Application Report
SPRA867 – December 2002

Parametric Equalization on TMS320C6000 DSP

Remi Payan
Catalog DSP

ABSTRACT

This application report details the implementation of a multiband parametric equalizer on the TMS320C6000™ DSP platform. The entire application is written in standard C; it reaches an excellent level of performance and allows user to control the equalizer through a graphic interface on the host computer. The purpose of this report is to demonstrate how TI DSP products and tools can be used in professional audio applications, and to propose solutions for such systems. The first part is dedicated to the design of the filter bank, its associated equations, coding, optimization, and benchmark; the second part shows how TI tools can leverage the integration of this module in a realistic professional audio environment.

Contents

List of Figures

List of Tables

1 Introduction

By announcing its recent TMS320C6713 audio digital signal processor (DSP), Texas Instruments has shown its engagement in delivering outstanding performance to the professional audio world, while maintaining an affordable cost.

This application report details the implementation of a multiband parametric audio equalizer on the TMS320C6000 platform. It is based on 32-bit floating-point processing (IEEE 754 single precision format), and optimized first for multiple cascaded biquad filters, and second for block-based processing.

A parametric equalizer is a filter bank where each filter can be tuned. Parameters are:

- Filter type including low-shelf, hi-shelf, peak, and eventually low-pass and high-pass frequency
- Gain in decibel (dB)
- Center (peak), mid-point (low-shelf, hi-shelf) or cut-off (low-pass, high-pass) frequency
- Quality factor (resonance)

The parametric equalizer is different than a graphic equalizer, where each filter selects a fixed band of frequency, and users can only adjust the gain in that band. The parametric equalizer shown in this report features a graphic user interface (GUI) based on real-time data exchange (RTDX™) technology provided by TI tools.

2 Implementation

A parametric equalizer provides a finite set of filters that users can tune. Each filter is in fact a second order infinite impulse response (IIR) filter in which the coefficients are calculated according to given parameters. This section details the structure retained for the application and the way it has been optimized, then discusses coefficients calculation, and finally exposes the performances obtained in terms of cycle count and noise level.

2.1 Filtering Equations

2.1.1 Filter Topology

The accuracy and stability of IIR filters depend on their topology. A good topology takes care of limiting accumulator overflow, and minimizing error feedback in the structure. As mentioned in [2], some topologies are recommended for audio applications. For the purpose of this document, a direct form II transpose is used, which takes into account these considerations.

2.1.1.1 Biquad Filter

The core of an equalizer is a biquad filter, i.e., a second order recursive filter (IIR). Figure 1 illustrates a standard audio topology (*direct form II transpose*), assuming normalization by a0.

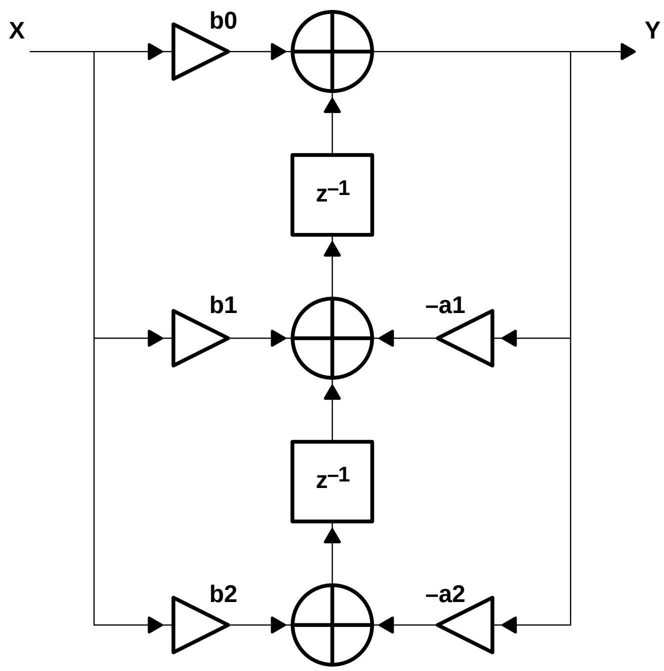


Figure 1. Biquad Filter, Direct Form II Transpose

The associated discrete transfer function is: $H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 \cdot z^{-1} + b_2 \cdot z^{-2}}{1 + a_1 \cdot z^{-1} + a_2 \cdot z^{-2}}$.

In the time domain, this translates into the following equation:

$$y(n) = b_0 \cdot x(n) + b_1 \cdot x(n-1) + b_2 \cdot x(n-2) - a_1 \cdot y(n-1) - a_2 \cdot y(n-2)$$

By calculating the coefficients, b_0, b_1, b_2, a_1, a_2, we can define the actual type and effects of the filter (see section 2.1.2).

In order to minimize the number of operations needed in the processing function, coefficient **b0** can be factored out of the whole biquad structure. When cascading the filters, this coefficient can be pre-calculated for the entire cascaded chain once only.

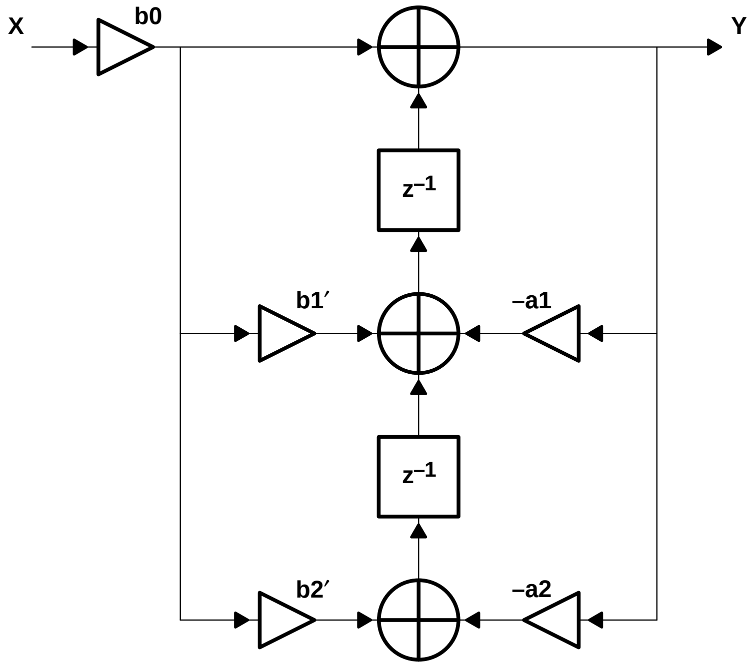


Figure 2. Biquad Filter, Direct Form II Transpose, b0 Factorization

The new values for **b1** and **b2** are: $b_1{'} = \frac{b_1}{b_0}$, $b_2{'} = \frac{b_2}{b_0}$.

2.1.1.2 Cascaded Biquad Filters

A multiband parametric equalizer requires several biquad filters to be cascaded. Factorizing all **b0** coefficients from all filters out of the whole cascaded structure is now possible, as shown in Figure 3.

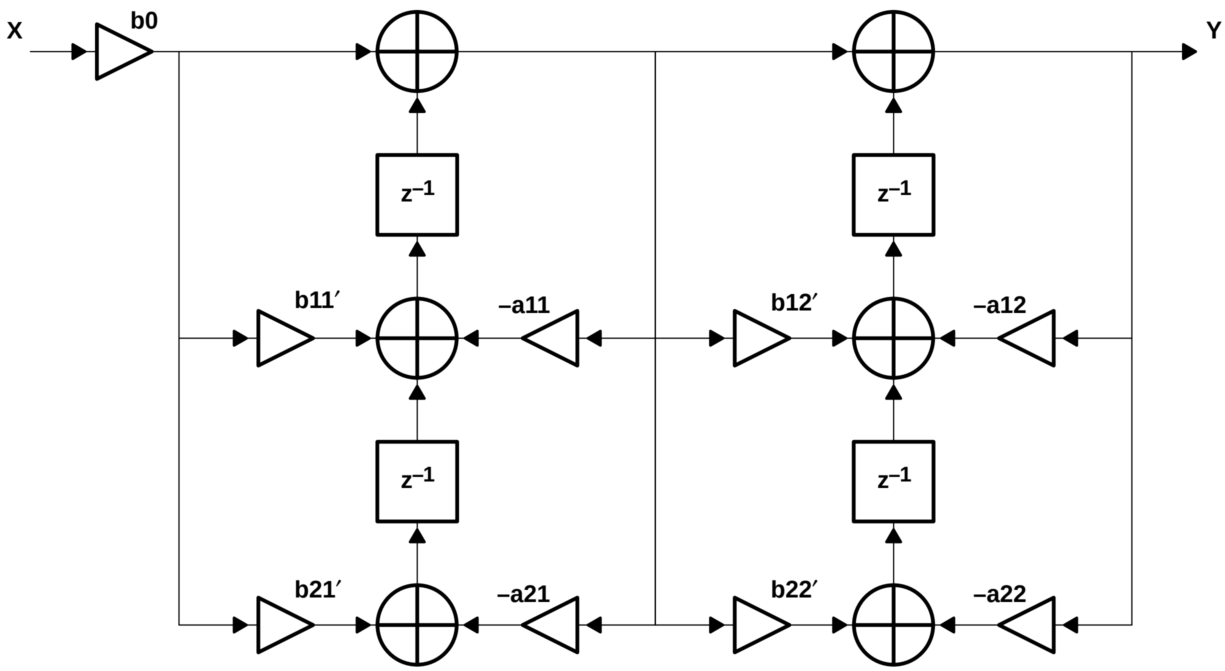


Figure 3. 2 Cascaded Biquad Filters, Direct Form II Transpose, b0 Factorization

In this figure, coefficients of the first biquad have been suffixed with a (1), whereas coefficients of the second biquad have been suffixed with a (2). A prime sign (′) means the coefficient is biased due to structural modifications. Compared to direct form II coefficients as defined in section 2.1.1.1, new coefficients are defined as:

$$b_0' = b_{0,1} \cdot b_{0,2}$$

$$b_{1,1}' = \frac{b_{1,1}}{b_{0,1}},\ b_{2,1}' = \frac{b_{2,1}}{b_{0,1}}$$

$$b_{1,2}' = \frac{b_{1,2}}{b_{0,2}},\ b_{2,2}' = \frac{b_{2,2}}{b_{0,2}}$$

In other words, there is only 1 **b0** coefficient per cascaded biquad structure, and 4 more coefficients (**b1**, **b2**, **a1**, **a2**) per biquad.

2.1.2 Coefficients Computation

In a parametric equalizer, five types of filters are typically required: shelving (low-shelf and high-shelf), peaking, and band-pass (low-pass and high-pass) filters. Their descriptions are given in terms of gain (for peaking and shelving filters), central frequency (for peaking filters), mid-point frequency (for shelving filters) or cut-off frequency (for band-pass filters), and quality factor. Appendix C shows frequency response curves associated to each type of filter, for various settings.

A typical method to transform those physical parameters into numerical coefficients is to start from the analog description of the filters in the Laplace domain, and to apply a bilinear transform, while taking into account frequency pre-warping (frequency axis distortion induced by the bilinear transform). This method [1] gives the results described in Appendix A.

Such computations, including trigonometric and exponential functions, cannot be done in real time. In systems requiring flexibility in those parameters, pre-computed coefficients tables are widely used. However, we will see further how TI DSP tools allow performing those calculations in the background, while the CPU continues to focus on its main processing task.

2.2 Coding and Optimization

This section describes the coding and optimization of the multiband equalizer function. It does not details the whole optimization process; for this purpose, please refer to [3].

2.2.1 Cascaded Biquad Filters

The code below shows an implementation of a cascaded biquads structure. The loop iterates on consecutive biquad filters. Notice the “restrict” keyword is used to specify that pointers are not aliased (they do not point to the same memory locations).

This core loop treats only four coefficients per biquads. It is assumed that the input has previously been (or will be later) scaled by b0, as shown in section 2.1.1.2.

```
float biquad_c(
int numBiquad,
const float * restrict c,
float * restrict d,
float y
)
{
    int i;
    float d0, d1, c0, c1, c2, c3, temp0, temp1;
    const double *c_ptr = (double *)c;
          double *d_ptr = (double *)d;

    for (i = 0; i < numBiquad; i++) {
        d0 = _itof(_lo(d_ptr[i]));
        d1 = _itof(_hi(d_ptr[i]));
        c0 = _itof(_lo(*c_ptr));
        c1 = _itof(_hi(*c_ptr++));
        c2 = _itof(_lo(*c_ptr));
        c3 = _itof(_hi(*c_ptr++));

        temp0 = d0 + c0*y;
        temp1 = c1*y;
        y += d1;

        d[2*i+1] = temp0 + c2*y;
        d[2*i+0] = temp1 + c3*y;
    }
    return y;
}
```

In this code, the variables c_ptr and d_ptr are pointers to double words (64 bits), respectively, to the coefficients array and the filter history array (delayed values). This is to insure the use of the double-word load capabilities of the TMS320C67xx architecture. Intrinsics permit separating the high and low parts of the loaded double word (_hi and _lo), and re-interpreting the result as single-precision floating-point values (_itof).

The compiler feedback (see section B.1) mentions an iteration interval of 4 cycles for this software-pipelined loop. This means a new iteration starts every 4 cycles.

2.2.2 Block Processing

The multiband equalizer consists of several cascaded biquad filters; we now need to modify the code in order to process a buffer of samples.

The first solution consists of adding an outer “for” loop, that would read a new input value before the inner loop and store the result afterwards. In that case, only the inner loop will software pipeline, and its prolog and epilog will be repeated within the outer loop [3].

A better solution consists of merging the inner and outer loops into one single loop, and using conditional statements to update the input values, indexes and pointers, as shown in the code below (the setup code has been removed).

```
y = in[0];

for (i=0,j=0,k=0; j < Ns*Neq; i++,j++)
{
       if (i==Neq)
       {
         i=0;
         c_ptr =  (double *)c;
         y = in[++k];
       }

       d0 = _itof(_lo(d_ptr[i]));
       d1 = _itof(_hi(d_ptr[i]));
       c0 = _itof(_lo(*c_ptr));
       c1 = _itof(_hi(*c_ptr++));
       c2 = _itof(_lo(*c_ptr));
       c3 = _itof(_hi(*c_ptr++));
       temp0 = d0 + c2*y;
       temp1 = c3*y;
       y += d1;

       d[2*i+1] = temp0 + c0*y;
       d[2*i+0] = temp1 + c1*y;

       out[k] = b0 * y;
    }
}
```

Section B.2 shows the compiler feedback for this code. The iteration interval is 7 cycles, which means the reload of the values impacted the code by adding 3 cycles to the iteration interval of the loop.

2.2.3 Stereo Processing

In a typical stereo equalizer, the same filtering is applied to both left and right channels. Performing the calculations on both sides at the same time can further optimize a stereo equalizer. The source code below shows the main processing loop for such an equalizer.

```
yL =  in[0];
yR =  in[Ns];

for (i=0, j=0, kL=0, kR=Ns; j < Ns*Neq; i++,j++)
{
       if (i==Neq) {
         i=0;
         c_ptr = (double *)c;
         d_ptr = (double *)d;
         yL =  in[++kL];
         yR =  in[++kR];
       }

       d0L = _itof(_lo(d_ptr[2*i+0]));
       d0R = _itof(_hi(d_ptr[2*i+0]));

       c0 = _itof(_lo(*c_ptr));
       c1 = _itof(_hi(*c_ptr++));
       c2 = _itof(_lo(*c_ptr));
       c3 = _itof(_hi(*c_ptr++));

       temp0L = d0L + c2*yL;
       temp0R = d0R + c2*yR;

       temp1L = c3*yL;
       temp1R = c3*yR;

       d1L = _itof(_lo(d_ptr[2*i+1]));
       d1R = _itof(_hi(d_ptr[2*i+1]));

       yL += d1L;
       yR += d1R;

       d[4*i+2] = temp0L + c0*yL;
       d[4*i+3] = temp0R + c0*yR;
       d[4*i+0] = temp1L + c1*yL;
       d[4*i+1] = temp1R + c1*yR;

       out[kL] = b0 * yL;
       out[kR] = b0 * yR;
}
```

This routine treats both left and right buffers (actually the same buffer, first half filled with Ns samples from the left channel, and the second half with samples from the right channel).

The compiler feedback (section B.3) reports an iteration interval of 12 cycles, which is less than twice the number of the mono version. Parallelism has been increased. It also points out that ideally, it could be further optimized down to 8 cycles from a resource partitioning and dependency point of view.

2.3 Benchmarks

2.3.1 Performance

Measurements of the CPU load were made using DSP/BIOS CPU load graph. Table 1 and Table 2 expose the results, in percentage of the CPU load and in megahertz per band (MHz), with respect to frame size and number of biquad filters used. The formula used is:

$$MHz/band = \frac{CPUload(\%).10^{-2}}{NumBands} \cdot 150$$

The idle load represents the overhead of the framework, including transformation of fixed-point samples from the codec into floating-point representation. Only the processing code and data (filter coefficients and buffers) sit in internal memory. All the rest is linked into external memory (SDRAM), and 1 way of L2 cache is activated, i.e., the higher 16kbytes section of internal memory is configured as cache memory.

Table 1. CPU Load versus Frame Size and Number of Biquads

Multiband Equalizer Benchmark (% CPU Load)				
	Frame Size			
	8	**16**	**32**	**64**
	Idle Load			
Num Bands	**10.98**	**7.13**	**5.17**	**4.27**
4	2.26	1.8	1.64	1.56
8	3.59	3.26	3.07	3.01
16	6.57	6.16	5.93	5.88
32	12.26	11.93	11.73	11.63
64	24.08	23.49	23.24	23.12
128	48.67	47.69	47.23	46.88

Table 2. MIPS per Band versus Frame Size and Number of Biquads

Multiband Equalizer Benchmark (MHz per Band)				
	Frame Size			
Num Bands	**8**	**16**	**32**	**64**
4	0.848	0.675	0.615	0.585
8	0.673	0.611	0.576	0.564
16	0.616	0.578	0.556	0.551
32	0.575	0.559	0.550	0.545
64	0.564	0.551	0.545	0.542
128	0.570	0.559	0.553	0.549

As expected, performance increases along with the number of biquads and the frame size. At some point (here, 128 biquad filters), it slightly decreases due to cache effect. Thrashing increases along with the amount of data to process. No cache optimization was achieved in this demo project (refer to [4] for TMS320C6000 cache architecture details).

As mentioned in section 2.2.3, the stereo processing code could be further optimized down to 8 cycles per loop, in case the application would justify spending time on this optimization. This would reduce the MIPS numbers shown above to 8/12 (around 67 %) of their respective values.

2.3.2 Signal-to-Noise Ratio (SNR)

This section describes the method used to measure the SNR of the filters implemented in the application, and comments on the results shown in Appendix D.

2.3.2.1 Test Bench

Figure 4 represents a diagram of the software system that was designed to perform SNR measurements. The whole test bench was realized on the TMS320C6000 platform.

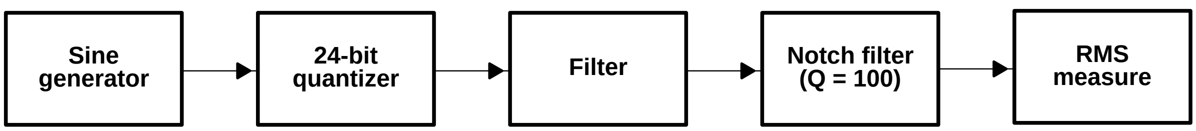


Figure 4. THD+N Measurement Software System

A 24-bit quantized sine generator is built using the double precision sin() function from the standard C math library. The signal is quantized in a 24-bit fixed-point format in order to fit into the format of an ADC converter, and then transformed into single precision floating-point representation to feed the input of the benchmarked filter.

The output, consisting in the filtered signal plus additional noise, is transformed into 64-bit double-precision floating-point format, and passed through a notch filter. The center frequency of the notch filter exactly equals the frequency of the generated sine. Its quality factor is set to 1000, which means the stop band spreads around 1/1000th of the center frequency. In order to deal with the latency of the notch filter, at least four frames of 32768 samples are generated before the output noise is measured. Both the notch filter and the computation of its coefficient are performed in double precision format.

The RMS level of the remaining signal at the output of the notch filter corresponds to the total harmonic distortion plus noise (THD+N) of the system, at the sine frequency. This latter is then swept logarithmically across the frequency axis to obtain the spectrum of the THD+N.

The SNR is calculated according to the following formula:

$$SNR_{(dB)} = 20.log\left(\frac{\sqrt{2}}{2}\right) + 20.log\,(|H(z)|) - (THD + N)_{(dB)}$$

The first term represents the RMS level of the sine. The second term corresponds to the amplification brought by the filter; the sum of these two terms equals the level of the signal (sine) at the output of the filter. The third is the THD+N measurement, i.e., the level of noise.

Figure D–1 in Appendix D shows the output of the test bench when no processing is performed, and when an all-pass filter is used. Both curves overlay exactly as expected. A 144 dB SNR is observed, which corresponds to the 24-bit quantization representing the ADC.

2.3.2.2 Results

Figure D–2, Figure D–3, Figure D–4, and Figure D–5 in Appendix D represents the SNR curves for each type of filter. A set of five frequencies logarithmically spread across the audio spectrum is tested: 100Hz, 300Hz, 1kHz, 3kHz and 10 kHz, using positive and negative gains (when applicable). The quality factor of the filter is set to typical values: Q=1 for shelving and band-pass filters, and Q=4 for peaking filters.

Results show that as the mid-point/center/cut-off frequency of the filter decreases, the SNR decreases as well. Actually, as the frequency decreases, the coefficients tend towards specific values (0.5, 1, 1.5 or 2), while increasing the amount of feedback in the biquad structure. Hence, the feedback error increases similarly due to the lack of accuracy of the coefficients. Improvements of the SNR could be obtained by using feedback error correction mechanisms [2].

3 DSP/BIOS Integration

This section describes how the multiband stereo equalizer routine is embedded in a DSP/BIOS object-oriented application, providing real-time kernel services, as well as real-time analysis tools and controls.

3.1 Frame Size and System Latency

System latency is a concern in most professional audio applications. In order to minimize it, single sample processing is widely used, impacting the whole application performance by increasing the overhead, induced by non-processing routines such as interrupt service routines (ISR).

Today, the best high-end professional audio systems feature a few milliseconds of latency (2 to 5 milliseconds). Converters represent a huge portion of this number, especially analog to digital converters (ADC). In order to reach high precision (24 bits) they use sigma-delta techniques, which are based on over-sampling and finite impulse response (FIR) decimation filtering. Those filters typically have latency between 10 and 50 output samples. This number is doubled for a codec chip (integrated ADC and DAC), which represents a total between 400 microseconds and 2.1 milliseconds at 48 kHz (20 and 100 samples, respectively).

The most recent digital signal processors, such as the TMS320C6000 family, perform better in block-based processing, thanks to both increased parallelism and pipeline capabilities. Any block-based process has an implicit latency of twice the size of the block. First, the system must wait for a complete block to be received; second, because of the real-time constraint, the processing routine must complete before the next block is available. Applied to audio domain, it means a 32 or 64 samples block is still reasonable to consider. This is why a block size of 32 samples was used in the multiband equalizer demo presented in this document, which represents 1.3 milliseconds of latency.

Furthermore, block-based processing leaves more headroom for the system to include background processing. The next sections will show how to use TI tools and foundation software to improve both the overall quality and the features of the system.

3.2 DSP/BIOS-Based Software Architecture

DSP/BIOS is a free of charge, real-time scalable kernel available across all TI platforms. It includes a task scheduler (TSK module), a hardware-interrupt service routine dispatcher (HWI module), and a software-interrupt module (SWI), as well as inter-process communication modules such as semaphores and queues.

In the multiband equalizer demo (Figure 5), the enhanced direct memory access (EDMA) engine, available in all TMS320C6000 DSPs, is in charge of buffering the incoming audio samples. The HWI module is responsible for servicing the interrupt request sent by the EDMA engine every frame, and posts a software-interrupt to signal to the main processing routine that a new buffer is ready.

Beside this main processing chain, two tasks are running in the background. The first one deals with the communication between target and host machines. The second one is in charge of computing and providing the processing routine with new filter coefficients, whenever the user makes changes to the parameters from the GUI on the host.

In order to avoid data access conflicts, two sets of coefficients are created. One is reserved to the process function, while the design function writes to the second. At each frame, the process function checks a flag indicating the availability of new coefficients, and swaps the pointers if required. These operations represents only a few cycles in the process function, while background processing allows real-time calculation of the coefficients.

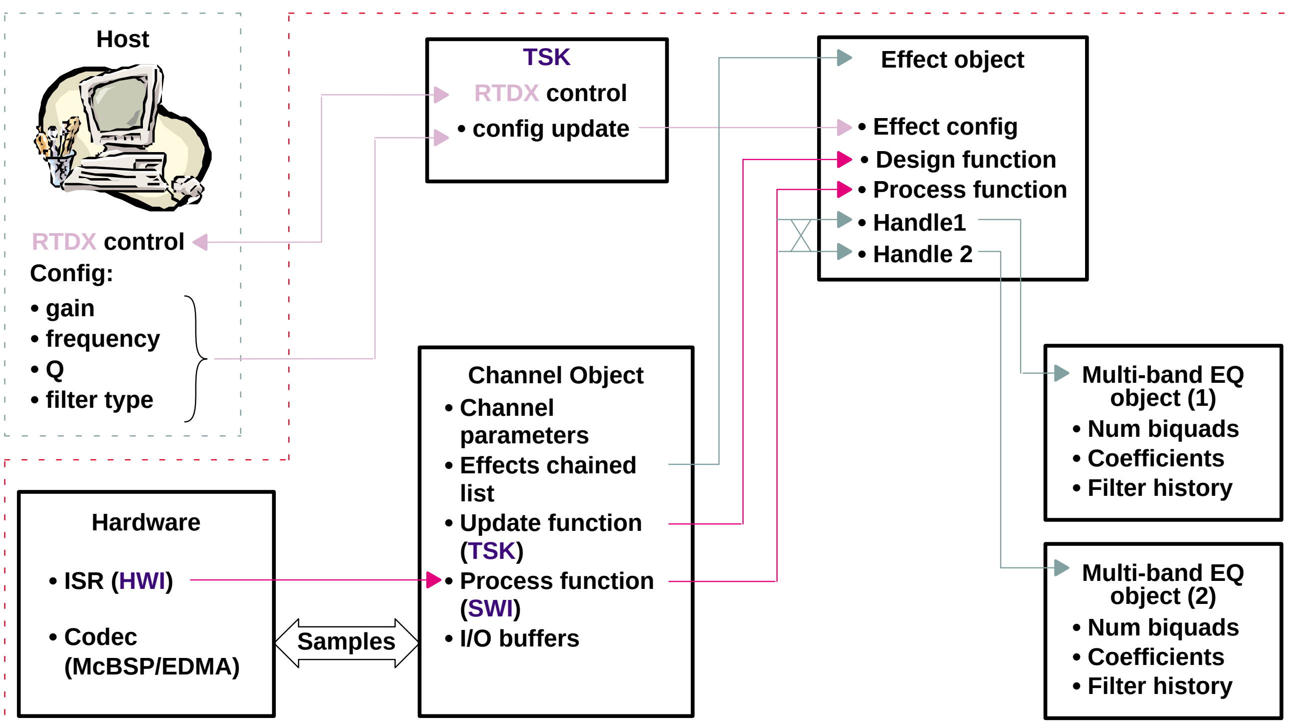


Figure 5. Multiband Equalizer Demo Software Architecture

4 Multiband Equalizer Demo

4.1 Getting Started

This section describes the steps required to set up the demo. A TMS320C6711 DSK is required, and optionally, a TLV320AIC23 daughter board (24 bits/96 kHz codec). Instructions are provided to run the demo on both hardware setups.

4.1.1 Hardware Setup

If only the DSK is used, you only need to connect the DSK correctly to the host computer by connecting an audio source to the microphone connector, and a pair of speakers to the headphone connector.

If the daughterboard is used, the jumper JP1 must be inserted on the DSK. This jumper is not present on a standard DSK; you will need to solder it yourself. It is located between the DSK daughterboard connectors, and the on-board codec microphone/headphone connector. Plug the daughter card on the DSK connectors. Connect an audio source to the “line-in” RCA connectors, and a pair of amplified speakers to the “line-out” RCA connectors (refer to the daughterboard documentation). If the headphone/microphone connectors of the AIC23 are to be used, steps are provided in section 4.1.2 to enable them in the software.

4.1.2 Software Setup

It is assumed the zip file was extracted to drive c:\. If different, just replace it with the path used.

1. Register the graph control OCX.

   The host application uses a custom graph control OCX to plot the response curve of the filter. This Visual Basic™ control needs to be registered in the windows registry. Two windows batch files are provided; click on the one corresponding to your operating system (Windows9X or NT/2000). Those files are located in the directory:

   C:\MultiEqDemo\bin

2. Run the target application.

   a. Project configuration

      Under Code Composer Studio, open the multi-eq project:

      C:\MultiEqDemo\projects\MultiEq.pjt

      Once the project opened, right-click on it in Code Composer Studio project window, and select “Options”. This opens the options dialog box. Select the “compiler” tab, and in the “category” list-box, select “preprocessor”.

      By default, the project is set up for AIC23. A symbol “_AIC23_” shall be defined in the “Defined symbols” text box. If the AIC23 is not used, remove this symbol. Click OK to validate the change.

   b. AIC23 configuration

      The AIC23 configuration uses default values defined in the file:

      C:\MultiEqDemo\drivers\aic23_hal.h

      These default values can be changed using the hardware abstraction layer (HAL) defined in this file. Refer to the AIC23 documentation for this purpose.

c. Frame size setting

   The frame size can be modified through the "FRAME_SIZE" compile constant located in the file:

   C:\MultiEqDemo\project\Multieq\codec.h

d. Rebuild the target application

   Rebuild the application by clicking "Rebuild all" from the "Project" menu in Code Composer Studio.

e. Load the target application

   From the "File" menu in Code Composer Studio, select "Load Program". The program file is located in:

   C:\MultiEqDemo\project\MultiEq\Debug\MultiEq.out

   This should load the program file to the target.

f. Enable RTDX

   In order for the host and target to communicate through RTDX, it must be enabled. This is done implicitly when starting the CPU load graph, or the message LOG window. Any or both of them can be found in the "DSP/BIOS" menu in Code Composer Studio.

g. Run the target application

   From the "Debug" menu in Code Composer Studio, select "Run".

3. Start the host application.

   The executable file for the host application is:

   C:\MultiEqDemo\bin\MultiEqHost.exe

   The demo traces the target behavior through the LOG window. When the host application is started, the first step is the verification of the host/target communication. A loop-back test is performed to check all the RTDX channels in use. Then, the host application inquires the target for the channel parameters (sample rate, stereo processing…), and the equalizer effect is dynamically created, using default parameters. Once all these steps are achieved, the user interface is launched.

4.2 Host Application

Figure 6 shows the graphic user interface of the host application. It allows setting the number of bands of the equalizer, enabling and disabling the effect, and setting parameters for each band.

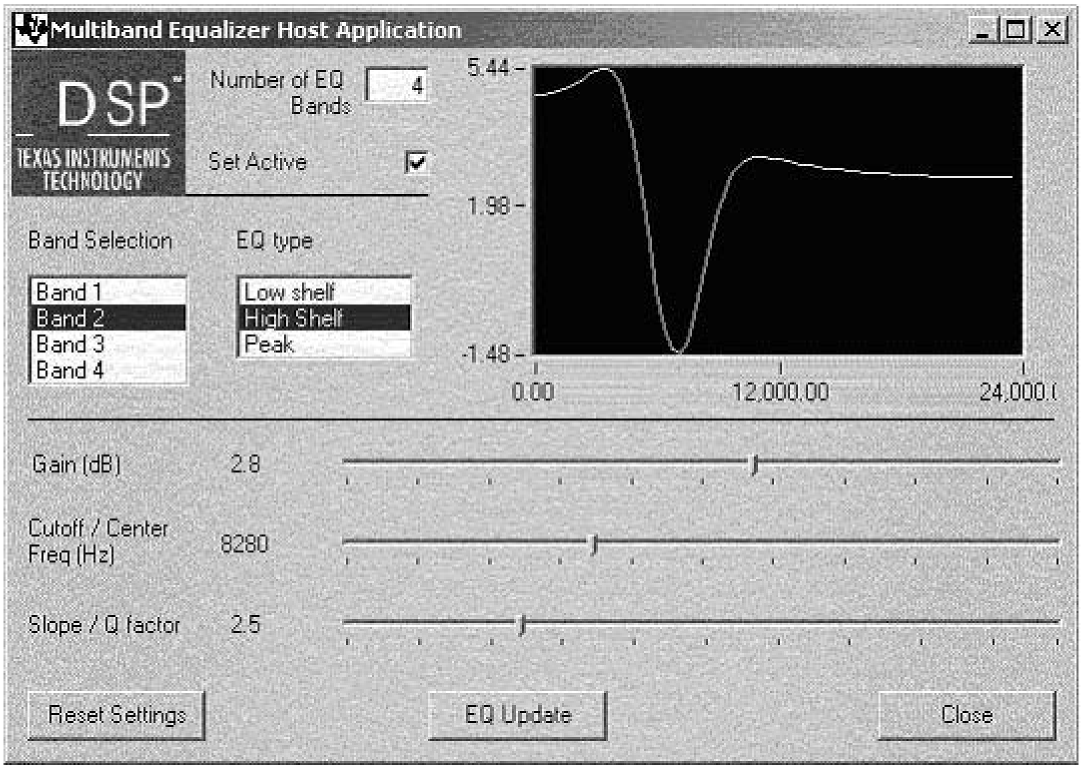


Figure 6. Host Application Graphic User Interface

Whenever changes are made, press the “EQ update” button to send the new parameters to the target. The frequency response is computed on the target in the background, and sent back to the host for display (linear scaling).

On the AIC23 running at 48 kHz, up to 192 bands can be used. The CPU load is then around 97%. When updating the settings for such a number of bands, it may take some time for the target to answer to the host, due to the little headroom left to the background functions by processing. At that time, a dialog box may pop-up asking for “Retry” or “Cancel.” Click on Retry to recover the host to target communication. Click on Cancel to terminate the host application; in this case, the DSK needs to be reset before restarting it.

At any time, if the host application is terminated properly by clicking the “Close” button, it can be restarted without any other action on the target.

5 References

1. *The Equivalence of Various Methods of Computing Biquad Coefficients for Audio Parametric Equalizers,* Robert Bristow-Johnson (electronic document)
   http://www.harmony-central.com/Effects/Articles/EQ_Coefficients/EQ-Coefficients.pdf
   http://www.harmony-central.com/Computer/Programming/Audio-EQ-Cookbook.txt
2. *The Implementation of Recursive Digital Filters for High-Fidelity Audio,* Jon Dattorro, Journal of the Audio Engineering Society, Vol. 36, No. 11, November 1988
3. *TMS320C6000 Optimizing C Compiler User’s Guide* (SPRU187)
4. *TMS320 C621x /C671x Two-Level Internal Memory Reference Guide* (SPRU609)

Appendix A Biquad Coefficients Computation

This appendix details the results of the calculation of biquad filter coefficients from its analog description.

The bilinear transform consists in substituting the Laplace variable (s) in the analog transfer function H(s), in order to obtain an equivalent digital transfer function H(z).

$$s = \frac{1}{\tan\left(\frac{w}{2}\right)} \cdot \frac{1 - z^{-1}}{1 + z^{-1}}$$

This formula includes frequency pre-warping. Trigonometric identities are used to simplify the equations.

Parameters:

F_s is the sample rate.

F_c is the center (peak) or midpoint (shelf) frequency.

g is the gain.

Q is the quality factor (peak) or slope (shelf).

Intermediate variables:

$$A = 10^{\frac{g}{40}},\ w = 2\pi \cdot \frac{F_c}{F_s},\ \sin = \sin(w),\ \cos = \cos(w),\ \alpha = \frac{\sin}{2.Q},\ \beta = \frac{\sqrt{2.A}}{Q}$$

Then the coefficients for the 5 types of filter are:

Low-shelf filter:

$$b_0 = A.\left[(A+1) - (A-1).\cos + \beta.\sin\right] \qquad a_0 = (A+1) + (A-1).\cos + \beta.\sin$$
$$b_1 = 2A.[(A+1) - (A-1).\cos] \qquad a_1 = -2.[(A-1) + (A+1).\cos]$$
$$b_2 = A.\left[(A+1) - (A-1).\cos - \beta.\sin\right] \qquad a_2 = (A+1) + (A-1).\cos - \beta.\sin$$

High-shelf filter:

$$b_0 = A.\left[(A+1) + (A-1).\cos + \beta.\sin\right] \qquad a_0 = (A+1) - (A-1).\cos + \beta.\sin$$
$$b_1 = -2.A.[(A-1) + (A+1).\cos] \qquad a_1 = 2.[(A-1) - (A+1).\cos]$$
$$b_2 = A.\left[(A+1) + (A-1).\cos - \beta.\sin\right] \qquad a_2 = (A+1) - (A-1).\cos - \beta.\sin$$

Peaking filter:

$$b_0 = 1 + \alpha.A \qquad a_0 = 1 + \frac{\alpha}{A}$$
$$b_1 = -2.\cos \qquad a_1 = -2.\cos$$
$$b_2 = 1 - \alpha.A \qquad a_2 = 1 - \frac{\alpha}{A}$$

Low-pass filter:

$$b_0 = 1 - \frac{\cos}{2} \qquad a_0 = 1 + \alpha$$
$$b_1 = 1 - \cos \qquad a_1 = -2.\cos$$
$$b_2 = 1 - \frac{\cos}{2} \qquad a_2 = 1 - \alpha$$

High-pass filter:

$$b_0 = 1 + \frac{\cos}{2} \qquad a_0 = 1 + \alpha$$
$$b_1 = -(1 + \cos) \qquad a_1 = -2.\cos$$
$$b_2 = 1 + \frac{\cos}{2} \qquad a_2 = 1 - \alpha$$

Appendix B Optimization: Compiler Feedbacks

B.1 Single Sample Cascaded Biquad Routine

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line               : 32
;*      Loop opening brace source line : 32
;*      Loop closing brace source line : 59
;*      Known Minimum Trip Count         : 2
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 4
;*      Unpartitioned Resource Bound     : 3
;*      Partitioned Resource Bound(*)    : 3
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     2        2
;*      .S units                     1        0
;*      .D units                     3*       2
;*      .M units                     2        2
;*      .X cross paths               2        3*
;*      .T address paths             2        3*
;*      Long read paths              1        1
;*      Long write paths             1        2
;*      Logical  ops (.LS)           0        0     (.L or .S unit)
;*      Addition ops (.LSD)          0        1     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             2        1
;*      Bound(.L .S .D .LS .LSD)     2        2
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 4  Schedule found with 6 iterations in parallel
;*      done
;*
;*      Epilog not removed
;*      Collapsed epilog stages     : 0
;*
;*      Prolog not removed
;*      Collapsed prolog stages     : 0
;*
;*      Minimum required memory pad : 0 bytes
;*
;*      For further improvement on this loop, try option –mh64
;*
;*      Minimum safe trip count     : 6
;*----------------------------------------------------------------------------*
```

Notice the minimum safe trip count is 6, which means this software pipelined loop can only run at least 6 cascaded biquads. Compiler generates an alternate code, less optimized, that is used whenever the loop count is less than 6. This extra code can be removed by adding a #pragma compiler directive, assessing that the loop count will always be at least 6. Then it is the user’s responsibility to fulfill this requirement when calling the function.

B.2 Block Cascaded Biquad Routine

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line               : 147
;*      Loop opening brace source line : 148
;*      Loop closing brace source line : 171
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 6
;*      Unpartitioned Resource Bound     : 5
;*      Partitioned Resource Bound(*)    : 6
;*      Resource Partition:
;*                                A–side   B–side
;*      .L units                     2        3
;*      .S units                     0        1
;*      .D units                     4        5
;*      .M units                     4        1
;*      .X cross paths               1        3
;*      .T address paths             4        3
;*      Long read paths              1        2
;*      Long write paths             2        1
;*      Logical  ops (.LS)           0        0     (.L or .S unit)
;*      Addition ops (.LSD)          5        9     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             1        2
;*      Bound(.L .S .D .LS .LSD)     4        6*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 6  Did not find schedule
;*         ii = 7  Schedule found with 4 iterations in parallel
;*      done
;*
;*      Epilog not entirely removed
;*      Collapsed epilog stages     : 1
;*
;*      Prolog not entirely removed
;*      Collapsed prolog stages     : 1
;*
;*      Minimum required memory pad : 0 bytes
;*
;*      Minimum safe trip count     : 2
;*----------------------------------------------------------------------------*
```

The loop carried dependency has increased versus (a), due to conditionally updating input and output values. But Epilog and Prolog have one stage removed, and the minimum safe trip count is only 2.

B.3 Stereo Block Cascaded Biquad Routine

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line               : 203
;*      Loop opening brace source line : 204
;*      Loop closing brace source line : 241
;*      Known Minimum Trip Count         : 4
;*      Known Maximum Trip Count         : 4
;*      Known Max Trip Count Factor      : 4
;*      Loop Carried Dependency Bound(^) : 8
;*      Unpartitioned Resource Bound     : 8
;*      Partitioned Resource Bound(*)    : 8
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     5        4
;*      .S units                     1        3
;*      .D units                     7        5
;*      .M units                     5        5
;*      .X cross paths               4        7
;*      .T address paths             6        6
;*      Long read paths              3        3
;*      Long write paths             2        2
;*      Logical  ops (.LS)           0        2     (.L or .S unit)
;*      Addition ops (.LSD)         11        7     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             3        5
;*      Bound(.L .S .D .LS .LSD)     8*       7
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 8  Did not find schedule
;*         ii = 9  Cannot allocate machine registers
;*                   Regs Live Always   :  4/6  (A/B-side)
;*                   Max Regs Live      : 19/17
;*                   Max Cond Regs Live :  2/1
;*         ii = 9  Did not find schedule
;*         ii = 10 Register is live too long
;*         ii = 11 Register is live too long
;*         ii = 12 Schedule found with 3 iterations in parallel
;*      done
;*
;*      Epilog not entirely removed
;*      Collapsed epilog stages     : 1
;*
;*      Prolog not entirely removed
;*      Collapsed prolog stages     : 1
;*
;*      Minimum required memory pad : 0 bytes
;*
;*      Minimum safe trip count     : 1
;*----------------------------------------------------------------------------*
```

NOTE: This compiler feedback was manually modified to fit into the page.

This is the stereo version of (b). The iteration interval is less than twice as much as in the previous example (7*2 = 14 versus 12). From a resource partitioning and dependency standpoint, we can see that the iteration interval could ideally be optimized further down to 8 cycles, which is only 1 cycle more than the mono version while performing twice as much.

Appendix C Filters Frequency Responses

This appendix shows frequency responses of all types of filters, with various settings. Mid-point/center/cut-off frequency is set to 1 kHz for all filters.

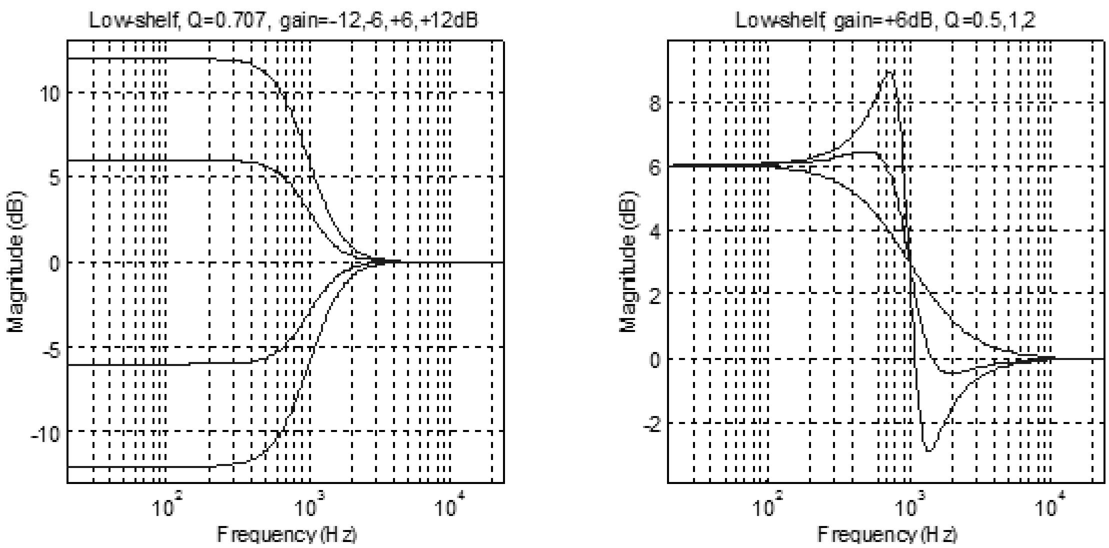


Figure C–1. Low-Shelf Filters Frequency Responses

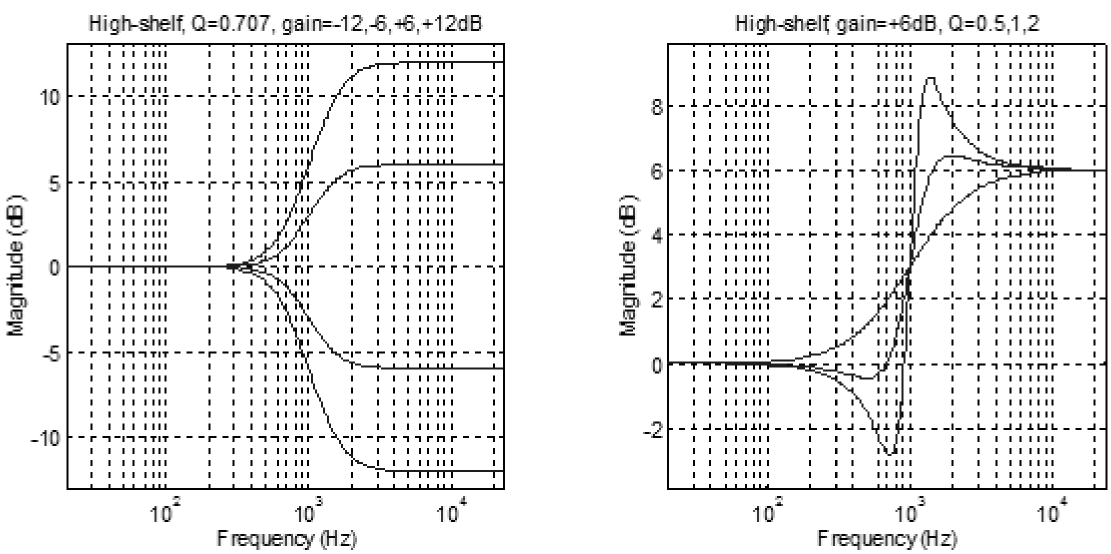


Figure C–2. High-Shelf Filters Frequency Responses

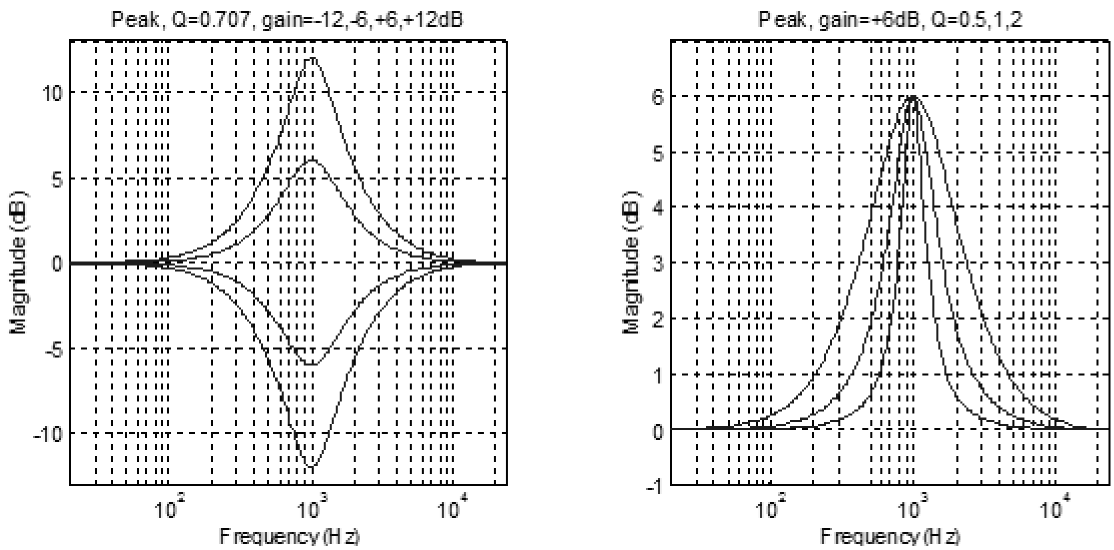


Figure C–3. Peaking Filters Frequency Responses

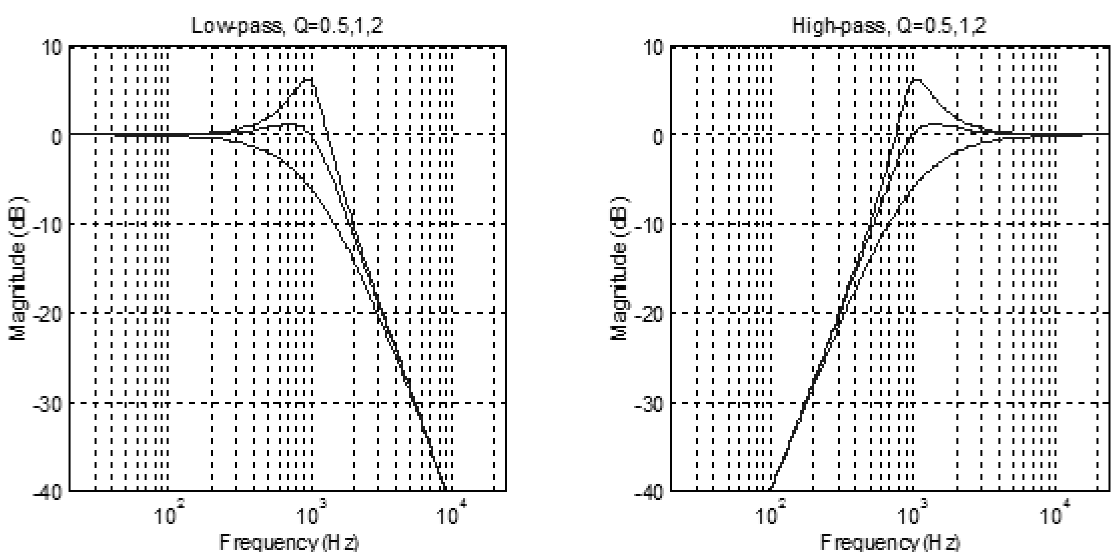


Figure C–4. Band-Pass Filters Frequency Responses

Appendix D Signal to Noise Ratio (SNR) Curves

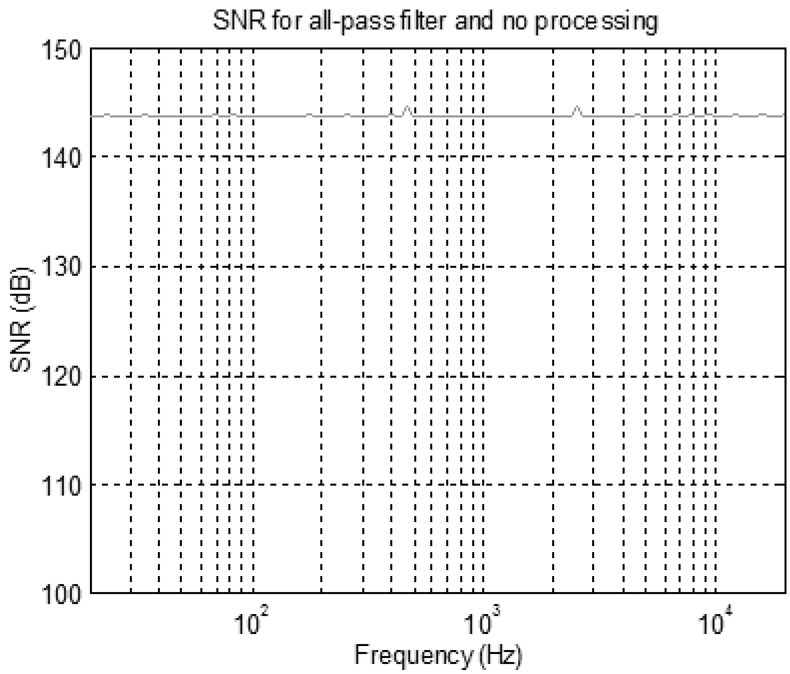


Figure D–1. SNR Without Processing

Two curves are plot on this graph: the first one was obtained by bypassing the filter in the system, i.e., the notch filter applies directly on the generated sine; the second one was obtained by using all-pass coefficients (in fact, a low-shelf filter with gain set to 0 dB). The noise is exclusively caused by the 24-bit quantization. The SNR equals 144 dB, and the two curves overlay exactly, as expected.

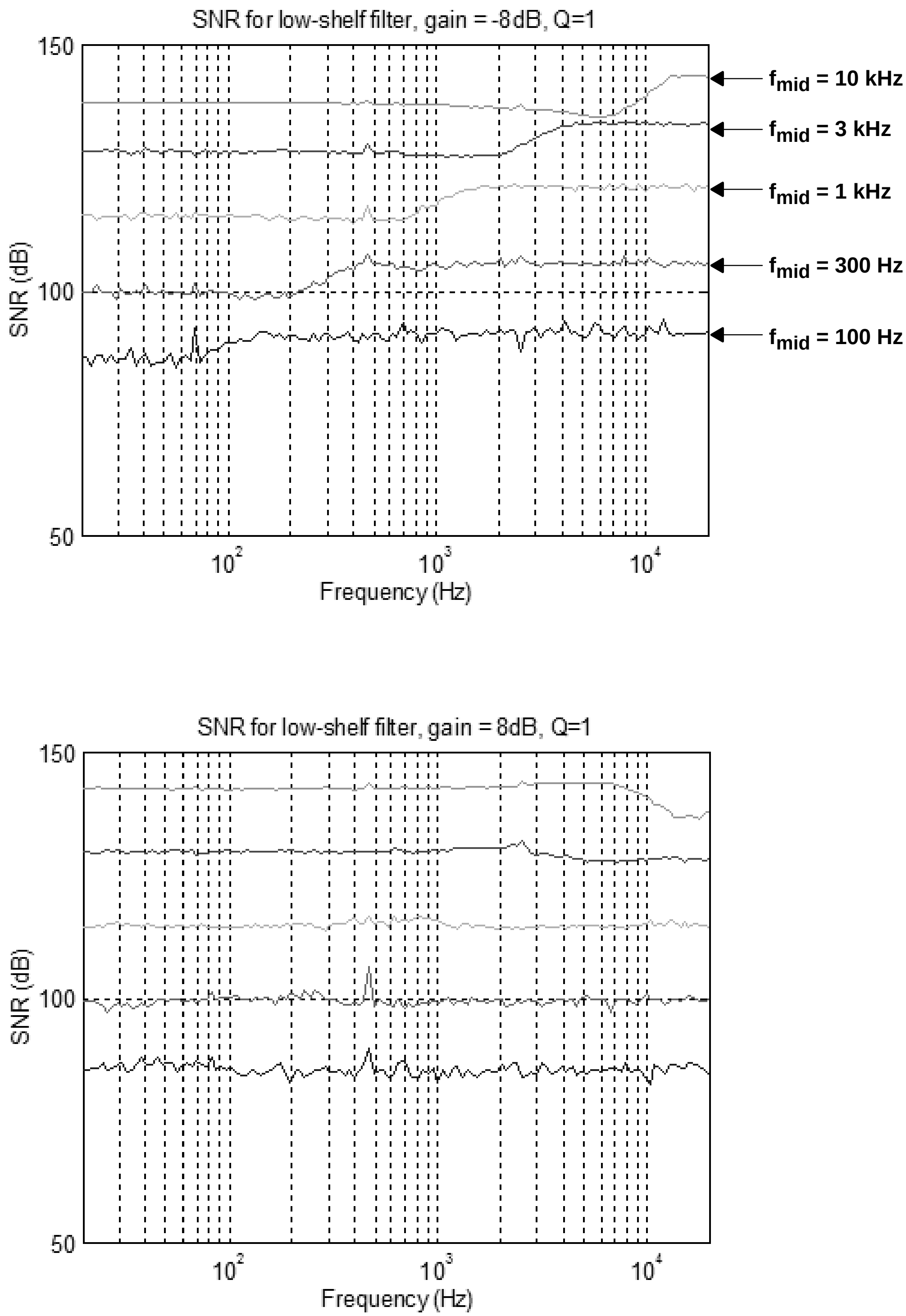


Figure D–2. SNR for Low-Shelf Filters

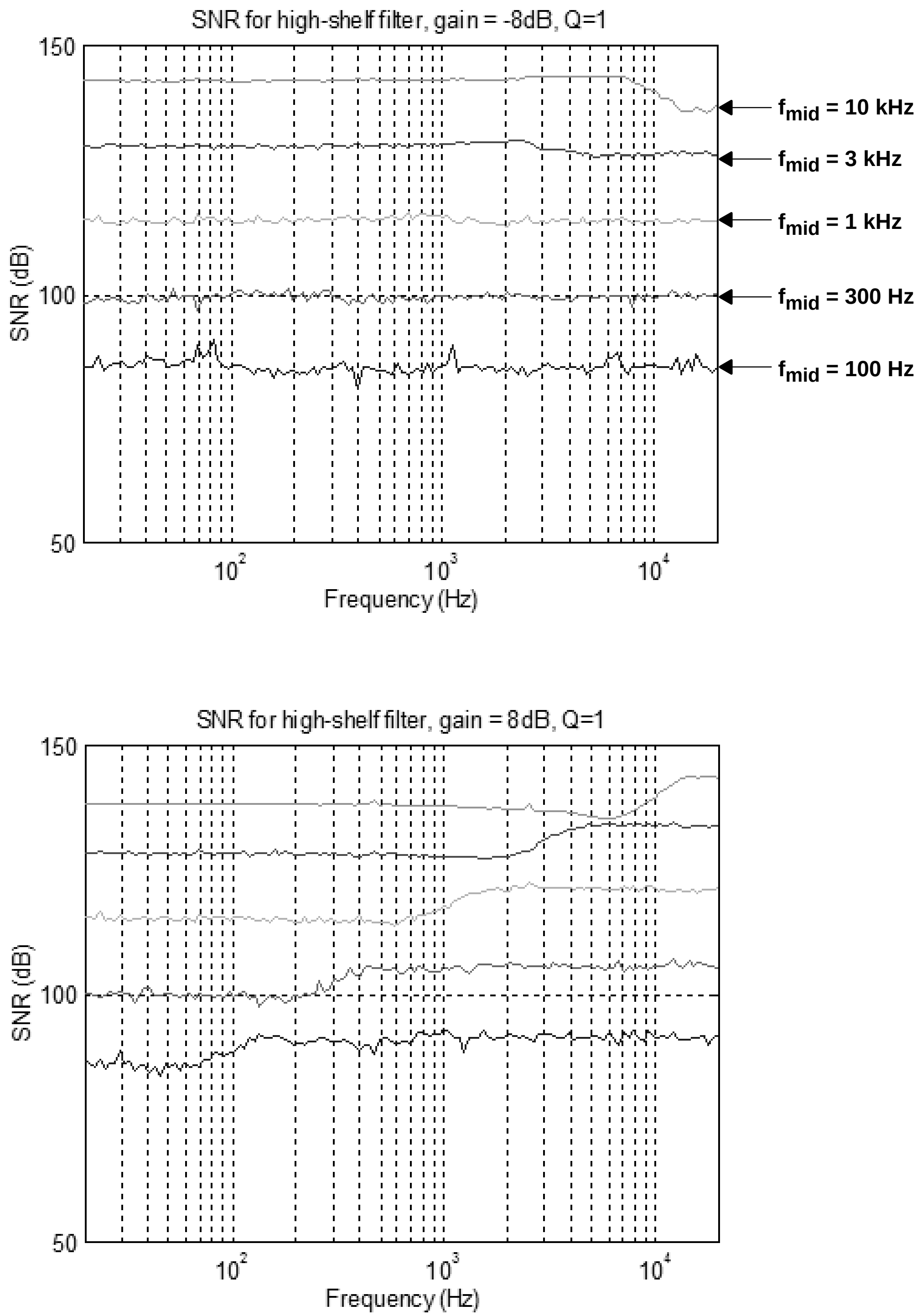
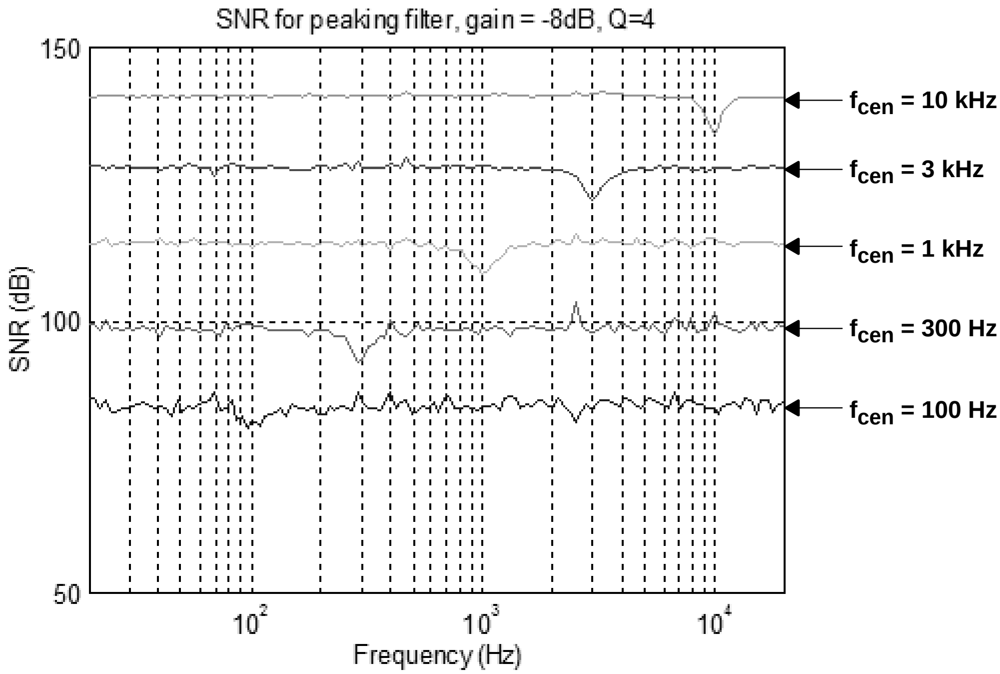


Figure D–3. SNR for High-Shelf Filters



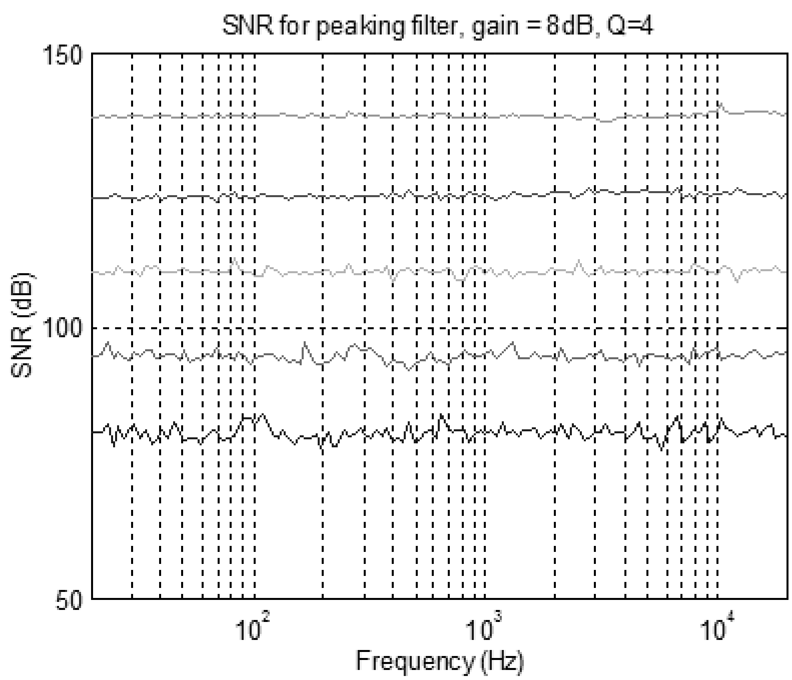
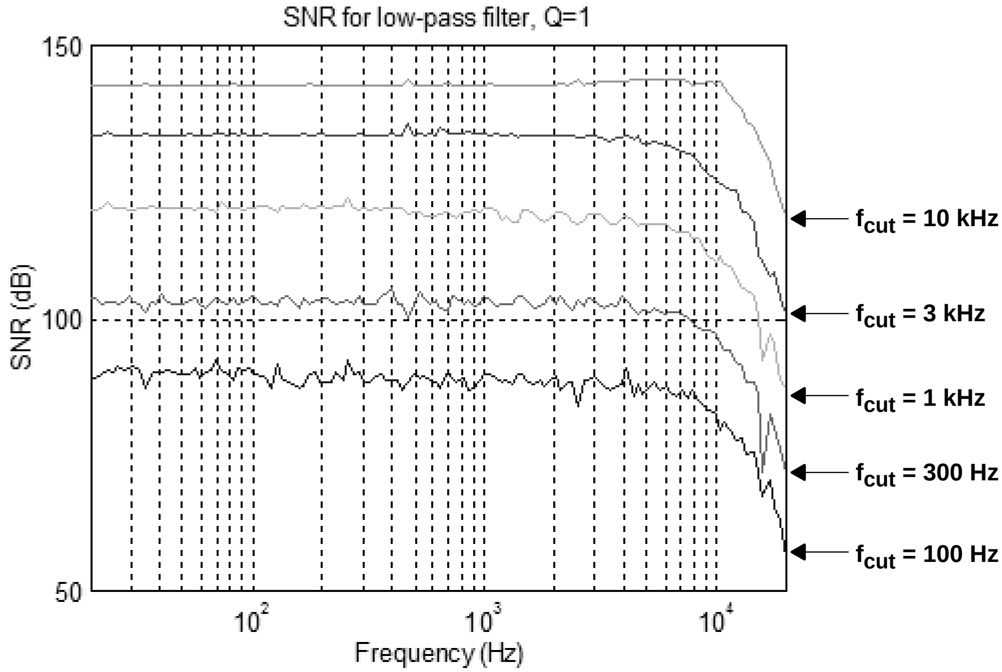


Figure D–4. SNR for Peaking Filters



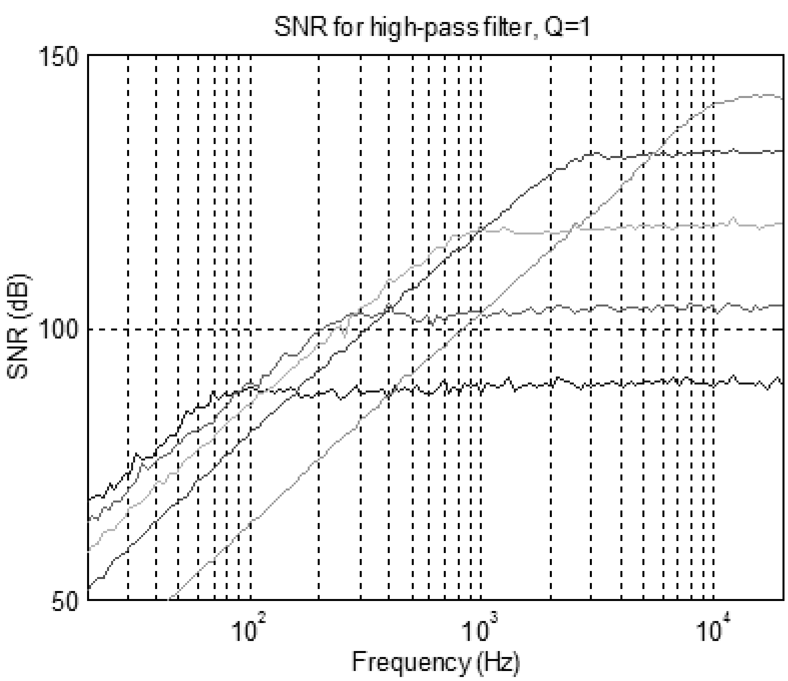


Figure D–5. SNR for Band-Pass Filters

IMPORTANT NOTICE

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265