A High-Quality, Energy Optimized, Real-Time Sampling Rate Conversion Library for the StrongARM Microprocessor

Chung-Tse Mar[1], Mat C. Hans, Mark T. Smith, Tajana Simunic, Ronald W. Schafer[1]
Mobile& Media Systems Laboratory
HP Laboratories Palo Alto
HPL-2002-159
June 3rd , 2002*

digital signal processing, sample rate conversion, audio, StrongARM,

This paper presents a digital audio sampling rate conversion library that converts between arbitrary sampling frequencies. The library is implemented as a time-varying fractional delay filter with coefficients stored in a lookup table. It is designed for use by real-time applications and optimized for execution on Intel's StrongARM microprocessor.

[1] Center for Signal & Image Processing, Georgia Institute of Technology, Atlanta, GA

A High-Quality, Energy Optimized, Real-Time Sampling Rate Conversion Library for the StrongARM Microprocessor

Chung-Tse Mar, Mat C. Hans, Mark T. Smith, Tajana Simunic, Ronald W. Schafer[*]

Abstract

This paper presents a digital audio sampling rate conversion library that converts between arbitrary sampling frequencies. The library is implemented as a time-varying fractional delay filter with coefficients stored in a lookup table. It is designed for use by real-time applications and optimized for execution on Intel's StrongARM microprocessor.

1. Introduction

Many different sampling rates are used in today's digital signal processing applications. Telephone systems sample speech at 8 kHz, 11.025 kHz is used for AM-radio quality audio, and 44.1 kHz is the standard for CD quality digital music. The problem of converting between different sampling rates has been discussed in numerous papers and books, and different methods have been proposed in [1]-[5].

The proposed sampling rate conversion library is targeted for processing blocks of audio data in real-time applications. It is implemented with a time-varying fractional delay filter that is controlled using a lookup table to convert between arbitrary rates. The library optimization has been performed with the energy profiling tools presented in [6] for Intel's StrongARM SA-110 microprocessor.

The motivation for this project is discussed in Section 2; the theory description is given in Section 3; the design and implementation of the library is described in Section 4; test results are in Section 5; optimization is discussed in Section 6. The library API is given in the Appendix.

2. Motivation

The motivation for implementing a sampling rate conversion library came from working with the audio on the SmartBadge IV [7]. The SmartBadge IV is the next-generation context-aware device from Hewlett-Packard Laboratories. It is controlled by Intel's StrongARM SA-1110 processor and a StrongARM SA-1111 companion chip. In addition to the processors, the Badge has on chip memory (FLASH and RAM), Philips UDA-1341 audio codec and various on-board sensors.

[*]Chung-Tse Mar and Ronald W. Schafer are with the Center for Signal & Image Processing, Georgia Institute of Technology, Atlanta, Georgia. Mat C. Hans, Mark T. Smith, and Tajana Simunic are with the Appliance Platforms Department, Hewlett-Packard Laboratories, Palo Alto, CA.

One challenge with digital audio on this research platform prototype is the rate at which the UDA-1341 codec samples the audio at both playback and recording. The codec on the SmartBadge IV does not have a dedicated clock; instead it uses the clock signal from the StrongARM. This will produce slight deviations from the standard sampling rates illustrated in Table 1.

Table 1 Differences in Sampling Rates on the SmartBadge IV

Ideal Sampling Frequency	8.0 kHz	11.025 kHz	16.00 kHz	22.05 kHz	32.0 kHz	44.1 kHz
Actual Sampling Frequency	8.0 kHz	11.01 kHz	16.05 kHz	22.46 kHz	31.2 kHz	43.2 kHz

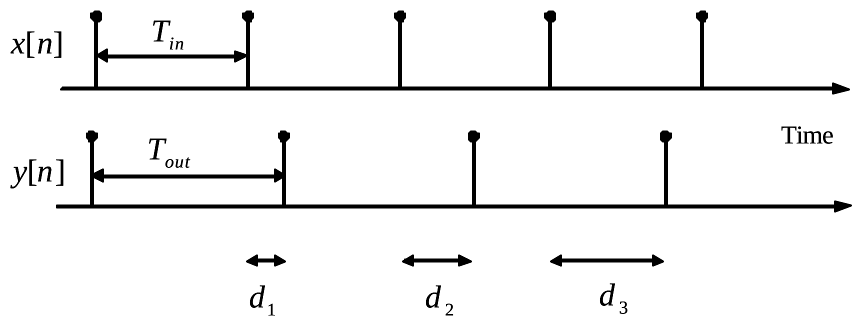
Real-time sampling rate conversion must be done on both the audio input and output to record and playout high quality audio, avoiding distortion and possible over or under-flow in real-time communication application. Different methods for sampling rate conversion are discussed in Section 3.

Energy consumption and power dissipation are critical to portable systems like the SmartBadge IV. System's energy consumption dictates its battery life. While microprocessors have become smaller and more powerful, the same advances have not been made by batteries. As a result, engineers need to explicitly design both hardware and software of portable systems for low energy consumption. The sampling rate conversion library proposed in this paper is optimized for execution on the StrongARM processor used by the SmartBadge IV. The optimization is summarized in Section 6.

3. Theory of Operation

Sampling rate conversion is a problem of interpolating a signal at non-integer multiples of the original sampling period. This may also be viewed as delaying the signal by a time-varying fractional amount as demonstrated in Figure 1, for a rate conversion of a signal with sampling period, T_{in}, to a new period, T_{out}.

Figure 1 Delaying the signal by a time-varying fractional amount



To calculate each samples of the output, $y[n]$, the input, $x[n]$, must be delayed by fractional amounts, d_i, of the original sampling period. Delaying a signal by d_i corresponds to a multiplication by the complex exponential, e^{-jwd_i}, in the frequency domain. This may be achieved by a system with the frequency response:

$$H_i(e^{jw}) = e^{-jwd_i}\text{, for all } w \tag{1}$$

which corresponds to an all-pass filter. For use in sampling rate conversion, a bandlimiting factor must be added to prevent aliasing if the input rate, F_{in}, is higher than the output rate, F_{out}. Specifically, the conversion requires a set of filters, known as fractional delay filters:

$$H_i(e^{jw}) = e^{-jwd_i}\text{, for } 0 \le w \le \min(p, \frac{pF_{out}}{F_{in}}) \tag{2}$$

The corresponding impulse response of these filters, $h_i[n]$, may be calculated to be:

$$h_i[n] = \frac{\sin[gp(n-d_i)]}{p(n-d_i)}\text{, for all } n, \tag{3}$$

where g is a bandlimiting factor computed as:

$$g = \min(1, \frac{F_{out}}{F_{in}}) \tag{4}$$

The impulse response is a delayed *sinc* function of infinite length. Different methods exist for approximating this unrealizable system with a causal filter of finite length. A detailed analysis may be found in [8].

Several different rate conversion methods were considered for use in this library. They differ in the method used to design the fractional delay filters. The method proposed in [3] uses B-spline functions. [5] uses the Farrow polynomial approximation proposed in [9]. In both cases, the filter coefficients are low-order polynomial functions of the delay:

$$h_i[n] = \sum_{k=0}^{N-1} c_n[k] d_i^k \tag{5}$$

where N is the order of the polynomial used to approximate each filter coefficient. The coefficients may be easily updated for an arbitrary delay at run-time as the input is being filtered. This is efficient only if the filter length is kept very short. For processing of high quality audio, the input signal must first be upsampled by some factor in order for the short variable-delay filter to generate accurate outputs. This method is feasible for a fixed conversion factor, F_{out}/F_{in}, but it may get complex if the conversion factor is to be made flexible because the coefficients, $c_n[k]$, in (5) must be recomputed to approximate a filter response with a different cutoff frequency.

The method proposed in [4] does not required upsampling of the input. It uses the window method for filter design [10], where the filter coefficients are computed as:

$$h_i[n] = w(n-d_i)\,\mathrm{sinc}[g(n-d_i)] \text{ for } n = 1 \ldots N, \tag{6}$$

where $w(n)$ is the window function, g is the bandlimiting factor computed in (4), and N is the length of the filter. Commonly used window functions include the Kaiser window and the Dolph-Chebyshev window. Since calculating *sinc* and window functions for arbitrary delays at run-time is too computationally expensive, a lookup table may be used. Interestingly, for a given factor of conversion, F_{out}/F_{in}, the total number of different delays, d_i, is finite. The delays also vary periodically during the conversion. Suppose that this factor may be reduced to the an integral fraction, R_{out}/R_{in}. The delays, d_i, may be computed as:

$$d_i = i\frac{R_{in}}{R_{out}} - \left\lfloor i\frac{R_{in}}{R_{out}} \right\rfloor \tag{7}$$

where $i = 0,1,\ldots(R_{out}-1)$, and $\lfloor x \rfloor$ denotes the largest integer less than or equal to x. Observe that $0 \le d_i < 1$. A table of filters corresponding to the set of R_{out} delays for a particular factor of conversion may be pre-computed. The number of elements in the filter table is equal to $R_{out} \times N$, where N is the filter length. For example, when converting 44.1 kHz to 48 kHz, we may use $R_{in} = 147$ and $R_{out} = 160$, there are 160 output samples for 147 input samples, and a total of 160 different delays to be calculated. If a filter length of 65 is used with 16-bit resolution for each coefficient, the lookup table would require 20 Kbytes of memory. An important point to note is that the filter must be operated at the output sampling rate. This can be achieved by shifting the input samples by a time-varying number of samples, s_i, into the filter's delay line:

$$s_i = \left\lfloor i\frac{R_{in}}{R_{out}} \right\rfloor - \left\lfloor (i-1)\frac{R_{in}}{R_{out}} \right\rfloor \tag{8}$$

The output is calculated using:

$$y[n] = \sum_{k=0}^{N-1} h_i(k)x(n+s_i-k) \tag{9}$$

where the variable i indexes the particular filter coefficients and shift values to use. It is periodic and is equal to $n \bmod R_{out}$.

The lookup table method is highly efficient since the coefficients are only computed once. On the other hand, different tables are needed for different rates, and for some factors of conversion, in which R_{out} is a large number, the table may take up more than 100 Kbytes of memory.

A more flexible method is proposed in [2]. A low-pass filter with an arbitrary delay may be computed by linearly interpolating an over-sampled and windowed *sinc* function. To illustrate how this may be done, suppose the windowed *sinc* function with zero delay is represented as $h[n]$, this is oversampled by a factor, *L*, to produce:

$$h^{oversampled}[n] = h[\frac{n}{L}] \tag{10}$$

This oversampled function can be computed beforehand and stored into a header file to be compiled with the library code. To calculate $h_i[n]$ for the delay d_i:

$$h_i[n] = h[n - d_i] = h^{oversampled}[nL - d_i L] \tag{11}$$

As in (7), $0 \le d_i < 1$, and $d_i L$ usually will not be an integer. But if L is large enough, linear interpolating between the oversampled coefficients will produce accurate estimates for $h[n - d_i]$. Details on how this is done with fixed point numbers and an analysis on the size requirement of *L* may be found in [2]. This is the method employed in this library. The filter coefficients may either be computed at run-time with minimum overhead (one multiplication and one addition for each coefficient), or they may be computed and stored in a lookup table. This library offers the user both options, but the lookup table should be used if possible because of the significant reduction in computational complexity (see Section 5).

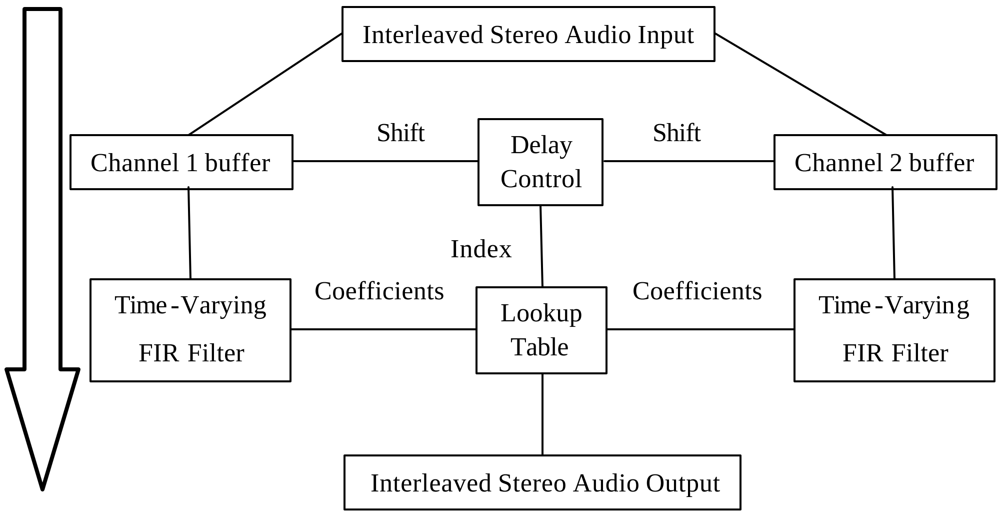
4. Library Design and Implementation

The sampling rate conversion is implemented as a time-varying FIR filter in our library. The filter coefficients may be either pre-computed and stored in a lookup table, or computed at run-time by linearly interpolating an oversampled filter response.

The design of the library is similar to the block FIR filtering functions in Intel's Integrated Performance Primitives Library [11]. The library can be used both to process large chunks of data as a whole and small blocks of data with repeated library calls; such is the case when streaming audio over the Internet in real-time. To avoid transient filter response between contiguous blocks of data, the internal state of the system should be maintained. This includes the index of the particular filter coefficients to use and some input samples at the end of a block. If the system is initialized at the start of each new block to the state that was reached at the end of the previous block, then a steady-state filter response is maintained.

A diagram of the sampling rate conversion system as implemented in the library is shown in Figure 2.

Figure 2 Block diagram of sampling rate conversion system.



A block of interleaved stereo data is first separated into two buffers internal to the library. The delay control element is implemented as a table containing the possible shift values as calculated in (8). It controls how many input samples should be shifted into the filter and which set of coefficients is to be used. The output is calculated following (9). The resulting effect is a time-varying filter operating at the output sampling frequency.

To avoid transient response at the end of a block, some input samples are saved in the internal buffers until the next library call. A flush routine is provided in the library for transient response calculation.

5. Tests Results

The library was tested using both natural audio signals and computer generated sine waves. The conversion from 44.1 kHz to 48 kHz and 22.05 kHz will be illustrated here as examples. The test input signal used is a sum of two sinusoids with the frequencies 4 kHz and 16 kHz; it was generated using 16-bits of resolution. Its spectrum is plotted in Figure 3.

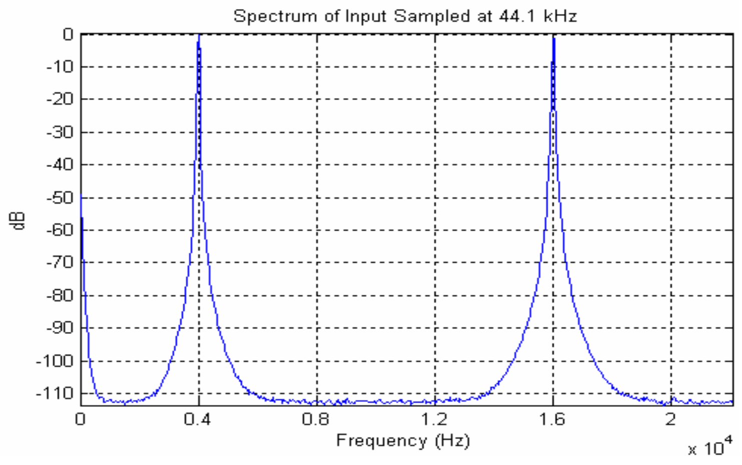


Figure 3. Spectrum of input signal sampled at 44.1 kHz.

The input was resampled to 48 kHz using the sampling rate conversion library, and the output spectrum is shown in Figure 4.

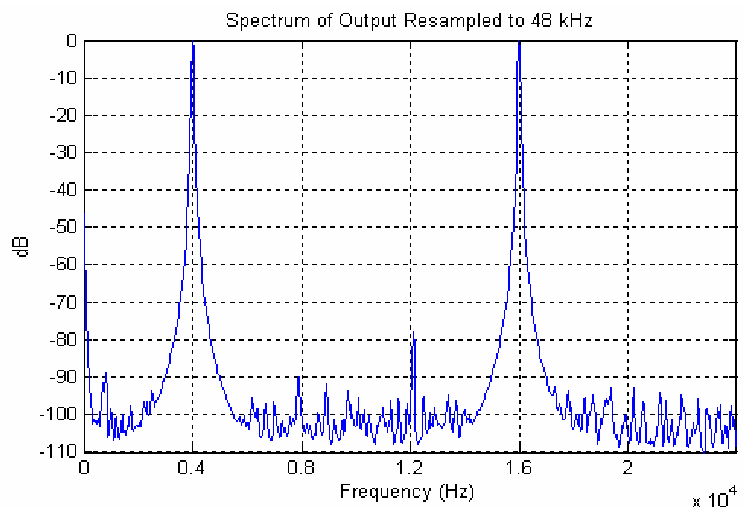


Figure 4. Spectrum of output signal resampled to 48 kHz.

Most distortions generated by the conversion are 80 dB below the signal level. The conversion was repeated for an output rate of 22.05 kHz, and the result is shown in Figure 5.

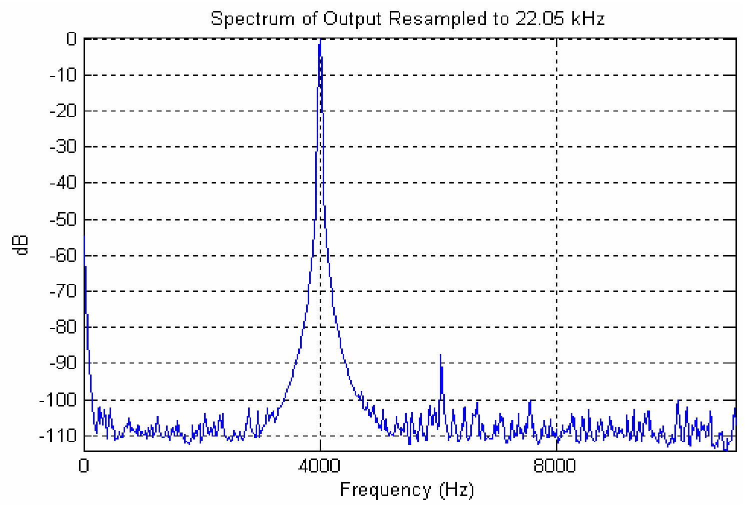


Figure 5. Spectrum of output signal resampled to 22.05 kHz.

As expected, the frequency component above 11 kHz has been eliminated by the bandlimiting factor in the interpolation filters. The distortions are still mostly 80 dB below the signal level.

6. Optimization

The fixed-point sampling rate conversion implementation code [12] for the method proposed in [2] was used as a starting point for the library implementation. It was modified to fit the library specifications and optimized for execution on the Hewlett-Packard Laboratories SmartBadge portable device. Optimization was done manually in the ARM Software Development Toolkit (SDT) [13] with the guidance of the cycle-accurate energy profiler proposed in [6] and the application notes published by ARM [14].

At the implementation level, many "tricks" from [14] were used to optimize the code. They included, among others, conditional execution, function inlining, and efficient register allocation. Two other major modifications are discussed below.

Stereo audio is usually stored as interleaved 16-bit integers, but for sampling rate conversion, each channel must be processed separately. In the original code, the separation was done by the user application, and two library calls must be made, one for each channel. This introduces redundancies. To process stereo data more efficiently, the channel separation was moved inside of the library, such that a single library call is used for conversion.

Additionally, a lookup table for the filter coefficients was added to the original code to eliminate the calculation of the coefficient at run-time. On the other hand, it requires an initialization function to compile the filter table.

The result of the optimization is summarized in Table 2.

Table 2 Performance and Energy for Sampling Rate Conversion Implementations, 44.1 kHz to 48 kHz

	Performance	**Energy (mWhr)**		
Code Revision	Total Cycles (%Reduction)	Processor	Memory	Total Battery (%Reduction)
Original [12]	111760942 (0%)	1.10E-02	6.79E-03	3.23E-02 (0%)
Optimized	67752335 (39%)	1.02E-02	5.97E-03	2.91E-02 (10%)
Optimized with Lookup Table	39863181 (64%)	6.89E-03	4.04E-03	2.02E-02 (37%)

The above data were collected from the energy profiler, which was configured for the StrongARM-110 processor using 1 MB of Flash memory and 2 MB of SRAM. The StrongARM SA-1110 processor on the SmartBadge IV is not yet supported by the ARM SDT, so the SA-110 is used instead for the simulation. To generate the data, the library was used to convert five seconds of stereo audio recorded at 44.1 kHz to 48 kHz in blocks of 1024 samples using a 13-tap filter. Similar reductions may be found in the down-sampling conversion from 44.1 kHz to 8 kHz shown in Table 3.

Table 3 Performance and Energy for Sampling Rate Conversion Implementations, 44.1 kHz to 8 kHz

	Performance	**Energy (mWhr)**		
Code Revision	Total Cycles (%Reduction)	Processor	Memory	Total Battery (%Reduction)
Original [12]	87634853 (0%)	8.00E-03	4.83E-03	2.32E-02 (0%)
Optimized	54581156 (38%)	6.31E-03	3.64E-03	1.78E-02 (23%)
Optimized with Lookup Table	33830392 (61%)	5.94E-03	3.50E-03	1.74E-02 (25%)

These results are overall lower than those in Table 2 because the down-sampling output data length is much shorter. The reductions may be even more dramatic if the filter length was increased to 65 as in Table 4.

Table 4 Performance and Energy for Sampling Rate Conversion Implementations, 44.1 kHz to 8 kHz

	Performance	**Energy (mWhr)**		
Code Revision	Total Cycles (%Reduction)	Processor	Memory	Total Battery (%Reduction)
Original [12]	2139419586 (0%)	7.39E-01	4.63E-01	2.21E+00 (0%)
Optimized	1206338682 (44%)	3.82E-01	2.46E-01	1.15E+00 (48%)
Optimized with Lookup Table	212445210 (90%)	5.06E-02	2.96E-02	1.48E-01 (93%)

The increased savings are mainly due to the large overhead in calculating 65 coefficients at run-time that is avoided by using a lookup table.

References

[1] R. E. Crochiere and L. R. Rabiner, *Multirate Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1983.

[2] J. O. Smith and P. Gosset, "A flexible sampling-rate conversion method," in *Proc. IEEE Int. Conf. Acoustic Speech Signal Processing*, vol.2, pp. 19.4.1-19.4.4, 1984.

[3] S. Cucchi, F. Desinan, G. Parladori, and G. Sicuranza, "DSP Implementation of Arbitrary Sampling Frequency Conversion for High Quality Sound Application," *Int. Conf. Acoustic Speech Signal Processing*, vol. 5, pp. 3609-3612, 1991.

[4] S. Park, G. Hillman, and R. Robles, "A Novel Structure for Real-Time Digital Sample-Rate Converters with Finite Precision Error Analysis," *Int. Conf. Acoustic Speech Signal Processing*, vol. 5, pp. 3613-3616, 1991.

[5] K. Rajamani, Y. Lai, and C. W. Farrow, "An Efficient Algorithm for Sample Rate Conversion from CD to DAT," *IEEE Signal Processing Letters*, vol. 7, no. 10, October, pp. 288-290, 2000.

[6] T. Simunic, L. Benini, and G. De Micheli, "Energy-Efficient Design of Battery-Powered Embedded Systems," *IEEE Transactions on VLSI Systems*, vol. 9, February, pp. 15-28, 2001. More information: http://akebono.stanford.edu/users/tajana/Papers.html

[7] M. T. Smith and G. Q. Maguire Jr., SmartBadge/BadgePad version 4, HP Labs and Royal Institute of Technology (KTH), http://www.it.kth.se/~maguire/badge4.html, date of access 2002-03-15.

[8] T. I. Laakso, V. Valimaki, M. Karjalainen, and U. K. Laine, "Splitting the Unit Delay," *IEEE Signal Processing Magazine*, vol. 13, January, pp. 30-60, 1996.

[9] C. W. Farrow, "A continuously variable digital delay element," in *Proc. IEEE Int. Symp. Circuits Syst.*, vol. 3, pp. 2641-2645, 1988.

[10] A. V. Oppenheim, R. W. Schafer, and J. R. Buck, "Filter Design Techniques," Chap. 7 in *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall, 1998.

[11] Intel Corporation, *Intel Integrated Performance Primitives for the Intel StrongARM SA-1110 Microprocessor Reference Manual*, version 1.01, 2001.

[12] Julius Smith, *Sampling Rate Conversion Program Version 1.6*, 2000. Available: http://www-ccrma.stanford.edu/~jos/resample/Available_Software.html

[13] Advanced RISC Machines Ltd (ARM), ARM Software Development Toolkit Version 2.11, 1997.

[14] Advanced RISC Machine Ltd (ARM), "Application Note 34: Writing Efficient C for ARM," [Online document], Available: http://www.arm.com/arm/Application_Notes?OpenDocument

[15] J. F. Kaiser, "Nonrecursive Digital Filter Design Using the Io-Sinh Window Function," in *Proc. IEEE Int. Symp. Circuits Syst.*, pp. 20-23, 1974.

Appendix

1. Library API

This library supports sample rate conversion of contiguous data blocks using repeated calls without loss of internal filter state. Transient filter responses are avoided at block boundaries by providing a state initialization mechanism for the function. If the filter is initialized at the start of each new block to the state that was reached at the end of the previous block, then a steady-state filter response is maintained when filtering a long data record on a block-by-block basis. Application code making use of the resample function should adhere to the following usage model:

a. Initialization
Prior to calling resample for the first time, invoke resampleINIT() with the appropriate parameters to initialize the internal variables used by the library.

b. Resampling
If samples from a long sequence are processed in blocks using repeated calls to the library, the application should not modify the RESAMPLE_STATE variable in between successive calls. This implementation will produce around (TAP*factor) samples of start-up transient output when it is called for the first time.

c. Exiting
When there are no more inputs, the application should first call resampleBLOCK() with inLength set to zero. This should resample what's left in the internal buffer. If transient response is desired at the end, resampleFLUSH() should then be called. Once conversion has been completed, call resampleEXIT() to free up the memory used by the internal variables (lookup tables and internal buffer).

Block size management: Block processing in sample rate conversion presents a unique problem because the input block size is different from the output. In this library, the user should request (int)(inLength*factor) samples of output per inLength input samples. Requesting too many or too few outputs may result in resampleBLOCK returning an abnormal exit condition.

Processing interleaved stereo audio: Two channel (stereo) audio data is often stored with the channels interleaved. This library provides functions for resampling interleaved data so the application won't have to separate the channels.

2. Example code

This is a simple example code that uses the library to convert a two-channel, raw PCM audio file recorded at 44.1 kHz to 32 kHz. Conversion is done in blocks of 100 ms.

```
#include <stdio.h>
#include <stdlib.h>
#include "resample_Smith_Gossett.h"

void example(void)
{
        RESAMPLE_STATE state;
        int quality, stereo, use_table;
        int Fin, Fout;
        int Nin;     /* number of input samples per block */
        int Nout;    /* number of output samples per block */
        int inBytes;            /* bytes per input block */
        int outBytes;           /* bytes per output block*/

        int *inBuffer, *outBuffer;
        double factor;

        FILE *inFp, *outFp;

        Fin = 44100;
        Fout = 32000;
        factor = (double)Fout/(double)Fin;

        /* the audio is processed in blocks of 0.1 second */
        Nin = 4410;
        Nout = (int)( (double)Nin*factor );

        /* Allocate memory for in/out blocks */
        inBytes = Nin*sizeof(int);
        outBytes = Nout*sizeof(int);
        inBuffer = (int *)malloc(inBytes);
        outBuffer = (int *)malloc(outBytes);

        /* Open up the files we need */
        inFp = fopen("in.pcm", "rb");
        outFp = fopen("out.pcm", "wb");

        quality = 0;            /* normal quality */
        stereo = 1;             /* stereo audio */
        use_table = 1;          /* use lookup table */
        resampleINIT(&state, factor, quality, stereo, use_table);  /* initialize internal variables */

        /* resample Nin input samples at a time until the end of the input file */
        while (fread(inBuffer, sizeof(int), Nin, inFp)==Nin) {
                resampleBLOCK_Stereo(inBuffer, outBuffer, Nin, &Nout, &state);
                fwrite(outBuffer, sizeof(int), Nout, outFp);
        }

        /* flush out what's left in the internal buffer */
        resampleBLOCK_Stereo(inBuffer, outBuffer, 0, &Nout, &state);
        fwrite(outBuffer, sizeof(int), Nout, outFp);
        Nout = resampleFLUSH_Stereo(outBuffer, &state);
        fwrite(outBuffer, sizeof(int), Nout, outFp);

        resampleEXIT(&state);           /* free up memory used by internal variables */

        fclose(inFp);
        fclose(outFp);
        free(outBuffer);
        free(inBuffer);
}
```