Audio Engineering Society
Convention Paper

Presented at the 119th Convention
2005 October 7–10 New York, New York USA

AN EFFICIENT ASYNCHRONOUS SAMPLING-RATE CONVERSION ALGORITHM FOR MULTI-CHANNEL AUDIO APPLICATIONS

Paul Beckmann[1], Timothy Stilson[2]

Audio Rendering Technology Center (ARTC), Analog Devices, Inc.
1741 Technology Drive, Suite 400, San Jose, CA. 95110

[1]Paul.Beckmann@analog.com, [2]Tim.Stilson@analog.com

ABSTRACT

We describe an asynchronous sampling-rate conversion (SRC) algorithm that is specifically tailored to multi-channel audio applications. The algorithm is capable of converting between arbitrary asynchronous sampling-rates around a fixed operating point, and is designed to operate in multi-threaded systems. The algorithm uses a set of fractional delay filters together with cubic interpolation to achieve accurate and efficient sampling-rate conversion.

1. INTRODUCTION

Sampling-rate conversion is the process of converting a discrete-time signal $x[n]$ sampled at a rate FS_{in} to another signal $y[m]$ sampled at a rate FS_{out}. If the ratio FS_{out}/FS_{in} is a constant then *synchronous* sampling-rate conversion may be used. If the ratio FS_{out}/FS_{in} is only approximately known at design time, irrational, or slowly varying over time, then *asynchronous* sampling-rate conversion is required. Asynchronous conversion is a generalization of synchronous conversion, and asynchronous conversion may be used in synchronous applications, though there may be a computational or memory penalty.

Synchronous sampling-rate conversion occurs when there is a single master clock in the system. For example, a portable audio player that accepts content at a variety of sampling-rates FS_{in} ={32 kHz, 44.1 kHz, 48 kHz} and must convert to a fixed output rate of FS_{out} = 48 kHz. Asynchronous conversion occurs when there are two or more separate clocks in the system. For example, multiple digital sources – each with its own sampling-rate – that must be jointly processed.

Audio applications requiring sampling-rate conversion were once restricted to high-end mixing consoles and post-production systems. However, with the advent of streaming audio, networked audio players, and compressed audio, sampling-rate conversion is being incorporated into a variety of consumer products.

Furthermore, converting all audio content to a single sampling-rate for subsequent equalization and effects processing may simplify the design of products such as home theater systems that must handle content in a variety of sampling-rates.

Sampling-rate conversion can be done in either hardware or software. Hardware implementations may be standalone IC's such as the Analog Devices AD1896 or peripherals integrated into audio specific digital signal processors, such as the Analog Devices ADSP-21364. Alternatively, software implementations are attractive when the audio system already contains a digital signal processor with spare resources or if the sampling-rate conversion occurs between two software modules such as an audio decoder and effects processing.

This paper details a software implementation of a sampling-rate conversion algorithm which is particularly efficient at converting multiple audio channels. The algorithm combines fractional sample delay filters with cubic polynomial interpolation. Although conceptually similar to existing methods, the manner in which the calculations are partitioned is novel and provides the computational efficiency.

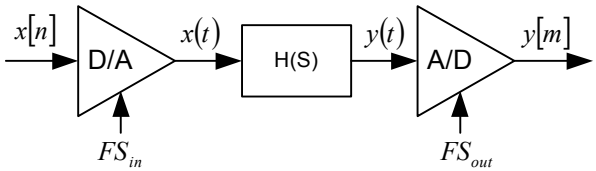
The design of the algorithm is based upon the following assumptions:

1. The clocks are asynchronous, but the instantaneous ratio stays within a small range during a run. The system will be designed for an "ideal" ratio within that range, but will be allowed to stray small distances from the ideal ratio.
2. Linear phase.
3. Will operate on multi-channel signals, where all the channels share the same sample times.

The paper is organized as follows. Section 2 presents background information and compares our approach to existing sampling-rate conversion algorithms. Section 3 describes the core algorithm which combines fractional sample delay filters and polynomial interpolation. Section 4 discusses generating a set of fractional delay filters using a polyphase decomposition. These filters provide the necessary delay, and when applicable, lowpass filtering. The theoretical performance limits of the algorithm are covered in Section 5. Section 6 then discusses an actual implementation on a commercial audio processor and presents MIPS and memory usage for typical applications. Finally, Section 7 summarizes the key innovations and benefits of the algorithm.

2. BACKGROUND

The subject of sampling-rate conversion, or alternatively, bandlimited interpolation, has been studied extensively, with just a portion of the work being presented in [6], [3], [4], [7], and [8]. The problem is usually attacked with what is called in [6] the "analog interpretation" of sampling-rate conversion:



The signal is converted to continuous-time at a rate FS_{in}, filtered by some filter $H(s)$, which combines the reconstruction filter of the D/A converter with the anti-aliasing filter of the A/D converter, and then sampled at a sampling-rate FS_{out} to give the output $y[m]$. The design challenge is to implement a discrete-time simulation of this process both accurately and efficiently.

It turns out [4] [7] that this design ends up, theoretically, as the implementation of a time-varying discrete-time filter, whose time-varying kernel is $h(n - mT_{\text{in}})$, and which is evaluated at a sampling period of T_{out} (where $T_{\text{in}} = 1/FS_{\text{in}}$, and $T_{\text{out}} = 1/FS_{\text{out}}$). In other words, the ideal time varying filter has a kernel consisting of samples of the ideal continuous-time filter, spaced according to the sampling-rates, and offset according to the instantaneous "phase" of the converter. The major differences between most techniques tend to be in the ways that they implement the time-varying filter and/or how they update the filter coefficients in order to be as efficient as possible.

2.1. Synchronous vs. Asynchronous

Some techniques restrict the input and output sampling-rate to a strict ratio, such that although the input and output are different sampling-rates, their ratio stays absolutely constant. These techniques are called "synchronous". They tend to make use of this restriction by noting that the time-varying filter will thus

become predictable, and in the case of rational ratios, it will assume only a finite set of configurations. Often these are precomputed and stored in order to remove the need to compute the time-varying filter settings at runtime (trading off storage space for reduced computation).

Many uses of sample-rate converters, however, do not satisfy the synchronous assumption, so another class of "asynchronous" sample-rate converter methods exist for these situations. In these cases, the sampling-rate ratio is assumed to be unknown at design time, probably irrational, and probably time-varying. Therefore, the configurations of the time-varying filter cannot be completely predicted or precomputed, and some amount of online computation of the filter settings must be done. Asynchronous methods implement various simplifications and/or approximations to make this online computation as cheap as possible.

2.2. FIR vs. IIR

The ideal filter $H(s)$ is not inherently either FIR or IIR, being a continuous-time idealization, but the discrete-time design must implement one or the other. IIR filters have the advantage of being able to implement designs with sharp features with much less computation than the equivalent FIR filter, but with the drawback of not being able to implement linear-phase designs (a system whereby all frequencies are delayed the same amount when passing through the system), being at best only able to approximate them. High-end audio systems tend to require linear-phase designs, so FIR-based techniques tend to dominate that field, at the cost of more expensive methods, both in computational cost and in storage cost.

2.3. Comparison with Other Techniques

Crochiere and Rabiner [6] present a good background for the sampling-rate conversion problem, and describe the "classical" rational-ratio design of implementing the ratio L/M as an upsampling by a factor of L followed by appropriate filtering, followed by a downsampling by a factor of M. This method tends to be useful for small values of L and M, otherwise the intermediate sampling-rate tends to get unwieldy. It is, however, often used as the starting point for other design derivations.

Ramstad [3] presents several methods, both FIR and IIR, including one that will be very close to the method derived in this paper. It first presents a straightforward implementation of the $h(n - mT_{\text{in}})$ filter, where a rational ratio is assumed, and thus the coefficients of the runtime filter are simply a periodic visiting of elements of a precomputed array containing the required samples of *h(t)*. While in essence a synchronous technique, it can be extended to asynchronous usage, at the penalty of needing a very large array to reduce artifacts due to not interpolating the coefficients. Further variations include implementing an integer-ratio upsampler followed by a low-order polynomial interpolator. The design presented in this paper will end up quite close to this design, but with some further optimizations.

Smith and Gosset [4] presents a design for *h(t)* based on the windowed sinc function, and an asynchronous conversion method which computes runtime FIR coefficients for the variable filter via linear interpolation from a precomputed table. The method is designed to be used for a very wide range of conversion ratios with a single table, which requires that the FIR coefficient generation handle a particularly general case. The design presented in this paper is closely related to the Smith-Gosset method (though it is derived more like the Ramstad method), but the derivation will make some simplifying and/or restricting assumptions that allow the resulting algorithm to be implemented more efficiently.

The Analog Devices AD1896 hardware sample-rate converter [11] implements a similar concept to the Smith-Gosset design, but with higher-order interpolation of the table lookups to keep the table size down. Efficiency comparison with the design presented in this paper is difficult, as this paper presents a method intended for software implementation, whereas the AD1896 implementation can make use of parallel computation if necessary.

Russell and Beckmann [7] present an innovative IIR-based technique which computes IIR filter coefficients recursively on the fly and can thus be very efficient. However, being an IIR technique, it cannot implement linear phase designs, and is not easily compared with the approach used in this paper.

A different approach to the basic problem is presented in [8] and enhanced in a series of papers, notably [9]. In this case, the *h(t)* filtering and the interpolation are transposed, such that instead of implementing an

interpolation between a number of oversampled samples (or sub-sample-delay filters), the coefficients of the interpolation itself are filters. In this situation, the design of the interpolators and the filters are combined into a single design (which may have useful implications from a design optimization standpoint), and the time-variation is limited to a single parameter, leaving the filters to be time-invariant. This technique is still being developed, however, and it appears that most of the development has been done with low-order filters, and a method for designing the filters in terms of *h(t)* is only recently shown [10], such that it is not yet clear how designs based on this method compare to designs following the above tracks, either from an efficiency standpoint or from a distortion performance standpoint.

Finally, almost all existing techniques either do not or cannot separate the variation in the time-varying filter's design from the running of the filter. As such, most methods end up having to replicate large portions of the algorithm (if not the whole algorithm) across channels when converting multi-channel signals (where all the channels are sampled synchronously). The method being presented here was designed with an emphasis on factoring out as much of the channel-independent processing as possible, in order to minimize the amount of processing that must be replicated on a per-channel basis.

3. THEORY OF OPERATION

Converting between sampling-rates is equivalent to interpolating between sample values in a discrete-time signal. Consider a bandlimited continuous-time signal $x(t)$ that has been sampled at a rate FS_{in} to generate a discrete-time signal

$$x[n] = x\left(\frac{n}{F_{in}}\right).$$

This is shown in Figure 1. Here we are using the notation used in [7], where discrete-time signals are denoted with square brackets ($x[n]$ and $y[m]$), and continuous-time with parentheses ($x(t)$ and $y(t)$). Assume that FS_{in} is high enough to avoid aliasing and loss of information. The circles represent the sample values $x[n]$ and the solid line represents $x(t)$. Converting to a different output sampling-rate is equivalent to calculating other values of the underlying continuous time function $x(t)$.

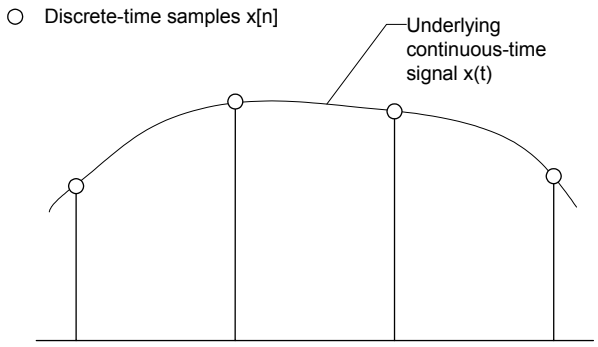


Figure 1 Sampling-rate conversion is equivalent to resampling the underlying continuous-time signal.

Let $y[m]$ represent the resampled output signal at rate FS_{out}. $y[m]$ is related to $x(t)$ via the equation:

$$y[m] = x\left(\frac{m}{F_{out}}\right)$$

The key quantity in sampling-rate conversion is the ratio $FS_{\text{out}}/FS_{\text{in}}$ and thus without loss of generality we can assume that $FS_{\text{in}} = 1$ Hz.

3.1. Core Algorithm

The sampling-rate converter described in this document is based on cubic polynomial interpolation but extends to arbitrary order polynomial interpolation. For ease of presentation (and notation), this paper assumes that cubic interpolation is used. However, the technique applies to other order polynomials in a straightforward fashion.

We begin with a precomputed set of M interpolation filters that calculate output values spaced every 1/M samples. These interpolation filters are FIR and are conceptually equivalent to fractional sample delay filters. For now, assume that such a set of filters exists; Section 4 describes how these filters are generated. With these M filters, the input signal values can be calculated exactly at all of the X's shown in Figure 2.

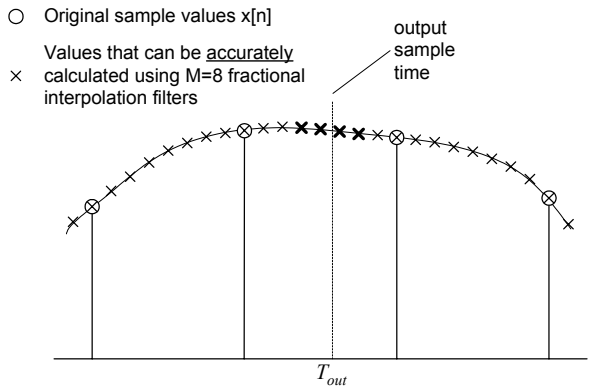


Figure 2. The input signal can be calculated between samples at exactly M=8 locations using a set of precomputed interpolation filters.

Suppose that we wish to calculate an output value at the time indicated by the vertical dashed line in Figure 2. Denote this time by T_{out}. Since T_{out} does not fall exactly on an X, we must estimate the value based on neighboring samples. We first calculate the output signal at the 4 nearest X values (shown in bold) using the precomputed interpolation filters. Two X values lies to the left and two X values lie to the right of T_{out}. Then, we fit a 3rd order polynomial $P(t)$ through these 4 values and evaluate $P(t)$ at T_{out} as shown below.

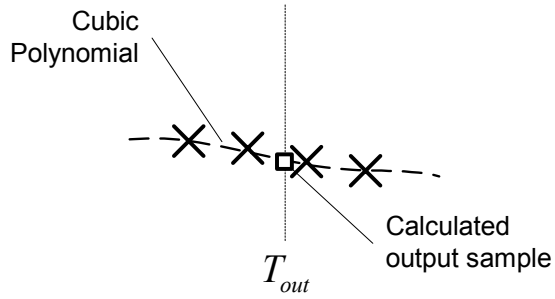


Figure 3. Close up of the interpolation operation around T_{out}.

The overall process for 1 sample value is then summarized as:

1. Calculate the 4 nearest X values: 2 to the left, and 2 to the right of the desired output time.

2. Fit a cubic polynomial $P(t)$ to these 4 values.

3. Evaluate this polynomial at the desired output time T_{out}.

Next, separate T_{out} into an integer and fractional portion[1]:

$$T_{out} = N + t_{out}$$

where

$$\begin{aligned} N &= floor(T_{out}) \\ t_{out} &= T_{out} - N \end{aligned} .$$

Based on this decomposition, $0 \le t_{out} < 1$. The set of interpolating filters to use depends solely on t_{out}; N only introduces a whole sample delay and specifies which set of input samples to use in the calculation. Let $h_k[n]$ denote the interpolation filter that calculates the output at time k/M, with $k = 0,1,...,M-1$. The 4 nearest sample values shown in Figure 3 would be calculated using the filters $h_k[n]$, $h_{k+1}[n]$, $h_{k+2}[n]$, and $h_{k+3}[n]$ where

$$k = floor(t_{out} * M) - 2 .$$

Examining this equation, we see that k falls in the range $-2 \le k \le M-3$. Furthermore, once k is calculated, we need a total of 4 filters: k, $k+1$, $k+2$, and $k+3$. The total set of filters required is thus: $k = -2,-1,0,1,...,M$. This set contains $M+3$ filters, not just the original M. We can simplify this set slightly by changing the previous equation to

$$k = floor(t_{out} * M) .$$

The only effect of this change is to introduce an additional latency of $2/M$ samples through the system. With this change, the set of filters required is $k = 0,1,...,M+2$. Note that the last 3 filters in this set are whole sample delays of the first 3 filters and are related by:

[1] This separation into integer and fractional portions is possible because we assumed that the input sampling-rate T_{in} equals 1 Hz.

$$\begin{aligned} h_M[n] &= h_0[n-1] \\ h_{M+1}[n] &= h_1[n-1] \\ h_{M+2}[n] &= h_2[n-1] \end{aligned}.$$

The interpolating filters calculate the output at times $\left\{0, \frac{1}{M}, \frac{2}{M}, ..., \frac{M+2}{M}\right\}$. Let L denote the length of these filters. These filters extend before and after the desired output time, and are thus non-causal. By introducing some latency into the processing we can make the overall system causal. We will apply the filter starting at time N and extending for L samples. The 4 "X" values are calculated as:

$$\begin{aligned} a &= \sum_{i=0}^{L-1} h_k[i]x[N+i] \\ b &= \sum_{i=0}^{L-1} h_{k+1}[i]x[N+i] \\ c &= \sum_{i=0}^{L-1} h_{k+2}[i]x[N+i] \\ d &= \sum_{i=0}^{L-1} h_{k+3}[i]x[N+i] \end{aligned}.$$

Now further decompose t_{out} into an "integer" and fractional portion, where the integer portion relates to the $1/M$ sample spacing.

$$t_{out} = T_M + tt$$

where

$$\begin{aligned} T_M &= \frac{1}{M} floor\left(t_{out} M\right) = \frac{k}{M} \\ tt &= t_{out} - T_M \end{aligned}.$$

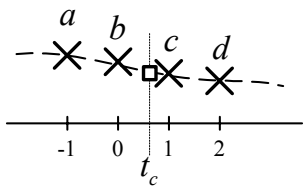
Note that tt is in the range $0 \le tt < \frac{1}{M}$. Now scale tt to obtain the quantity t_c which lies in the range $0 \le t_c < 1$,

$$t_c = M\ tt\ .$$

t_c represents the fractional position between the precomputed interpolation values (the X's in Figure 3).

3.2. Efficient Polynomial Interpolation

Using these definitions, we can redraw Figure 3 as:



Let $P(t)$ represent the polynomial used for interpolation. The output $y[m]$ equals $P(t)$ evaluated at $t = t_c$. Since the four values a, b, c, and d are used to define $P(t)$, $P(t)$ will be a 3rd order polynomial. Let $P(t)$ be written as:

$$P(t) = c_3 t^3 + c_2 t^2 + c_1 t + c_0\ .$$

Traditionally, polynomial interpolation is performed using a two step procedure. First the polynomial coefficients $\{c_0, c_1, c_2, c_3\}$ are calculated using the four values $\{a, b, c, d\}$. This requires solving 4 simultaneous linear equations. Next, this polynomial is evaluated at the desired output time t_c. These steps will be examined in more detail below.

The four simultaneous equations to solve are:

$$\begin{aligned} P(-1) &= a = -c_3 + c_2 - c_1 + c_0 \\ P(0) &= b = c_0 \\ P(+1) &= c = c_3 + c_2 + c_1 + c_0 \\ P(+2) &= d = 8c_3 + 4c_2 + 2c_1 + c_0 \end{aligned}.$$

Rewriting this using matrix notation yields

$$\begin{bmatrix} 1 & -1 & 1 & -1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

and the desired coefficients $\{c_0, c_1, c_2, c_3\}$ can be computed by inverting the 4x4 matrix:

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \frac{1}{6}\begin{bmatrix} 0 & 6 & 0 & 0 \\ -2 & -3 & 6 & -1 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix}\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}.$$

The desired output $y[m]$ can then be calculated with these coefficients as:

$$y[m] = P(t_c) = \begin{bmatrix} 1 & t_c & t_c^2 & t_c^3 \end{bmatrix}\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}.$$

Expanding this we find

$$\begin{aligned} y[m] = &\frac{1}{6}(-a+3b-3c+d)t_c^{\,3} \\ &+\frac{1}{6}(3a-6b+3c)t_c^{\,2} \\ &+\frac{1}{6}(-2a-3b+6c-d)t_c \\ &+b \end{aligned}.$$

The first key insight made during the development of this sampling-rate conversion algorithm was to note that the output $P(t_c)$ can be rearranged as a linear weighted sum of the values a, b, c, and d in which the weighting coefficients depend solely on t_c:

$$y[m] = w_a a + w_b b + w_c c + w_d d$$

where

$$\begin{aligned} w_a &= -\frac{1}{6}t_c^3 + \frac{1}{2}t_c^2 - \frac{1}{3}t_c \\ w_b &= \frac{1}{2}t_c^3 + -t_c^2 - \frac{1}{2}t_c + 1 \\ w_c &= -\frac{1}{2}t_c^3 + \frac{1}{2}t_c^2 + t_c \\ w_d &= \frac{1}{6}t_c^3 - \frac{1}{6}t_c \end{aligned}. \quad (1)$$

These weighting functions are plotted in Figure 4.

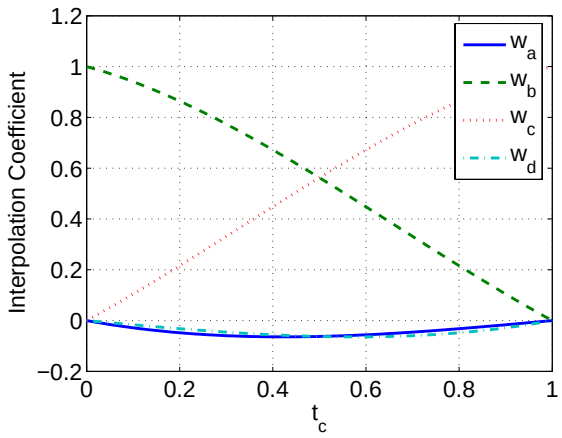


Figure 4. Weighting coefficients plotting over the interval from 0 to 1.

3.3. Interpolating Filter Coefficients

The second key insight made was that instead of computing a weighted some of filter outputs, we can apply the same weighting to the filter coefficients themselves and thereby derive a single time-varying filter. Substituting in the definitions for a, b, c, and d into $y[m]$ above yields:

$$\begin{aligned} y[m] &= w_a\left(\sum_{i=0}^{L-1} h_k[i]x[N+i]\right) \\ &+ w_b\left(\sum_{i=0}^{L-1} h_{k+1}[i]x[N+i]\right) \\ &+ w_c\left(\sum_{i=0}^{L-1} h_{k+2}[i]x[N+i]\right) \\ &+ w_d\left(\sum_{i=0}^{L-1} h_{k+3}[i]x[N+i]\right) \\ &= \sum_{i=0}^{L-1}\left(\begin{matrix} w_a h_k[i] + w_b h_{k+1}[i] \\ + w_c h_{k+2}[i] + w_d h_{k+3}[i] \end{matrix}\right)x[N+i] \\ &= \sum_{i=0}^{L-1} h_{net}[i]x[N+i] \end{aligned}$$

where

$$h_{net}[i] = w_a h_k[i] + w_b h_{k+1}[i] + w_c h_{k+2}[i] + w_d h_{k+3}[i].$$

This rearrangement of computation does not yield a benefit when a single channel is processed, but allows multiple synchronous channels to be computed efficiently as described in the next section.

3.4. Efficient Multi-Channel Operation

When multiple channels, all sampled at the same sample times, are involved, the filter $h_{net}[i]$ will be the same for each channel. Thus, $h_{net}[i]$ can be computed once and then applied to each channel in turn. The incremental computation required to process subsequent channels is equivalent to that required by a simple FIR filter.

We now summarize the overall algorithm. For each output time sample perform the following steps once:

1. Keep track of the fractional time offset t_c using the input and output sampling-rates.
2. Calculate the weighting coefficients w_a, w_b, w_c, and w_d shown in Equation (1).
3. Calculate $h_{net}[i]$ as a weighted sum of FIR filters.

Then per channel

4. Apply $h_{net}[i]$ separately to each input channel to calculate one sample of each output channel.

4. GENERATING COEFFICIENT SETS

The operation of the sampling-rate converter relies upon a set of fractional delay filters. This section describes one possible method of generating these filters based on a polyphase decomposition. The section first describes constraints on the design problem and then presents the polyphase decomposition.

4.1. Design Constraints

The design starts with specifications on the magnitude response of the fractional sample delay filters. These filters operate at the input sampling-rate and provide two functions in addition to the fractional sample delay:

- They filter out the phase discontinuity at $FS_{in}/2$
- Eliminate frequencies above $FS_{out}/2$ if $FS_{out} < FS_{in}$

Let R_{pass} and R_{stop} denote the desired passband ripple and stopband attenuation. The ideal design constraints when $FS_{out} < FS_{in}$ are shown in Figure 5. The ideal design constraints when $FS_{out} > FS_{in}$ are shown in Figure 6.

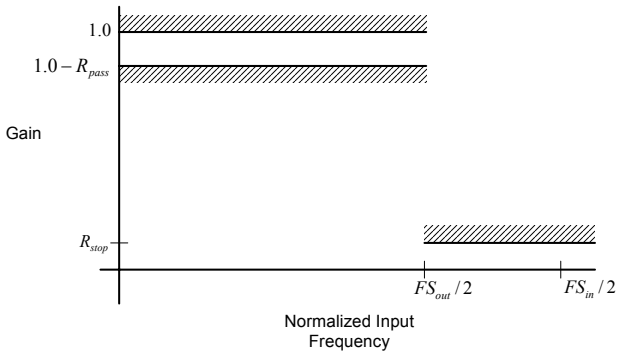


Figure 5 Ideal design constraints on the magnitude of the fractional sample filters when $FS_{out} < FS_{in}$. These constraints cannot be achieved because the transition region is zero width.

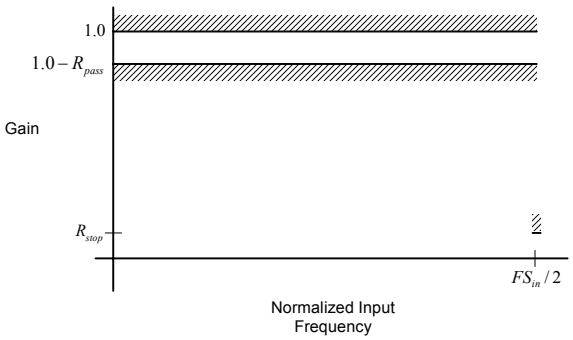


Figure 6 Ideal design constraints on the magnitude of the fractional sample filters when $FS_{out} > FS_{in}$. These constraints cannot be achieved because the transition region is zero width.

In practice, an additional constraint must be placed on the width of the transition region, defined as the frequency interval between the end of the filter's passband and the start of its stopband. The length of an FIR filter (and hence its implementation cost) is in general inversely proportional to the width of its transition region. The ideal design constraints in Figure 5 and Figure 6 have transition bands of zero width and realizable filters meeting these constraints therefore cannot be designed. Instead, the design constraints must be relaxed by the incorporation of a transition region of width W_{trans}. This is illustrated in Figure 7 and Figure 8.

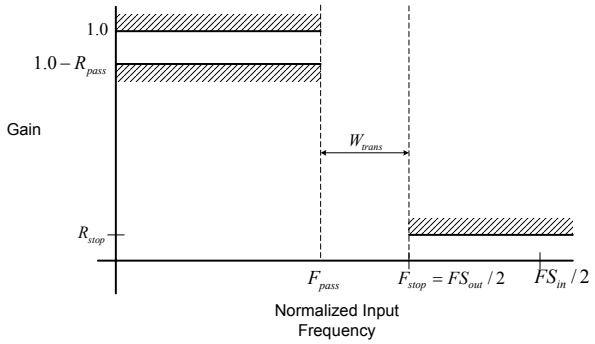


Figure 7. Realistic design constraints when $FS_{out} < FS_{in}$.

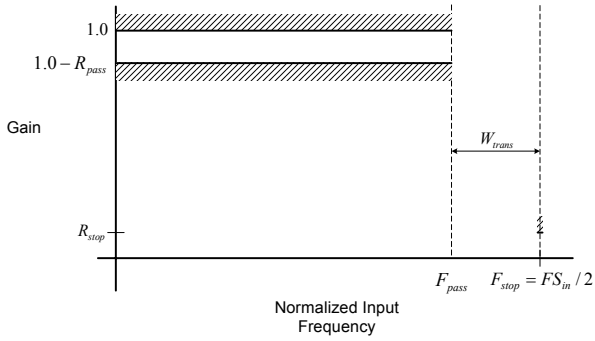


Figure 8 Realistic design constraints when $FS_{out} > FS_{in}$.

As a result of this transition region, the high frequency components of the input signal will be attenuated, and this leads to one of the fundamental design tradeoffs: longer filter lengths, and more computation/memory, which preserve the high frequency components of the input signal, versus shorter/cheaper filter lengths which attenuate high frequencies somewhat.

4.2. Polyphase Decomposition

The design constraints shown above are on the magnitude response of the fractional sample delay filters. Let F_{pass} and F_{stop} denote the normalized passband and stopband frequencies of these filters. All *M+3* filters can be designed jointly using a polyphase decomposition as follows. First, design a high order prototype filter $h_{proto}[n]$ with passband and stopband frequencies of F_{pass}/M and F_{stop}/M meeting the ripple constraints from before. Let L_{proto} denote the length of this prototype filter. Decompose the prototype filter in *M* polyphase components.

$$h_k[n] = h_{proto}[(M-k-1)+nM],$$

$$\text{for } k = 0,1,...,M-1, \text{ and } n = 0,1,...,L-1$$

where $L' = \text{ceil}(L_{proto}/M)$. These components will be referred to as "subfilters" because they are derived from a single longer filter. Note that because of the ceil() operation, some samples in the above equation may exceed the length of $h_{proto}[n]$; set these to zero. The final 3 subfilters are simply copies of the first 3 subfilters with an additional sample of delay added:

$$h_k[n] = h_{k-M}[n-1], \text{ for } k = M, M+1, M+2$$

$$h_k[0] = 0$$

These subfilters have a length of $L'+1$ samples. To simplify the implementation, zero pad the first *M* subfilters to length $L'+1$ so that all *M*+3 subfilters have the same final length. To be consistent with our previous notation, we will use $L = L'+1$.

A reasonable question to ask is "Why store 3 additional subfilters when a set of *M* would suffice?" This brings up a tradeoff between efficiency of computation and memory usage. This algorithm *can* be implemented by storing only *M* subfilters and computing the additional 3 on the fly. However, this complicates the control logic and consumes additional MIPS at run-time. In our DSP implementation we have chosen computational

efficiency over memory efficiency and store all *M+3* subfilters.

4.3. Design Example

This section provides an example of fractional delay filters designed using the polyphase decomposition. The example assumes that FS_{in} =48 kHz, FS_{out} =44.1 kHz, M =16, $W_{trans} = 0.175$ and that the prototype filters has 90 dB of stopband attenuation and 0.25 dB ripple.

The magnitude response of the prototype filter is shown in Figure 9. It clearly shows the lowpass characteristic and the specified stopband attenuation.

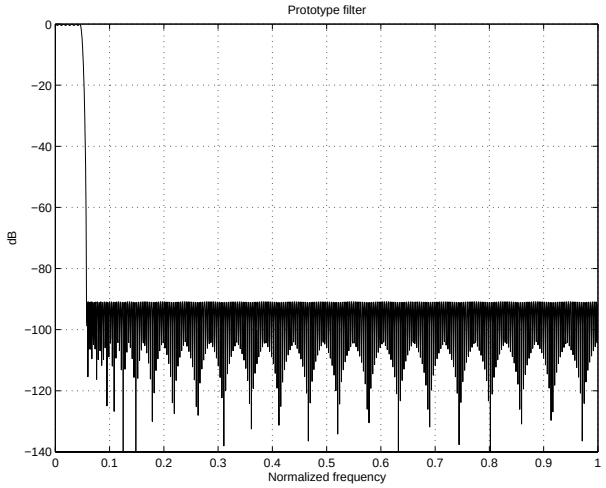


Figure 9 Magnitude response of the prototype filter.

The polyphase decomposition yields a total of 16 subfilters and 3 additional components are added by simple integer delay. Figure 10 shows the group delay of the 19 subfilters. The underlying latency is 17.5 samples and the k[th] subfilter has k/M additional samples of delay.

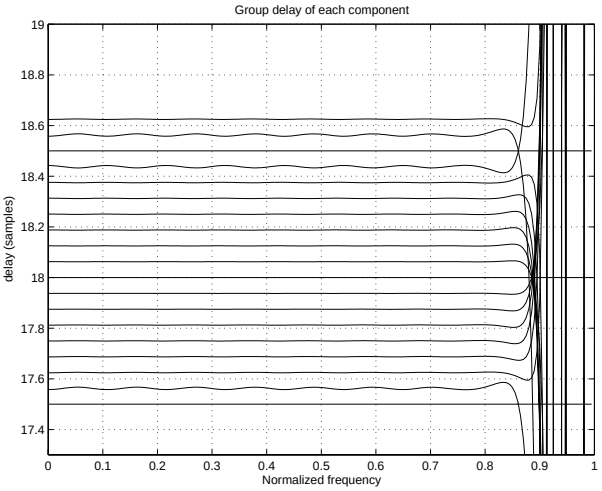


Figure 10. Group delay of the subfilters.

Lastly, Figure 11 shows the magnitude response of each subfilter. It shows the overall shape and then the passband and stopband details separately. Each response is lowpass, has roughly 90 dB of stopband attenuation and +/- 0.25 dB of ripple. The subfilters have also been scaled to have a peak gain of 0 dB.

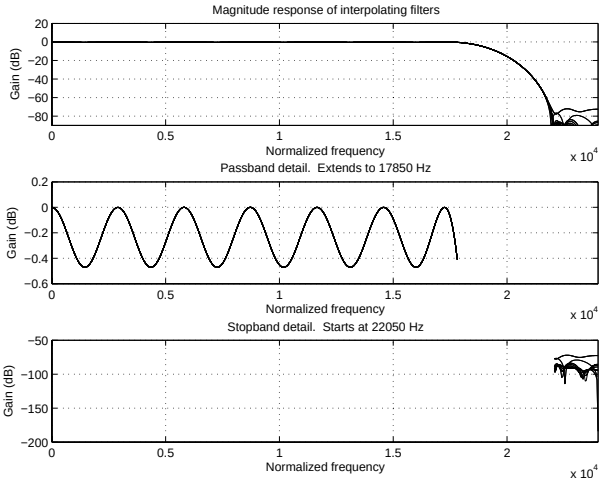


Figure 11 Magnitude response of each of the subfilters. From top to bottom, we see the overall response, passband detail, and stopband detail.

5. PERFORMANCE LIMITS

The performance of the sampling-rate converter algorithm is determined by 5 design parameters:

1. The length L of the subfilters (or equivalently, the length of the prototype lowpass filter).
2. The technique (Parks-McClellan, Kaiser, etc.) used to design the prototype lowpass filter.
3. Allowable passband and stopband ripple in the prototype lowpass filter.
4. The number of subfilters M.
5. The order of the polynomial interpolation.

The effects of the first three design parameters are well-known from classical FIR filter design techniques [1] [2] and will not be covered here. The last two items are particular to our algorithm and are examined in more detail below.

5.1. Quantifying Performance

The performance of a sampling-rate conversion algorithm can be quantified by MIPS and memory usage and also by its *audio performance.* The latter refers to the noise and distortion introduced by the processing. An analytical analysis is difficult to do given the complexity of the algorithm. Instead, the algorithm was implemented in MATLAB and then stimulated with a variety of test signals. Each test signal was a pure sine wave and if the sampling-rate converter was operating ideally, the output would also be a pure sine wave. We applied a notch filter at the output to eliminate the sinusoidal component and then measured: (1) the total amount of energy remaining (equal to the total harmonic distortion plus noise, THD) and (2) the worst case peak distortion. Since the THD always includes the peak distortion, THD will always be greater than or equal to the peak distortion.

5.2. Interpolation Method and Number of Subfilters

If we look at measurements of THD versus input sine frequency (Figures 12 and 13), we see that the performance is dominated by the stopband attenuation of the prototype filter design at low input frequencies. However, for high input frequencies, the THD starts rising above the prototype filter's stopband specs. This rise in THD turns out to be due to a set of sidebands around the test sine peak, which get higher in amplitude as the sine frequency increases. [The spacing of these sidebands was found to be equal to the "subfilter crossing rate", i.e. the rate at which the interpolation fraction crossed over subfilter boundaries, and therefore the rate at which the interpolation "crosses through" the range of the interpolation fraction. As one may expect, the interpolated FIR filter can be suboptimal according to the accuracy of the interpolation (and the interpolatibility of the prototype filter), and the shortcomings of non-ideal interpolation tend to be largest at high frequencies. Therefore, as the system passes through various amounts of interpolation, the actual implemented filters' frequency responses vary slightly, causing a small amplitude and/or phase modulation of the test signal. This modulation will be more pronounced for high signal frequencies, as that is where the variation in the frequency responses due to non-ideal interpolation will be greatest.

The total amount of this modulation can be affected in (at least) three ways: (1) the "smoothness" of the subfilter designs, (2) the type of interpolation, and (3) the number of subfilters. First, one should to design the subfilters so that adjacent subfilter coefficients are as close as possible, to reduce any unnecessary variation in the interpolation of the coefficients. We addressed this by designing the subfilters as a polyphase decomposition of a single lowpass FIR prototype filter. Given that the prototype is designed as a lowpass filter, its coefficients will tend to form a "smooth" shape, which will result in adjacent subfilters being well-suited to interpolation.

Second, a more accurate coefficient interpolation will have less interpolation error (assuming a smooth filter design), and hence less modulation. See Figure 12 and note that the slope of the modulation-based distortion increases by approximately 6 dB/oct between 1st-order (12 dB/oct), 2nd-order (18 dB/oct), and 3rd-order (24 dB/oct) polynomial interpolations, thus shrinking the range of frequencies which are noticeably affected by the modulation distortion up against the top of the frequency band.

Third, as the number of subfilters increases, the range of the sub-sample fraction parameter is "more densely

sampled", leading to adjacent subfilters which are more similar to each other (again, as long as the filter design is smooth), and hence reducing the amount of error that can occur in the interpolation between them. See Figure 13, and note that the modulation-based distortion dropped by 18 dB for each doubling of the number of subfilters. In this particular simulation, quadratic coefficient interpolation was used, which has a slope of 18dB/oct for its distortion, and so doubling the number of subfilters gave the same amount of distortion as when operating on a signal with half the frequency. This makes intuitive sense, since the input sine would be sampled at a similarly increased density in both cases.

As noted elsewhere, one of the benefits of this particular design is that by computing the interpolation weights only once per sample, one can often afford a more expensive interpolation than a method which might compute the weights every tap, and therefore allow a much smaller prototype filter and hence a smaller data set.

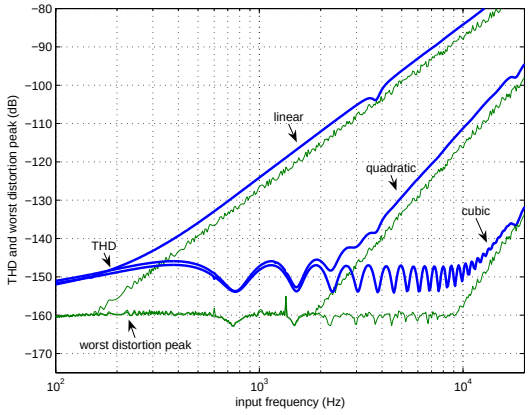


Figure 12. Interpolation error artifacts -- THD and worst-case aliasing peak amplitude vs. input sine-wave frequency for linear, quadratic, and cubic coefficient interpolation. The example is for 48 kHz to 44.1 kHz conversion, M=32, and a prototype filter with stopband attenuation of -160 dB.

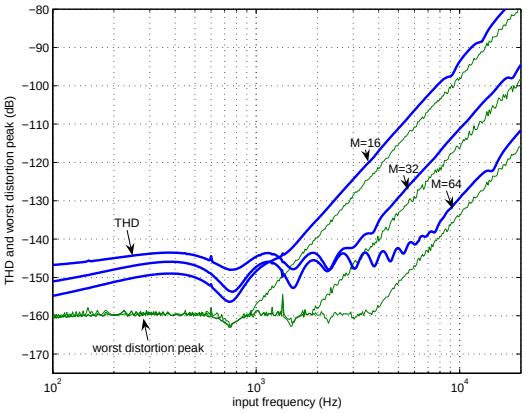


Figure 13. Interpolation error artifacts --- THD and worst-case aliasing peak amplitude vs. input sine-wave frequency for M=16, 32, and 64 subfilters. The example is for 48 kHz to 44.1 kHz conversion, quadratic coefficient interpolation, and a prototype filter with stopband attenuation of -160 dB.

5.3. Instantaneous Conversion Ratio Variation from Ideal Designed Ratio

As discussed elsewhere, this method treats the instantaneous (or actual) sampling-rate ratio separately from an ideal ratio, which is used to design the filters. It is assumed that the actual ratio will be "close" to the ideal ratio, and this is an important approximation used in the simplification of the design (see discussion in Section 2.3 on how this method relates to the Smith-Gosset method). As such, one expects the design to not be "exact" when the actual ratio differs from the designed ratio. But how is it not exact, and how does it affect the THD performance of the system?

As seen from the derivation of the subfilters, adjacent coefficients in each of the subfilters represent a sampling of the ideal prototype filter with a particular step size, which is appropriate for the ideal conversion ratio. Variations in the ideal ratio would correspond to variations in the step size for the sampling of the ideal filter. The effect of a variation in the step size can conversely be viewed as though the step size were held constant and the ideal filter was effectively expanded or contracted along the coefficient/time axis, which corresponds to a contraction or expansion of the filter frequency response along the frequency axis.

When the instantaneous conversion ratio is different from the ideal conversion ratio, the filters are designed for a different step size than the instantaneous ratio would imply, and hence implement a filter with either an expanded or contracted frequency response from that which would be best for the instantaneous ratio. Therefore, the effects of ratio variation would tend to be a breaking of the filter design specs, either by cutting off at too low a frequency or, cutting off too high and increasing the amplitude of aliasing artifacts. However, since contraction/expansion of the response will be proportional to the variation in the ratio from ideal, and since the filter responses will be smooth, whatever undesirable effects result from the variation will appear in proportion to the amount of variation, and will come in smoothly (i.e. they will gradually get worse rather than "popping in").

Therefore, the amount of ratio variation that can be tolerated can be analyzed and the prototype filter "overdesigned" by a sufficient amount, according to the amount of contraction/expansion of the prototype filter response that can be tolerated before design specs are significantly broken.

5.4. Spectral Effects of Sampling-rate Servoing

Asynchronous sampling-rate converters tend to be used in conjunction with FIFO buffers as "jitter buffers", to interface independent sample-processing systems together, with the possibility of (large or small-scale) variation in their relative time bases, along with a possible basic sample-rate difference. Such systems tend to implement some sort of servoing of the conversion ratio to absorb short-term variations, as well as handling long-term drifts. As such, the actual conversion ratio may not be a constant. Unfortunately, this variation (no matter how small) acts as a slight frequency modulation on the signal being converted, which can show up as additional sidebands in the output spectrum and adversely affect performance measurements such as THD (especially when designing an sampling-rate converters to have THDs down in the –120 to –160 dB range). This modulation is purely an effect of the servoing rather than of the underlying algorithm (and would show up regardless of the underlying method). As such, sampling-rate converter performance measurements tend to be made with any servoing disabled or removed.

6. PRODUCTIZATION

A commercial implementation of the sampling-rate conversion algorithm was created for the Analog Devices SHARC and Blackfin processor families. On the SHARC, the algorithm was implemented using 32-bit floating-point arithmetic and took advantage of the SIMD capabilities of the SHARC processor whenever possible. The Blackfin, on the other hand, is a native 16-bit processor that also has SIMD capabilities. Audio data and subfilter coefficients are stored as 32-bit fractional values and arithmetic was implemented using double precision (32-bit)[2]. The library utilizes cubic interpolation but flexibly supports different numbers of interpolation filters M, and different subfilter lengths L.

The overall sampling-rate converter "product" consists of optimized DSP code and a set of precomputed coefficient sets for typical audio conversion ratios. The coefficient sets support converting between the rates {32 kHz, 44.1 kHz, and 48 kHz}. One coefficient set supports a specified conversion ratio (e.g., 48 kHz to 44.1 kHz), and is valid for a small range of frequencies around this operating point. Coefficient sets are switched when major changes in the sampling-rates occur (e.g, originally operating at 48 kHz to 44.1 kHz, and then switching to 32 kHz to 44.1 kHz).

The product also contains a set of MATLAB design functions for computing additional coefficient sets. This allows the user to support non-standard conversion ratios or to meet specific audio performance metrics.

6.1. Key API Functions

The implementation contains a jitter buffer to deal with small variations in the input and output sampling-rates and to allow the control algorithm to react without over- or underflowing the available data. The library is designed to be used in a multi-threaded environment and has two primary processing functions. ASRC_WriteInput() accepts a block of multi-channel audio data and copies it into the state buffer of the sampling-rate converter. No other processing is performed. Next, the function ASRC_ReadOutput() is called, typically from a separate thread. This function performs the actual sampling-rate conversion and writes the converted data into an output buffer. It also

[2] To be precise, multiplications on the Blackfin were implemented with only 31-bits of precision because this reduced computation by 33% in the inner loop over a full precision 32x32→32 multiplication.

removes samples from the internal state buffer once they have been processed. Processing continues in this manner; input samples are written into the state buffer; then the samples in the state buffer are consumed as output samples are requested.

Each call to ASRC_ReadOutput() specifies the number of output samples to calculate and also f_{ratio}, the ratio of input to output sampling-rates. f_{ratio} can be adjusted from call-to-call, and in this manner the system can be adapted to time-varying sampling-rates. The audio system must also implement a control system, which tracks the input and output sampling-rates and calculates f_{ratio}. This control system is highly implementation dependent and is outside the scope of this paper.

6.2. MIPS and Memory Usage

This section provides examples of the performance of the sampling-rate conversion algorithm under typical operating conditions. The examples assume that the prototype filters are designed with 130 dB of stopband attenuation and 0.025 dB of passband ripple.

FS in	FS out	M	L	C	End of passband (Hz)	THD+N (dB)	Peak spur (dB)
32000	32000	32	66	2,310	13,440	-116.5	-125.9
44100	32000	32	66	2,310	12,472	-117.4	-129.6
48000	32000	32	70	2,450	12,400	-115.6	-123.8
32000	44100	32	66	2,310	13,440	-118.0	-130.1
44100	44100	32	66	2,310	18,522	-116.5	-125.9
48000	44100	32	62	2,170	17,970	-116.4	-126.9
32000	48000	32	66	2,310	13,440	-117.7	-129.1
44100	48000	32	66	2,310	18,522	-117.8	-130.5
48000	48000	32	66	2,310	20,160	-116.5	-125.9

Figure 14. Table of filter coefficient parameters for a variety of standard audio sampling rates.

Figure 14 shows typical coefficient sets and is applicable to both the SHARC and Blackfin. The columns should be interpreted as follows:

- M – number of subfilters.
- L – length of each subfilter.
- C – number of 32-bit memory words used to store coefficients for all of the subfilters (equal to $(M+3)L$).
- Edge of passband – the highest frequency that can be passed through the sampling-rate converter algorithm without attenuation.
- THD+N – total amount of noise in the output (dB).
- Peak spur – peak distortion in the output (dB).

The data memory is dominated by the coefficient table(s) and the internal state buffer. The coefficient table sizes correspond to column "C" in Figure 14**.** The state buffer size is a function of several parameters. Let B_{in} and B_{out} represent the maximum input and output block sizes and let $jitterSize$ equal the number of additional samples to be used as a jitter buffer. Also let $nChannels$ denote the number of channels processed. Then, the total length of the state buffer is:

$$nChannels \times \left(L + \max\left(B_{in}, B_{out}\right) + jitterSize\right)$$

in 32-bit words.

On the SHARC, code side totals 1,144 words of 48-bit program memory and on the Blackfin 1,980 bytes. Several of the functions provided are for reading and writing data in a variety of formats and not all functions are typically used in a single application.

The MIPS consumed is a function of the input and output block sizes and the number of channels processed. In general, larger blocks yield more efficient implementations, and in this analysis we assume that $B_{in} = B_{out} = 32$ samples.

Figure 15 and Figure 16 show the MIPS consumed by the algorithm on the SHARC and Blackfin processors, respectively. Note that there is a large computational cost incurred for the first channel, but that each subsequent channel increases computation by only about 20%. This is the key benefit of our approach, and yields great efficiency when multiple channels are processed. For example. Converting 6 channels from 48 kHz to 44.1 kHz requires 19.9 MIPS on the SHARC and 67.1 MIPS on the Blackfin.

		Number of channels					
FS in	FS out	1	2	3	4	5	6
32000	32000	7.6	9.1	10.6	12.2	13.7	15.3
44100	32000	7.6	9.1	10.6	12.2	13.7	15.3
48000	32000	7.9	9.5	11.1	12.7	14.3	15.9
32000	44100	10.4	12.5	14.5	16.6	18.7	20.7
44100	44100	10.4	12.5	14.5	16.6	18.7	20.7
48000	44100	9.9	11.9	13.9	15.9	17.9	19.9
32000	48000	11.3	13.5	15.8	18.0	20.3	22.5
44100	48000	11.3	13.5	15.8	18.0	20.3	22.5
48000	48000	11.3	13.5	15.8	18.0	20.3	22.5

Figure 15. Table of MIPS consumed on the SHARC processor for a variety of standard input and output sample ranges.

		Number of channels					
FS in	FS out	1	2	3	4	5	6
32000	32000	28.4	33.0	37.6	42.2	46.9	51.5
44100	32000	28.4	33.0	37.6	42.2	46.9	51.5
48000	32000	30.0	34.9	39.8	44.7	49.6	54.5
32000	44100	39.1	45.5	51.9	58.2	64.6	71.0
44100	44100	39.1	45.5	51.9	58.2	64.6	71.0
48000	44100	37.0	43.0	49.1	55.1	61.1	67.1
32000	48000	42.6	49.5	56.4	63.4	70.3	77.2
44100	48000	42.6	49.5	56.4	63.4	70.3	77.2
48000	48000	42.6	49.5	56.4	63.4	70.3	77.2

Figure 16 Table of MIPS consumed on the Blackfin processor for a variety of standard input and output sample ranges.

7. CONCLUSION

The sampling-rate conversion algorithm described in this paper provides a computationally efficient method of performing asynchronous conversion in software on a modern digital signal processor. The algorithm combines fractional sample delay filters with cubic polynomial interpolation. The key innovation is that a large part of the computation is shared among channels and thus yields a low incremental cost for converting additional channels; roughly 20% per additional channel. The algorithm has been productized by Analog Devices and is available for both SHARC and Blackfin processors. The performance of the algorithm has been demonstrated through simulation and is currently used in several commercial products.

8. ACKNOWLEDGEMENTS

This work was supported by Analog Devices, Inc. and was performed at the Audio Rendering Technology Center in San Jose as part of the VisualAudio development effort. Special thanks to David Jaffe who was instrumental in helping refine the details of the algorithm's APIs.

9. REFERENCES

[1] Oppenheim, A. V., and Schafer, R. W, "Discrete-time Signal Processing", Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[2] Parks, T. W., and Burrus, C. S., "Digital Filter Design", John Wiley and Son, New York, NY, 1987.

[3] T.A Ramstad "Digital Methods for Conversion Between Arbitrary Sampling Frequencies". IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-32, No. 3, June 1984

[4] J. O. Smith and P. Gosset: "A Flexible Sampling-Rate Conversion Method" ICASSP-84, Volume II, pp. 19.4.1-19.4.2. New York, IEEE Press.

[5] J. O. Smith "Digital Audio Resampling Page," www-ccrma.stanford.edu/~jos/resample/

[6] Crochiere and Rabiner. *Multirate Digital Signal Processing,* Englewood Cliffs, NJ; Prentice-Hall, Inc., 1983.

[7] A. I. Russell and P. E. Beckmann "Efficient Arbitray Sampling-rate Conversion With Recursive Calculation of Coefficients". IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. 50, No. 4, April 2002

[8] C. W. Farrow "A Continuously Variable Digital Delay Element". Proc. 1988 IEEE In. Symp. Circuits Syst. (Espoo, Finland), pp. 2641-2645, June 1988.

[9] J. Vesma and T. Saramäki "Design and Properties of Polynomial-Based Fractional Delay Filters" ISCAS 2000 - IEEE Int. Symp.on Circuits and Systems, , Geneva Switzerland, pp. I-104 – 107. May 2000.

[10] J. Vesma "A Frequency-Domain Approach to Polynomial-Based Interpolation and the Farrow Structure" IEE Transaction on Circuits and Systems – II: Analog and Digital Signal Processing, Vol. 47, No. 3, March 2000

[11] Analog Devices, Data Sheet for AD1896, 192 kHz Stereo Asynchronous Sampling-rate Converter.