Sample Rate Conversion Library

The XMOS Sample Rate Conversion (SRC) library provides both synchronous and asynchronous audio sample rate conversion functions for use on xCORE-200 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 the are multiple digital audio inputs or allow cost saving by removing the need for physical clock recovery using a PLL.

Features

- Conversion between 44.1, 48, 88.2, 96, 176.4 and 192KHz 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
- 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 135db (ASRC) or 140db (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

Components

- Synchronous Sample Rate Converter function
- Asynchronous Sample Rate Converter function

Software version and dependencies

This document pertains to version 1.0.0 of this library. It is known to work on version 14.2.0 of the xTIMEcomposer tools suite, it may work on other versions.

This library depends on the following other libraries:

- lib_logging (>=2.0.0)
- lib_xassert (>=2.0.0)

Related application notes

The following application notes use this library:

- AN00230 - [Adding Synchronous Sample Rate Conversion to the USB Audio reference design]
- AN00231 - [SPDIF receive to I^2S output using Asynchronous Sample Rate Conversion]

Typical Resource Usage

This following table shows typical resource usage in some different configurations. Exact resource usage will depend on the particular use of the library by the application.

Configuration	Pins	Ports	Clocks	Ram	Logical cores
SSRC	0	0	0	~30.5K	1

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 44.1KHz family of sample rates to the 48KHz family of sample rates.

The below table 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.

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

Table 1: SSRC Processor Usage per Channel (MHz)

This following table shows typical resource usage in some different configurations. Exact resource usage will depend on the particular use of the library by the application.

Configuration	Pins	Ports	Clocks	Ram	Logical cores
ASRC	0	0	0	~28.5K	1

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; the first table enumerates the MHz required for the first channel with adaptive coefficients calculation and the second table specifies the MHz required for filtering of each additional channel processed by the ASRC instance.

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.

Typically you will need to allow for performance headroom for buffering (especially if the system is sample orientated rather than block orientated) and inter-task communication. Please refer to the application notes for practical examples of usage.

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

Table 2: ASRC Processor Usage (MHz) for the First Channel in the ASRC Instance

Configurations requiring more than 100MHz cannot currently be run in real time on a single core. The performance limit for a single core on a 500MHz xCORE-200 device is 100MHz (500/5). Further optimization of the library, including assembler optimization and pipelining of the adaptive filter generation and FIR filter stages, is feasible to achieve higher sample rate operation within the constraints of a 100MHz logical core.

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

Table 3: ASRC Processor Usage (MHz) for Subsequent Channels in the ASRC Instance

1 Usage

Both SSRC and ASRC functions are accessed via a 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 your processing or placed on a logical core within it's own task to provide guaranteed performance. By placing the calls to SRC functions on sepearte logical cores, multiple instances can be processed concurrently.

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

1.1 Initialization

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 must be called to ensure the correct selection and 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:

```
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);
```

The initialization call is the same for ASRC:

```
unsigned 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);
```

The settings include:

- Nominal input sample rate as an enumerated type
- Nominal output sample rate as an enumerated type
- The number of channels to be handled by this instance of SRC
- The number of input samples to expect. Minimum 4 samples input per call, must be power of 2
- The dither setting. Dithers the output from 32bit to 24bit

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`.

There are a number of arrays of structures that must be declared from the application which contain the state, 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 the case of 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 the ASRC, the state structures must be declared. 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;
```

1.2 Processing

Following initialization, the processing API is called for each block of input samples. 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.

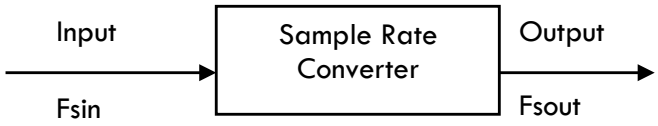

Figure 1: SRC Operation

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

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

In the case of ASRC, additionally a fractional frequency ratio is supplied:

```
unsigned asrc_process(int *in_buff, int *out_buff, unsigned fs_ratio, asrc_ctrl_t asrc_ctrl[])
```

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 44.1KHz and the output rate is 192KHz 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 internally 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 44.1KHz to 48KHz 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 44.1KHz to 48KHz, 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 the ASRC must be used.

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, you can see that the samples are interleaved into a single buffer of size 8.

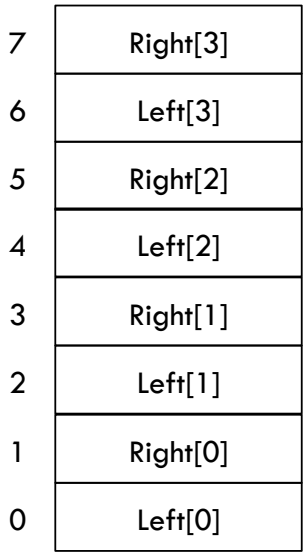

Figure 2: Buffer Format for Single Stereo SRC instance

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.

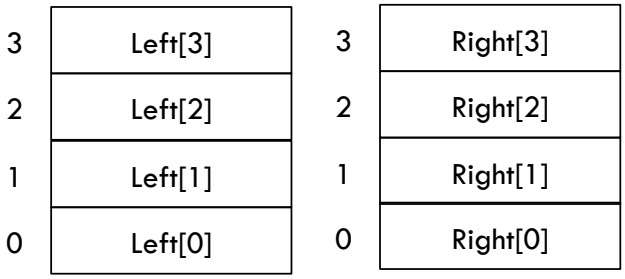

Figure 3: Buffer Format for Dual Mono SRC instances

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. 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.

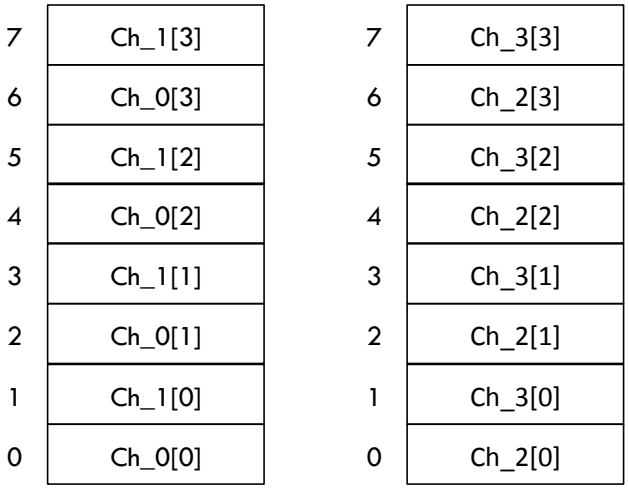

Figure 4: Buffer Format for Dual Stereo SRC instances (4 channels total)

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. An example of this calculation, based on measuring the input and output rates, is provided in AN00231.

Further detail about these function arguments are contained within the API section of this guide.

2 SSRC Performance

The performance of the SSRC library is as follows:

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

The performance was analyzed by converting output test files to 32 bits integer wav files. These files were then run through an audio analysis tool (WinAudio MLS: http://www.dr-jordan-design.de/Winaudiomls.htm).

Below are a series FFT plots showing the most demanding rate conversion case. These clearly show that the above targets are comfortably exceeded. All outputs have been generated using 8192 samples at input sampling rate. A Kaiser-Bessel window with alpha=7 has been used.

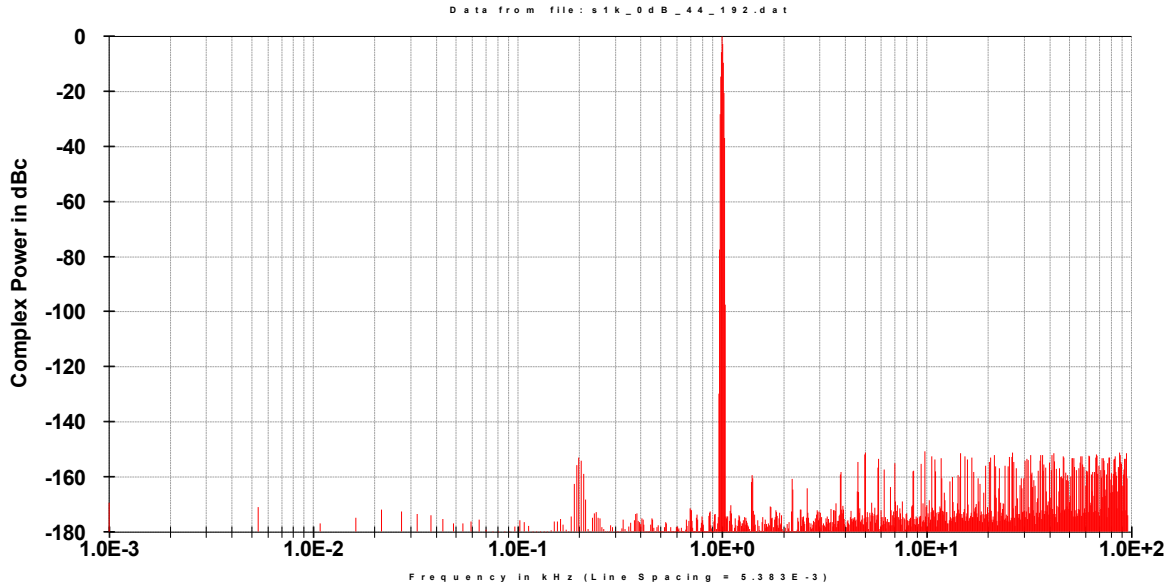

Figure 5: FFT of 1kHz sine, 0dB, 44.1kHz to 192kHz

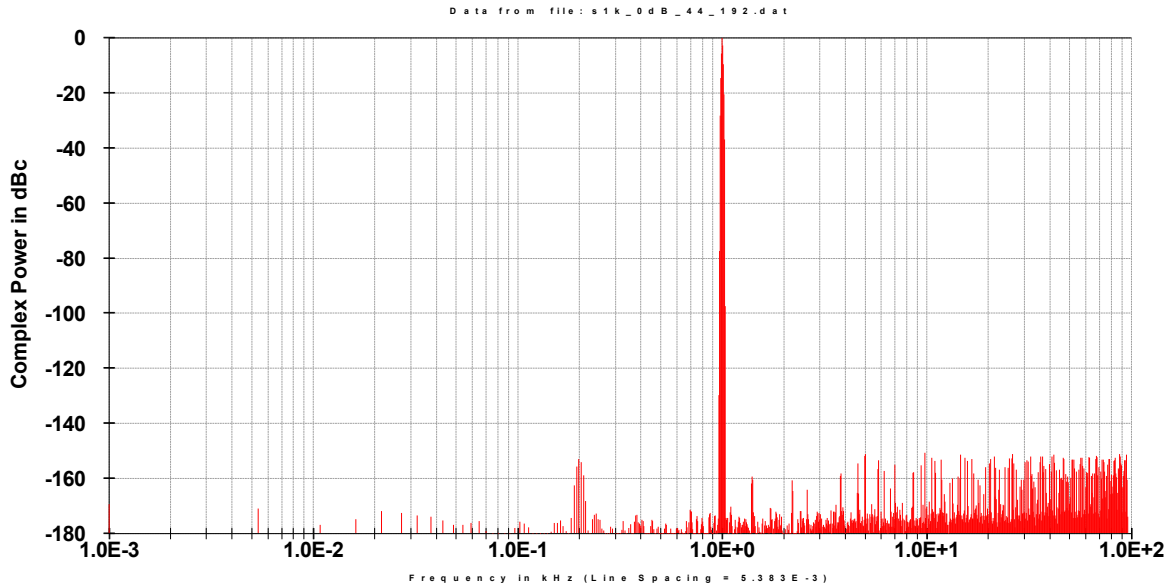

Figure 6: FFT of 1kHz sine, 0dB, 176.4kHz to 48kHz

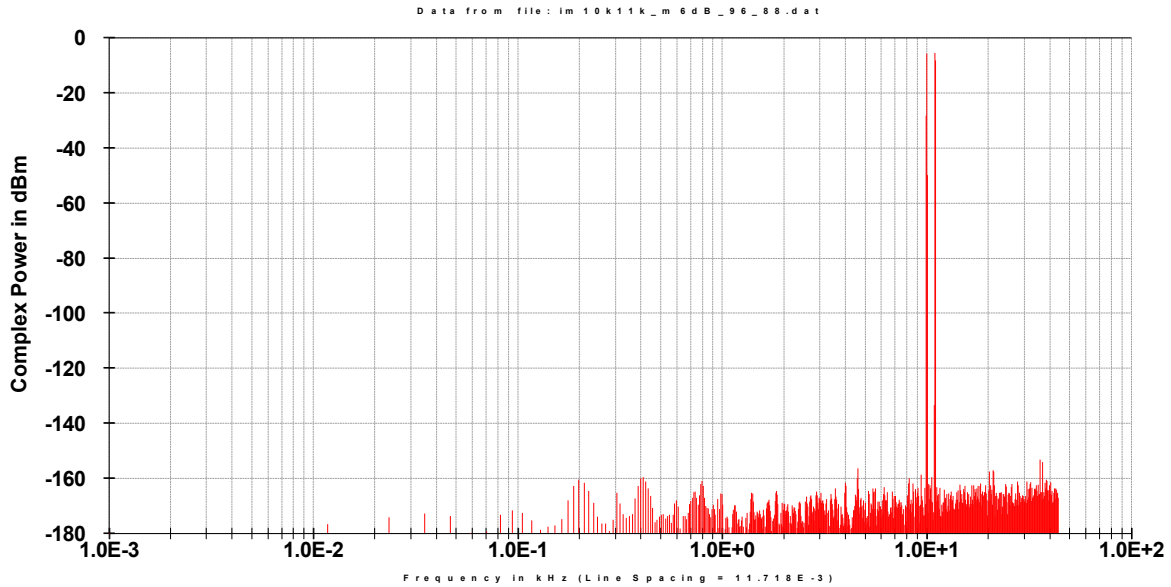

Figure 7: FFT of 10kHz+11kHz sines, -6dB, 96kHz to 88.2kHz

3 ASRC Performance

The performance of the SSRC library is as follows:

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

The performance was analyzed by converting output test files to 32 bits integer wav files. These files were then run through an audio analysis tool (WinAudio MLS: http://www.dr-jordan-design.de/Winaudiomls.htm).

Below are a series FFT plots showing the most demanding rate conversion case. These clearly show that the above targets are comfortably exceeded. All outputs have been generated using 8192 samples at input sampling rate. A Kaiser-Bessel window with alpha=7 has been used.

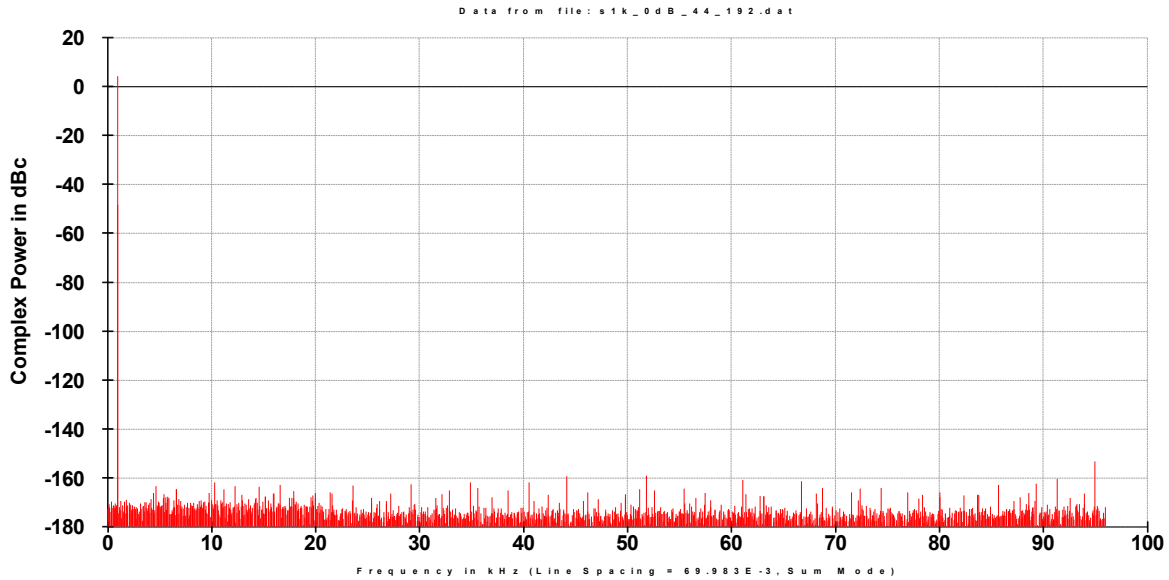

Figure 8: FFT of 1kHz sine, 0dB, 44.1kHz to 192kHz

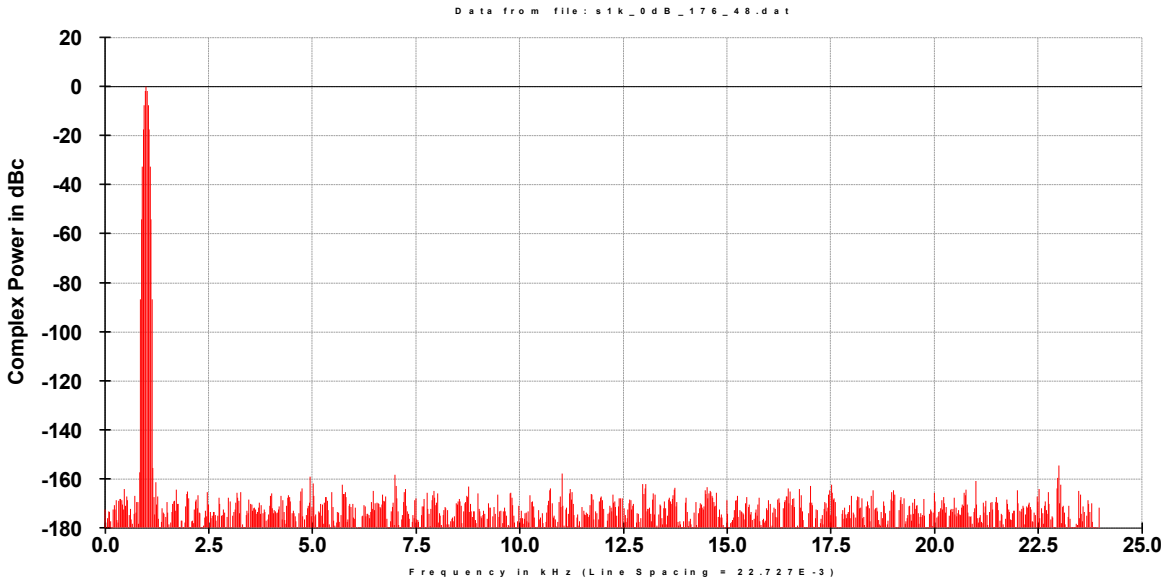

Figure 9: FFT of 1kHz sine, 0dB, 176.4kHz to 48kHz

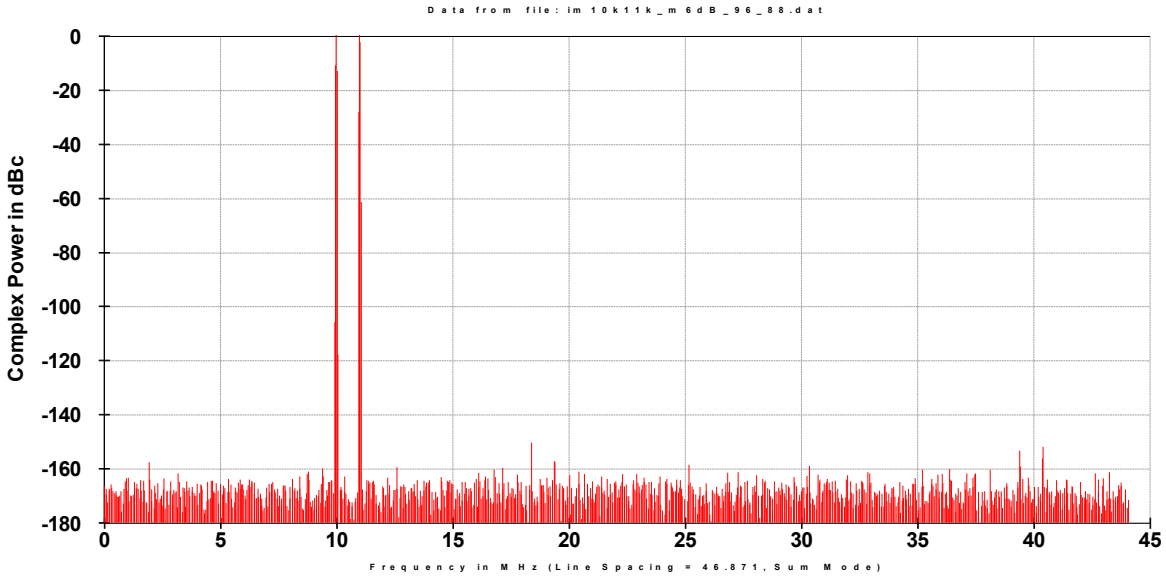

Figure 10: FFT of 10kHz+11kHz sines, -6dB, 96kHz to 88.2kHz

4 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.

4.1 SRC Nominal Rate Changes

The nominal rate change ratios between 44.1KHz and 192KHz are shown in the below table.

	Output sample rate					
Input sample rate	**44.1KHz**	**48KHz**	**88.2KHz**	**96KHz**	**176.4KHz**	**192KHz**
44.1KHz	1	160/147	2	2x160/147	4	4x160/147
48KHz	147/160	1	2x147/160	2	4x147/160	4
88.2KHz	1/2	1/2x160/147	1	160/147	2	2x160/147
96KHz	1/2x147/160	1/2	147/160	1	2x147/160	2
176.4KHz	1/4	1/4x160/147	1/2	1/2x160/147	1	160/147
192KHz	1/4x147/160	1/4	1/2x147/160	1/2	147/160	1

Table 4: Rate Changes for Sample Rate Conversion

The table shows the case for SSRC where the ratios are equal to the nominal sample rate ratio. In the case of ASRC, where the ratios cannot be expressed rationally, these are the nominal ratios from which there will usually be a rate deviaton.

4.2 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. The following diagram shows an overall view of the SSRC algorithm:

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 effectively converts between the 44.1kHz and the 48kHz families of sample rates.

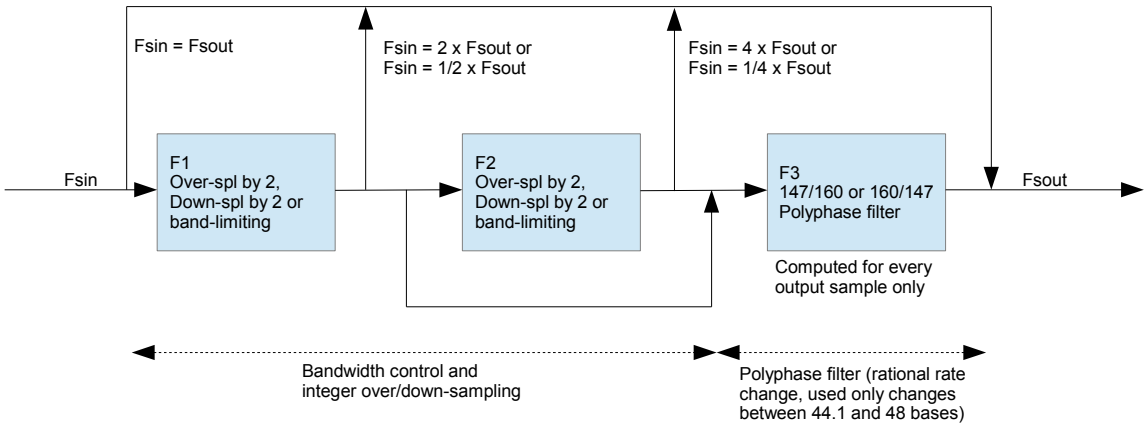

Figure 11: SSRC Algorithm Structure

4.3 ASRC Structure

Similar to the SSRC, the ASRC algorithm is based 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. The following diagram shows an overall view of the ASRC algorithm:

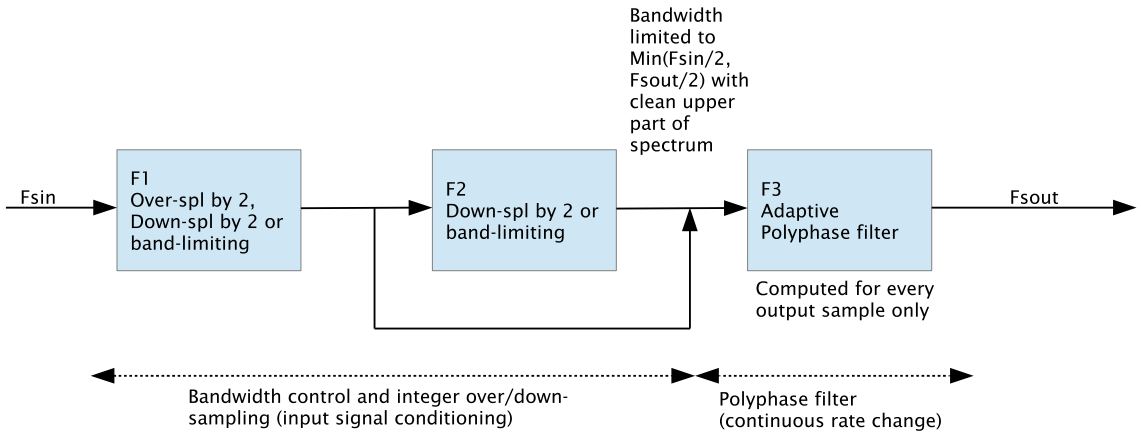

Figure 12: ASRC Algorithm Structure

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 (to min(Fsin/2,Fsout/2) 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.

4.4 SRC Filter list

A complete list of the filters supported by the SRC library, both SSRC and ASRC, is shown in the below table. 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`.

Filter	Fs (norm)	Passband	Stopband	Ripple	Attenuation	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

Table 5: SSRC Processor Usage per channel (MHz)

5 SRC File Structure and Overview

- ssrc_wrapper.c / ssrc_wrapper.h
  These wrapper files provide a simplified public API to the SSRC initialization and processing functions.
- asrc_wrapper.c / asrc_wrapper.h
  These wrapper files provide a simplified public API to the ASRC initialization and processing functions.
- SSRC.c / SSRC.h
  These files contain the core of the SSRC algorithm. It sets up the correct filtering chains depending on rate change and applies them in the processing calls. The table sFiltersIDs declared in SSRC.c contains definitions of the filter chains for all supported rated changes. The files also integrate the code for the optional dithering function.
- ASRC.c / 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 (code 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 poly-phase filter coefficients.
- FIR.c / FIR.h
  These files provide Finite Impulse Response (FIR) filtering setup, with calls to the assembler-optimized inner loops. It provides functions for handling down-sampling by 2, synchronous or over-sampling by 2 FIRs. It 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.
- FilterDefs.c / FilterDefs.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 into the source from 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.
- fir_inner_loop_asm.S / fir_inner_loop_asm.h
  Inner loop for the standard FIR function optimized for double-word load and store, 32bit * 32bit -> 64bit MACC and saturation instructions. Even and odd sample long word alignment versions are provided.
- fir_os_inner_loop_asm.S / fir_os_inner_loop_asm.h
  Inner loop for the oversampling FIR function optimized for double-word load and store, 32bit * 32bit -> 64bit MACC and saturation instructions. Both (long word) even and odd sample input versions are provided.
- spline_coeff_gen_inner_loop_asm.S / 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, 32bit * 32bit -> 64bit MACC and saturation instructions.
- adfir_inner_loop_asm.S / 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, 32bit * 32bit -> 64bit MACC and saturation instructions. Both (long word) even and odd sample input versions are provided.
- IntArithmetic.c / IntArithmetic.h
  These files contain simulation implementations of following XMOS assembler instructions. These are only used for dithering functions, and may be eliminated during future optimizations.

6 SSRC API

All public SSRC functions are prototyped within the `src.h` header:

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

Please ensure that you have reviewed the settings within `src_config.h` and they are correct for your application. The default settings allow for any input/output ratio between 44.1KHz and 192KHz.

You will also have to add `lib_src` to the `USED_MODULES` field of your application Makefile.

6.1 Initialization

Function	**ssrc_init**
Description	Initialises synchronous sample rate conversion instance.

Type	`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)`

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

6.2 SSRC Processing

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

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

Continued on next page

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	

7 ASRC API

7.1 Initialization

Function	asrc_init
Description	Initialises asynchronous sample rate conversion instance.

Type	`unsigned 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)`

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 Q4.28 format

7.2 ASRC Processing

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

Type	`unsigned asrc_process(int in_buff[],`
	`  int out_buff[],`
	`  unsigned fs_ratio,`
	`  asrc_ctrl_t asrc_ctrl[])`

Continued on next page

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 Q4.28 format
	`asrc_ctrl`	Reference to array of ASRC control structures
Returns	The number of output samples produced by the SRC operation.	

APPENDIX A - Known Issues

Certain ASRC configurations, mainly conversions between 176.4/192KHz to 176.4/192KHz, require greater than 100MHz for a single audio channel and so cannot currently be run in real time on a single core. The performance limit for a single core on a 500MHz xCORE-200 device is 100MHz (500/5), due to a 5 stage pipeline. A number of potential optimizations have been identified to permit these rates:

- Further inner loop optimization using assembler
- Increase in scope of assembler sections removing additional function calls
- Pipelining of the FIR filter stages into separate tasks
- Calculation of adaptive filter coefficients in a separate task

These optimizations may be the target for future revisions of this library.

APPENDIX B - lib_src change log

B.1 1.0.0

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