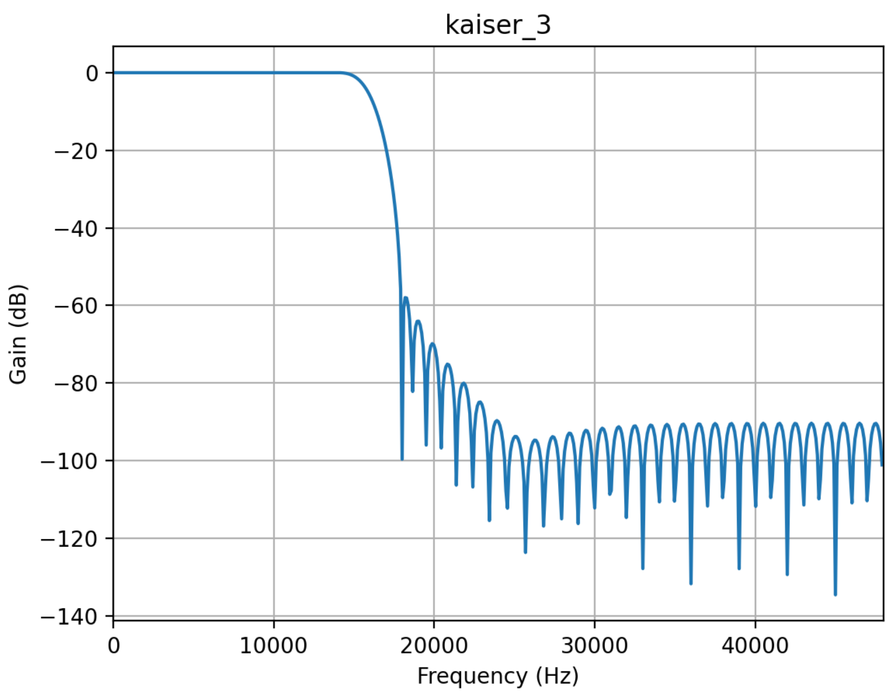

Fig. 5.3: Fixed Factor of 3/2 Voice VPU SRC filter response

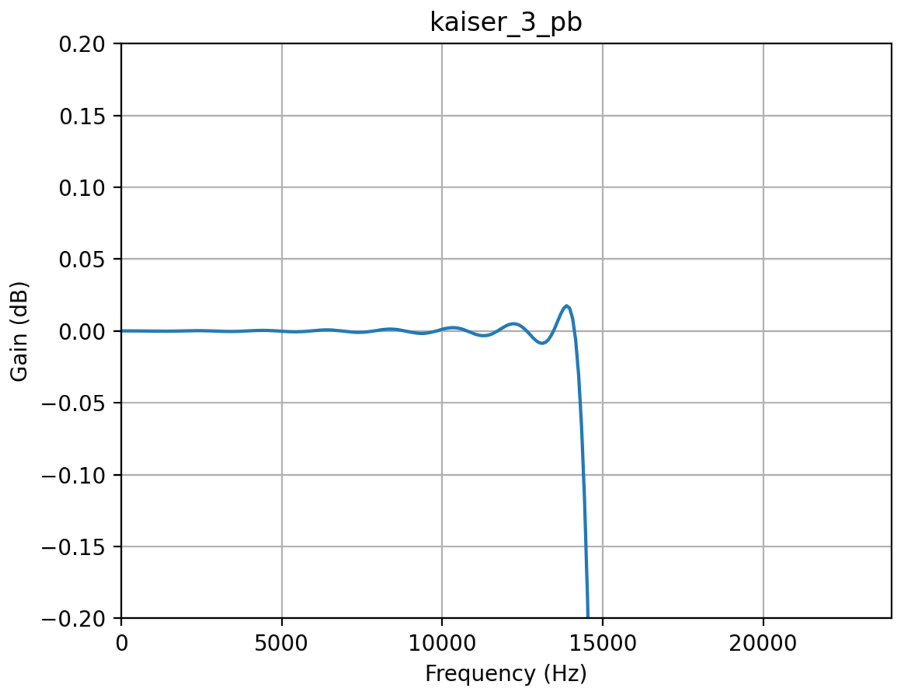

Fig. 5.4: Fixed Factor of 3/2 Voice VPU SRC passband ripple

5.6 Voice DS3/2 API

static inline void **`src_rat_2_3_96t_ds`**(int32_t samp_in[3], int32_t samp_out[2], const int32_t coefs_ds[2][48], int32_t state_ds[48])

Performs VPU-optimised 96 taps polyphase rational factor 2/3 downsampling.

Parameters

- **`samp_in`** – Values to be downsampled
- **`samp_out`** – Downsampled output
- **`coefs_ds`** – Two-phase FIR coefficients array with [2][48] dimensions
- **`state_ds`** – FIR state array with 48 elements in it

5.7 Voice US3/2 API

static inline void **`src_rat_3_2_96t_us`**(int32_t samp_in[2], int32_t samp_out[3], const int32_t coefs_us[3][32], int32_t state_us[32])

Performs VPU-optimised 96 taps polyphase rational factor 3/2 upsampling.

Note: samp_in and samp_out have to be different memory locations

Parameters

- **`samp_in`** – Values to be upsampled
- **`samp_out`** – Upsampled output
- **`coefs_us`** – Three-phase FIR coefficients array with [3][32] dimensions
- **`state_us`** – FIR state array with 32 elements in it

6 lib_src change log

6.1 2.4.0

- ADDED: Support for building the core ASRC code in the C emulator as a library
- ADDED: Auto-generated ASRC and SSRC performance plots in documentation
- CHANGED: Documents built under Jenkins instead of Github Actions
- ADDED: Documentation warning about overflow in XS3 optimized SRC components
- CHANGED: Tested against fwk_core v1.0.2 updated from v1.0.0

6.2 2.3.0

- ADDED: XS3 VPU optimised voice fixed factor of 3 upsampling/downsampling
- ADDED: XS3 VPU optimised voice fixed factor of 3/2 upsampling/downsampling
- CHANGED: OS3 uses firos3_144.dat coefficients by default inline with model
- CHANGED: Replaced xmostest with pytest for all SRC automated tests
- CHANGED: Used XMOS doc builder for documentation
- CHANGED: Golden reference test signals now generated automatically by CI
- RESOLVED: Linker warning on channel ends
- REMOVED: AN00231 ASRC App Note. See github.com/xmos/sln_voice/examples
- CHANGED: Increased precision of the fFsRatioDeviation used in the C emulator from float to double
- CHANGED: Allow for 64 bits in the rate ratio passed to asrc_process() for extra precision
- Changes to dependencies:
  - lib_logging: 2.0.1 -> 3.1.1
  - lib_xassert: 2.0.1 -> 4.1.0

6.3 2.2.0

- CHANGED: Made the FIR coefficient array that is used with the voice fixed factor of 3 up and down sampling functions usable from within C files as well as XC files.
- CHANGED: Aligned the FIR coefficient array to an 8-byte boundary. This ensures that the voice fixed factor of 3 up and down sampling functions do not crash with a LOAD_STORE exception.
- ADDED: Missing device attributes to the .xn file of the AN00231 app note.
- ADDED: Minimal cmake support.

6.4 2.1.0

- CHANGED: Use XMOS Public License Version 1

6.5 2.0.1

- CHANGED: Pin Python package versions
- REMOVED: not necessary cpanfile

6.6 2.0.0

- CHANGED: Build files updated to support new "xcommon" behavior in xwaf.

6.7 1.1.2

- CHANGED: initialisation lists to avoid warnings when building

6.8 1.1.1

- RESOLVED: correct compensation factor for voice upsampling
- ADDED: test of voice unity gain

6.9 1.1.0

- ADDED: Fixed factor of 3 conversion functions for downsampling and oversampling
- ADDED: Fixed factor of 3 downsampling function optimised for use with voice (reduced memory and compute footprint)
- ADDED: Fixed factor of 3 upsampling function optimised for use with voice (reduced memory and compute footprint)

6.10 1.0.0

- Initial version
- Changes to dependencies:
  - lib_logging: Added dependency 2.0.1
  - lib_xassert: Added dependency 2.0.1