APPLIED AUDIO DIGITAL SIGNAL PROCESSING: A HANDS-ON APPROACH

Michael Gay
Purdue University
West Lafayette, Indiana

ABSTRACT

Radio engineers have always been on top of technology relating to broadcast equipment. Broadcast engineers could design custom filters, make circuit changes and whatever else was necessary to get quality sound. Now that digital signal processing (DSP) has taken over the world, broadcast engineers have found themselves lacking in the ability to understand the new digital equipment at the level they understand the old analog gear. This is not due to a lack of trying. Most books on the subjects are glorified calculus books that don't cover the meat and potatoes of DSP. The broadcast engineer used to design projects with a hand full of capacitors, resistors and solder. DSP chips don't seem very friendly, and has anyone tried to solder one of these things with a 30-watt soldering iron recently?

DSP processors typically come in surface mount packages with high-density pin counts. It is impractical for the average broadcast engineer to work with this type of hardware. However, there is a little known product available from the manufacturers of most DSP processors. It is called a development board.

Development boards are self-contained systems consisting of the DSP processor, an EPROM for programs, an audio codec, and computer interface. Programming is performed using a PC based computer to write, compile, and upload a program to the DSP processor for execution. These boards typically come with sample programs, which can be built upon and modified to design a custom system.

This paper will start the reader out with a quick tutorial on DSP hardware and common algorithms including digital filters.

Next this paper will touch upon PC based DSP applications and then describe some of the latest self-contained DSP hardware solutions.

Finally, the reader will be taken step by step through the process of using a development board, starting with a sample program and creating an audio processor that performs a function useful to the radio broadcast engineer.

INTRODUCTION

Digital signal processing (DSP) theory has been around for quite some time in the mathematical world. It has only been in the past 10-15 years that DSP has migrated from a mathematical science to an applied science via specialized microprocessors. DSP processors are popping up in almost every piece of broadcast equipment imaginable, including audio processing, digital effects boxes, audio consoles, and more. Even the RF realm, once sacred to the broadcast engineer, has surrendered to new digital IF sections and other digital modulation techniques.

Unfortunately, engineers who once were kings of the radio castle, now look upon these DSP based wonders as if they were watching a magician sawing the woman in half. They know there is a trick somewhere and that anyone should be able to do it with enough practice. The magicians of DSP have been guarding their secrets well, though. Broadcast engineers used to be able to open a piece of equipment and, from the schematic diagram, know its every working. They could think like the analog audio circuits inside. They could even design and build their own equipment, or build on someone else's work by making modifications to existing platforms. No longer is that the case. Or is it?

TYPICAL AUDIO APPLICATIONS

Broadcasters are seeing more and more digital equipment. Some are intimidated by it, while others have embraced it. In any case there are some key applications the broadcaster might encounter. Some DSP audio applications that might be used by the broadcaster are:

- Dynamics Processing
    - Compression
    - Limiting
    - Expansion
    - Clipping
- Filters
    - Low Pass
    - High Pass
    - Band Pass
    - Custom Equalization
- Reverb & Delay Lines

- Echo Cancellation
- Real time FFT Computation
    - Pitch shifting
    - Spectral analysis
    - Adaptive Filtering

There are more applications other than the ones listed above. However, these listed are more commonly found in existing broadcast audio equipment.

Dynamics Processing

It was a little over 10 years ago when DSP based dynamic processing made its debut into mainstream broadcast audio processing with the introduction of Audio Animation's Paragon, Gentner's Lazer and Prizm, and Orban's Optimod 8200. Obviously some were successes while others are now audio processing history. These processors were the first to implement dynamics processing entirely within the digital realm of the DSP processors. Since the inception of these processors, there has been a long-standing debate over whether DSP-based dynamics processing sounds better than its analog predecessors. This is a very subjective topic and beyond the realm of this paper, yet still worth mentioning.

While the processing algorithms and DSP hardware have made significant improvements since that time, the basics of the dynamics processing algorithms have remained the same.

Any DSP-based audio compressor algorithm will look at the level of the input audio and make appropriate gain adjustments to keep the output within a particular range. The threshold levels and the gain reduction ratio define the method in which these adjustments are made. What varies from algorithm to algorithm is the math used to calculate the amount of gain reduction and attack/release times. This is where each of the manufacturers of these devices adds their "art."

Filters

The ability to implement digital filters is one of the greatest strengths of digital signal processing as a science. Filters can be designed with roll-offs that are not feasible to build in the analog world. Custom filters with extremely odd responses that might take an analog designer many hours to design, and days to implement and tweak, can be designed and implemented in a matter of minutes using DSP computer aided design tools.

Reverb & Delay Lines

In the analog world it took specialized "bucket brigade" delay chips or spring contraptions to make an effective reverb. A reverb is created using several delay lines, of which the outputs are added back to the original signal to create an echo effect. Delaying audio is one of the simpler tasks within DSP. Adding audio signals together would be a close second in simplicity. An analog profanity delay-line is not unheard of, but would be very impractical today. Now virtually every profanity delay uses DSP technology.

Echo Cancellation

Echo cancellation requires the processor to look at the audio going out, compare to the audio coming back, make assessments, and implement a canceling signal to counteract the echo. This is commonly used in speakerphones and two-way videoconferencing facilities. It is also becoming common in digital wireless phones.

Real-time FFT Computation

The ability to compute the Fast Fourier Transform (FFT) in real time has been a key breakthrough in DSP hardware development. The FFT is a highly efficient method of computing the Discreet Fourier Transform (DFT). The DFT is the method of performing the Fourier Transform on discreet time data, commonly known as samples. This is important because if the time domain data can be converted to frequency domain data, audio can be manipulated in ways truly impossible in the analog world. This has led to algorithms such as frequency shifting, frequency detection, noise reduction, new methods of dynamics processing, and more. Here are a few audio applications, which use a real-time FFT computation as part of their algorithm.

Pitch shifting

If an audio stream can be converted to the frequency domain, shifting its pitch is simply a matter of shifting the spectral information of the frequency domain and then converting back to the time domain.

Spectral analysis

Imagine trying to build a 1024 band spectrum analyzer using analog components. With DSP it is a matter of computing a 1024-point FFT.

Adaptive Filtering

Using the FFT, a DSP chip can analyze a burst of pink noise and adapt a filter to compensate for abnormalities. This can already be seen in today's commercial digital broadcast telephone hybrids.

Feedback elimination can be accomplished in the same way. Feedback elimination requires extremely steep cutoffs and very low bandwidth filters. This is well suited for DSP. The FFT can be used to detect where the filtering is required and to what extent it needs to be implemented.

INTRODUCTION TO DIGITAL FILTERS

The heading of this section could literally be the title of a book. There is so much information regarding digital filters that this section will barely scratch the surface. There are many detailed books on the subject of digital filter design. However, very few of these actually cover implementation. This section will introduce some basic concepts behind two of the most common digital filter types and discuss some of the advantages and disadvantages of each.

IIR Filters

Infinite-Impulse Response (IIR) filters can be thought of as a digital implementation of an equivalent analog filter. Another name for this type of filter is "recursive filter." This is due to the fact that this filter uses feedback similar to an analog circuit. This is evident in the typical IIR filter block diagram in Figure 1.

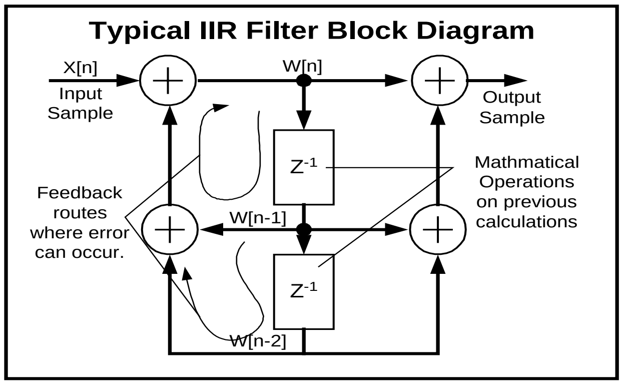

Typical IIR Filter Block Diagram
X[n]
Input Sample
W[n]
Output Sample
Z-1
Mathmatical Operations on previous calculations
Feedback routes where error can occur.
W[n-1]
Z-1
W[n-2]

Figure 1. Typical IIR Filter Block Diagram.

The design of an IIR filter is very similar to designing an analog filter. First, the order and frequency response of the filter need to be determined based on the needs of the end user. Next, a process known as "Frequency Pre-warping" is performed to account for distortions in the frequency response that will occur in the final digital filter. The amount of this shift is dependant on the sample rate of the digital system in which the filter is to be implemented. After the pre-warping, the transfer function of the filter needs to be determined using the pre-warped specification. Lastly, a mathematical process known as the "bilinear transform" is applied to the analog transfer function to convert its poles and zeroes from the *s-plane* of the analog world to its digital counterpart known as the *z-plane*. The coefficients for DSP code will be obtained from this new resultant transfer function.

One drawback of designing an IIR filter this way is if the analog transfer function produces a non-linear phase response, so will the digital version of that filter.

Another potential problem that can occur with an IIR filter is rounding errors. If the feedback signal is rounded off and sent back through the system and rounded off again, the rounding errors can accumulate, thereby rendering the filter unstable. An unstable digital IIR filter can oscillate, as would an unstable analog filter. These rounding errors more commonly occur in fixed point processing. For example, if two 16-bit binary numbers are multiplied together, the result is a 32-bit number. The only way to convert the information back to a 16-bit format is to truncate the least significant bits. This is the point in the process that round off error occurs. A floating-point processor is less susceptible to this type of problem, however round off error still exists.

IIR filters are still commonly used since they can be designed with Butterworth, Chebychev or Bessel responses. Conversion calculations can then be used to design the digital implementation of the filter. Since the filter is recursive, one advantage of the IIR filter is a low delay time in the system.

FIR Filters

Finite Impulse Response (FIR) filters are extremely stable in both fixed-point and floating-point environments due to the fact that any results containing round off errors (which still occur) are not fed back for another calculation. The block diagram for an FIR filter can be seen in Figure 2 below.

An FIR filter consists of a series of "taps." These are noted as h_n in the block diagram in Figure 2. The value of each tap is called its coefficient. Coefficients typically have values less than one. The number of taps required for a given filter is calculated during the filter design phase. Formulas to calculate the number of taps take into account parameters such as frequency roll off, pass-band and stop-band ripple, and sample frequency. The complete series of these coefficients represents the impulse response of a particular FIR filter.

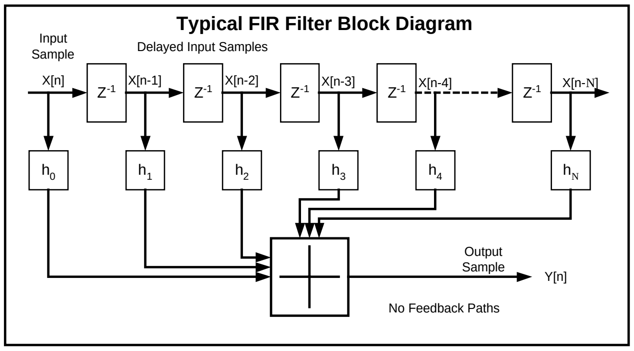


Figure 2. Typical FIR Filter Block Diagram.

The impulse response is simply a plot of the coefficients as demonstrated in Figure 3. Typically it is symmetrical about a central point tapering off on each side. These coefficients are convolved with the input samples to yield the filtered output. Linear convolution is the method by which FIR filters are implemented. As can be seen in the block diagram in Figure 3, each coefficient is multiplied by each sample as it passed through the system. Each multiplication is added to the next multiplication. This is how linear convolution works in a DSP system. Now it can be seen why the multiply-and-accumulate operation is so important to DSP.

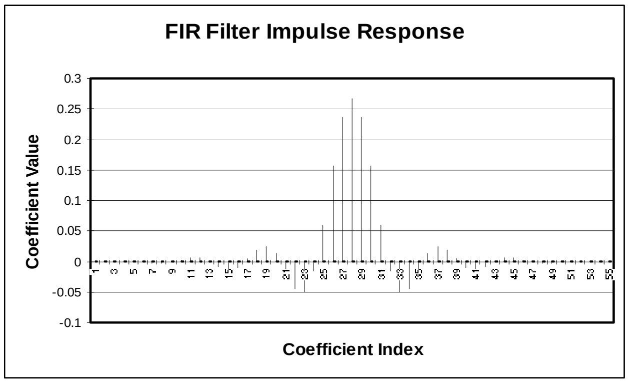


Figure 3. Typical FIR Filter Impulse Response.

One main advantage of FIR filters is true linearity of the phase response. Figure 4 contains a phase response graph of a typical digital filter. It is easy to see that the phase is a straight line.

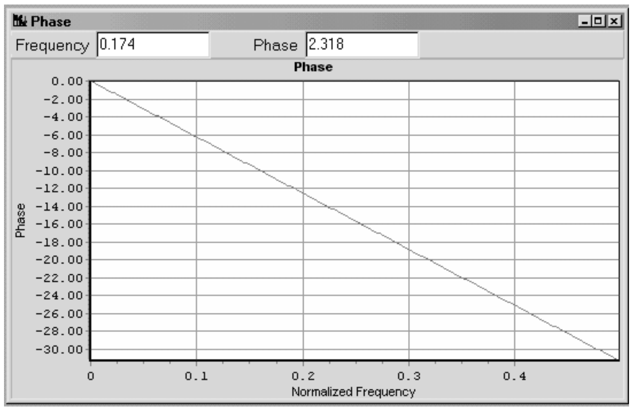

Frequency 0.174 Phase 2.318

Figure 4. Typical Phase Response of an FIR Filter.

In contrast, the phase of an IIR filter is non-linear as can be seen in the IIR filter phase graph contained in Figure 5.

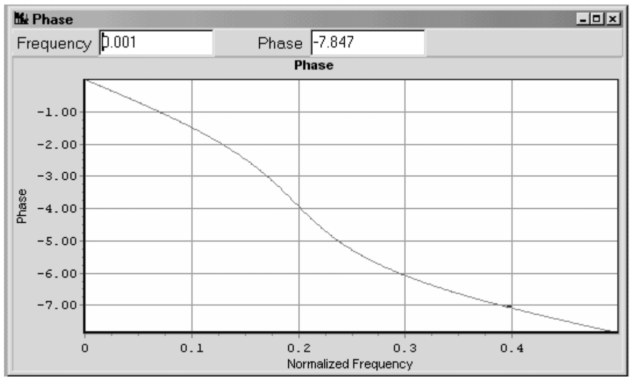

Frequency 0.001 Phase -7.847

Figure 5. Typical Phase Response of an IIR Filter.

In addition, FIR filters are less affected by bit truncation in the multiplication than their IIR counterpart.

The design of an FIR filter can be quite math intensive, requiring an extensive mastery of calculus. However, computer aided design (CAD) software is readily available to ease the design process. Some filter design software products simply require the user to enter the desired frequency response curve. The software then performs the math and tabulates the filter coefficients. Some of the more advanced software will go so far as to write a section of assembly code for particular DSP processors.

With all the advantages to the FIR filter, there must be a trade-off somewhere. In the case of the FIR filter it happens to be in the delay. The sharper the cutoff specification for a filter, the more taps that will be required to implement the filter. The delay of the filter will be equal to the number of taps required multiplied by the sample period of the digital stream. For instance:

In a particular system, the sampling frequency Fs =44.1KHz. What is the sampling period T?

$$T = \frac{1}{Fs} = \frac{1}{44.1KHz} = 22.68\mu s$$

If a particular filter has 50 taps, N=50. Multiply the sampling period by N to get the resultant system delay of the filter.

$$D = N \times T = 50 \times 22.68\mu s = 1.13ms$$

From these simple equations, it can be observed how significant amounts of audio delay can be built up after the implementation of only a few FIR filters.

FIR filters also require more memory and arithmetic than an equivalent IIR filter. This might be enough to push some single chip implementations of an algorithm beyond the limit of the processor. These factors need to be considered when designing a digital system.

DSP HARDWARE

Processor Optimization

There are two questions that always seem to be asked about hardware implementations of a DSP algorithm: Why are there DSP processors? & Why can't it all be done with a Pentium? First, DSP processors are optimized for the execution of mathematical functions common is DSP processing. Second, Pentium class processors are designed for general purpose. They do not contain the optimizations necessary for efficient DSP implementation. The different architectures should be discussed in more detail.

Von Neumann Architecture

Most general-purpose processors, like Pentium class processors, have one data bus, and one address bus. One memory space is used to hold both data and instructions. This type of setup is known as the Von Neumann architecture, after American mathematician John Von Neumann (1903-1957). Von Neumann pioneered many works in twentieth-century mathematics. He also developed the concept of a stored computer program.

In a Von Neumann architecture based processor, several instructions must be executed to fetch two data bytes, perform a multiply, and accumulate the results with a previous multiply/accumulation result. As discussed earlier, the multiply and accumulate is the basis for the implementation of an FIR filter. MMX instruction set extensions for the Pentium do allow instances where two simultaneous instructions could be executed, thereby increasing DSP algorithm performance in programs designed for the modified architecture. However, the implementation of DSP using these types of processors is simply a brute force approach. General-purpose processors are much better at DSP today simply because they can execute more instructions per second.

Harvard Architecture

Virtually all DSP processors on the market today are designed using Harvard Architecture, named after work done at Harvard in the 1940s. This design has the advantage of two data busses, two address busses, and two separate data spaces. In essence a processor can get two pieces of data simultaneously for an instruction. This increases performance in a DSP processor since the goal is to move data quickly.

Other DSP Enhancements

In addition to the enhanced architecture with the data busses, DSP chips typically have a data address generator (DAG). This is a hardware device that actually computes the next address for a pointer to a particular piece of data. This is important for use in FIR filters and FFT computation. In FFT computation in particular, the DAG performs an action called "bit reversal." This action is necessary to enhance the performance of FFT computation.

Another enhancement of the DSP chip is the multiply/accumulator (MAC). In a single instruction the MAC can multiply two pieces of data and add them to a third piece of data already in the accumulator. An FIR filter tap is calculated by performing two data fetches, a multiply and then accumulate. For a basic Pentium this activity would take four or more clock cycles. For a typical DSP chip it takes only one. Although the advantages of the optimized hardware are obvious for implementing DSP algorithms, it would not be a good idea to use a DSP chip to run an Excel™ spreadsheet.

There are more optimizations that various manufacturers use in their hardware to give themselves an edge over the competition. Typically these optimizations help to speed data transfer from one area of the processor to another. Others optimize the hardware to speed the execution of the FFT. This is why DSP selection guides and benchmarking are useful for determining the right DSP for a particular application.

HANDS ON REAL-TIME DSP

There are several ways to perform real-time processing. Three methods that are suitable for the experimenter will be covered here. These methods are:

- Personal Computer Based Processing
- Self-Contained General Purpose DSP Devices
- Manufacturer's Development Boards

Personal Computer Based Processing

The PC has come a long way in the past 10 years. Faster processor speeds have made real-time audio signal processing on a home PC simple and effective. It is so common that audio playback programs, such as WinAmp, have included graphic equalizers and spectrum analyzers as part of the free version. These same programs rely on DSP to decode the latest perceptual encoding algorithms such as MP3. The reality is that any audio compression or coding algorithms such as MP3 extensively use the FFT in computations. The PC needs to be able to compute the FFT hundreds of times per second to convert the compressed audio data into pleasing audio.

Many software multi-track editing programs now come with real-time dynamics-processing capabilities as well as equalization and noise filtering. This includes some of the latest shareware editors. As the PC processors are beginning to break the 1-Gigahertz barrier, the capabilities of audio signal processing will increase.

The brute force approach that general-purpose processors use to execute DSP algorithms is completely acceptable, just not the most efficient or cost effective means of doing so. However DSP processors will always be able to surpass a general-purpose processor of the same speed when it comes to implementing a DSP algorithm.

Self-Contained General Purpose DSP Devices

The installed sound market is currently using self-contained general-purpose DSP devices in small and large installations. While the units may not be ideal for a production room or broadcast control room environment, they can be used as problem solvers. Most have excellent filtering abilities that exceed many analog filter design capabilities. This ability would be useful in circumstances where sharp cutoff filters are required. The dynamics processing ability makes this type of equipment a great general-purpose level controller.

In applications such as speaker delay systems, these devices are extremely simple to set up. Typically the user enters the required delay in feet and leaves the mathematics to be handled by the DSP device.

Generally these devices are addressable and daisy-chainable to allow one computer to program and operate multiple devices.

Symetrix 9022 DSP Engine

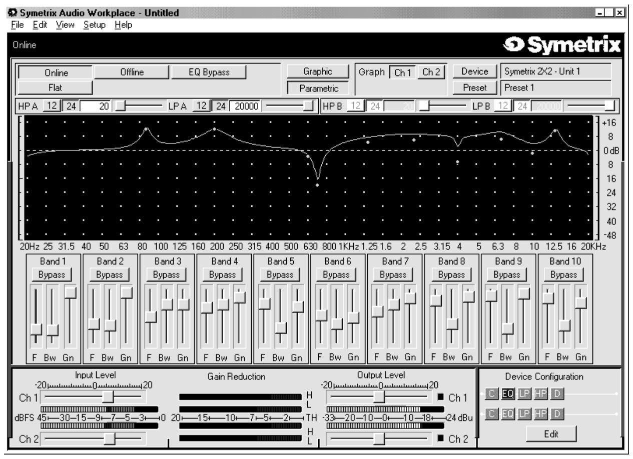
The Symetrix 9022 DSP Engine has the following functional modules built in to the system:

- Pink Noise Generator
- Low Pass Filter
- High Pass Filter
- Parametric or Graphic Equalizer
- One or Two Band Dynamics Processing
- Delay
- Settings Independent for Each Channel

The settings for each processing block are handled via the ***Symetrix Audio Workplace***® software, which is downloadable from www.symetrixaudio.com. This software allows the user to choose which processing blocks they want to use based upon the needs of their particular system. The user has the ability to put these blocks in the desired order in the signal chain. For instance, if a user wants the dynamics processor before the graphic EQ but after a low pass filter, they simply put the blocks in that order.

Each Block has a button labeled "Master." If that button is set, then the opposite channel's setting will follow the master channel. This is very useful in stereo applications to ensure that both channels get the same processing. Otherwise, both channels need to be adjusted individually. This would likely be the case in a dual-mono system. Once the blocks are set in the desired order, the user can double click the desired block for editing, which brings up a full screen graphical editor for that function.

The equalizer type can be set for graphic or parametric. A parametric EQ screenshot is shown in Figure 6. The parametric is useful for notching out particular frequencies such as AC hum.

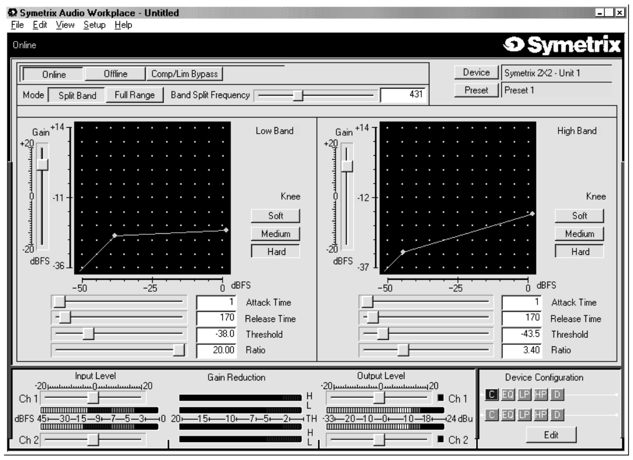


Device: Symetrix 2X2 - Unit 1
Preset: Preset 1
HP A: 12 | 24, 20
LP A: 12 | 24, 20000

Figure 6. Screenshot of Symmetrix *Audio Workplace* Parametric Equalizer Function.

The graphic equalizer is useful for adjusting the equalizer for the room acoustics. Symetrix included a built in pink-noise generator to aid in this setup.

The dynamics processor can be wide band or dual band. The user can graphically adjust the threshold and ratio. Sliders are used to control the attack and release times. A screenshot of the dynamics processor is displayed in Figure 7.



Device: Symetrix 2X2 - Unit 1
Preset: Preset 1
Band Split Frequency: 431

Low Band: Attack Time 1, Release Time 170, Threshold -38.0, Ratio 20.00

High Band: Attack Time 1, Release Time 170, Threshold -43.5, Ratio 3.40

Figure 7. Screenshot of Symmetrix *Audio Workplace* Compressor Function.

While editing each stage, live program material can be monitored with real-time meters, keeping the user abreast of the I/O levels as settings are adjusted. This is especially useful within the equalizer section, where too much boost could drive the audio into digital clipping.

Another powerful feature of the 9022 is up to eight different presets containing entirely different processing structures that can be programmed into the unit. The setting is recalled by a contact closure to one of the eight selector inputs in the rear of the unit. This means eight completely different processing structures can be programmed into one unit and selected without the need for an attached computer.

IVIE 626 with Audionet control

The IVIE 626 has the following functional abilities built in to the system:

- 2 X 6 Signal Router
- Pre and Post EQ Feeds
- Parametric EQ
- Compression/Limiting
- Delay
- Pink Noise Generator
- 2 and/or 3 Way Crossovers

Many of the functions are similar to the Symetrix 9022. In addition, the IVIE has six audio outputs, all of which can be fed either a mono source or left and right channels. This is useful for installed systems where parts of a facility may require stereo processing but other parts require a feed from mono sum.

Using the ***Audionet+***® software, downloadable from www.ivie.com, up to 16 different processing structures can be stored in the system. Settings are recallable by a closure to one of the selector inputs on the rear of the unit. These processing structures contain routing information, meaning the unit can act as a 2x6 router as well as changing the equalization and dynamics of the entire unit.

The processing structure of the IVIE 626 can be observed from the screen shot in Figure 8. The two input signals can have individual twelve-band parametric equalization, dynamics, and delay added to the signal. Unlike the 9022, the order of these functions cannot change.

Each of the six outputs can, however, have their own processing added. The source for each output is selected by choosing the audio buss it will be assigned. This can be raw left or right, equalized and dynamics processed sum of left and right, or equalized, dynamics processed, and delayed left or right. After the source is selected, the output can be delayed further with a three-band equalizer and dynamics added before the final output. The unit also contains a pink noise generator useful for installation equalization set up.

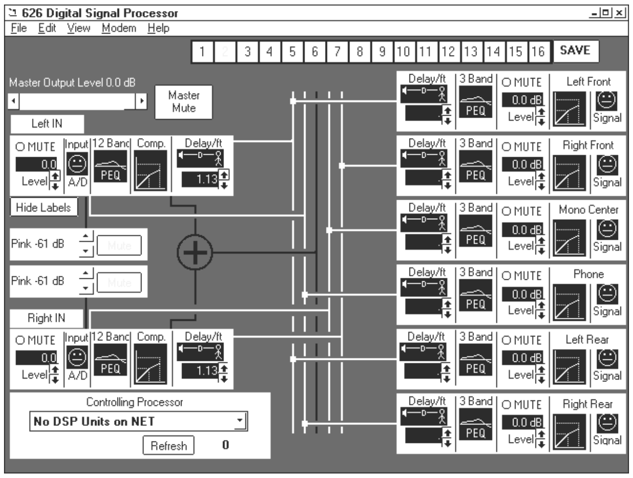


Figure 8. Screenshot of IVIE *Audionet+* 626 Digital Signal Processor Window.

The versatility of this unit lies in the fact that each of the six outputs can have individual processing and source assignments. While the design lends itself primarily to the installed market, there are potential uses in the broadcast market such as feeding a slightly different processed signal to an FM transmitter, AM transmitter, music on hold, and webcast.

Rane RPM 26v and RPM 26i

The Rane RPM26v has the following functional abilities built in to the system:

- Two and Six-Band Parametric EQs
- Compressor
- Limiter on Every Output
- Delay
- Pink Noise Generator
- 14 User-Selectable Processing Structures
- AES/EBU input on RPM 26i

Programming is performed using ***Raneware***® software downloadable from www.rane.com. To set up the unit, the user first selects the processing structure needed for the application. A different processing structure can be programmed for every preset. Most of the processing structures are designed for crossover applications with some of the others for routing. Some of the structures are similar to the structures seen in the IVIE 626 and Symetrix 9022, containing equalization, compression, and delay for the inputs, selectable output routing, individual output channel equalization, and individual delay and limiting for each output.

The units each have a stereo or dual mono input, and six outputs. The user can select one of 14 processing and routing architectures before editing the parameters on the units various equalizers and dynamics processors. One such structure can be observed in Figure 9. Sixteen user programs can be stored with the setup software and recalled using contact closures to the memory recall port located on the rear of the unit.

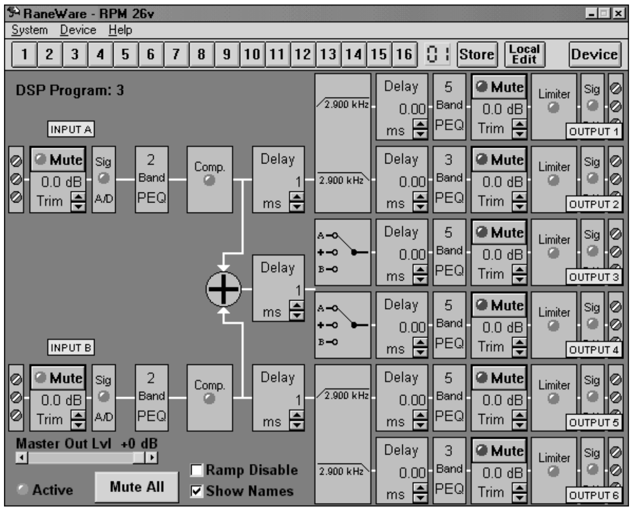


Figure 9. Screenshot of Rane *RaneWare* RPM26v Dual Two-Way Crossover Window.

The equalizer screen allows the user to graphically adjust the equalization frequencies, gain and bandwidth. The display shows the response for each filter.

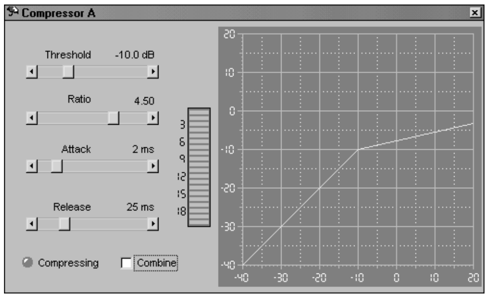


Figure 10, Screenshot of Rane RaneWare RPM26V Dynamics Processing Window.

The compressor window, shown in Figure 10, has a graphical representation of the threshold and ratio. The attack and release are adjustable via a slider. The limiter screen is similar to the compressor except that the ratio is not selectable.

Manufacturer's Development Boards

So far, this paper has addressed complete system level solutions that ship with commercial software and are extremely simple to use. These have great advantages to the end user, but do nothing to educate broadcast professionals about the inner workings of a digital signal processing system.

DSP system designers test their algorithms, before actually building them, using manufacturer's development boards. These boards typically have an onboard DSP processor, a generic codec capable of various sample rates, access to other ports on the DSP processor, ROM chip containing a programming interface to a PC, and a UART for direct serial connection to a PC. A typical development board block diagram is shown in Figure 11.

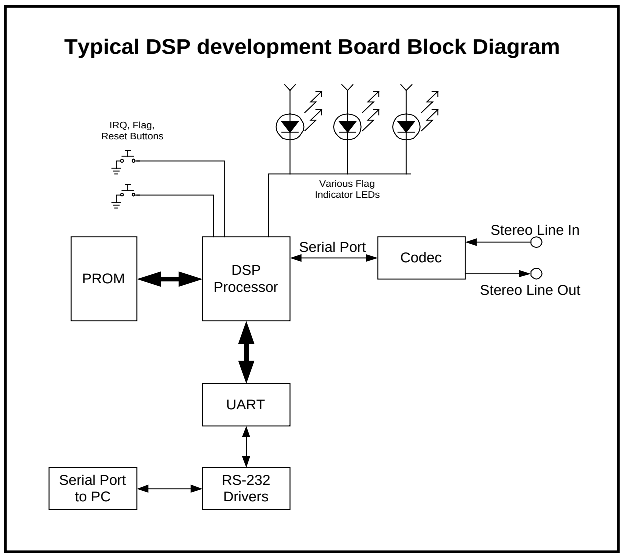


Figure 11. Typical DSP Development Board Block Diagram.

Boards usually ship with a selection of software tools used to develop applications for the DSP processor. Some of these tools are fully professional development packages while others are scaled down tools that may only work with a particular evaluation board.

Evaluation boards typically come in several varieties ranging from $100 starter kits to $3000 complete developer kits. While the more expensive systems come with more capable hardware, typically the major difference in these systems is the capabilities of the included software development tools. Texas Instruments calls their low-end development boards "Starter Kits" while Analog Devices calls theirs "EZKIT-LITE." In any case, the purchaser of these boards should research what they are getting.

Texas Instruments

Texas Instruments has several development boards to choose from depending on application. The most common is the C3X DSP Starter Kit. It features a TMS320C31 floating-point DSP processor running at 25MHz. An on-board variable-rate codec interfaces audio signals from the outside world to the processor. A Parallel port interface connects the board to a user's PC.

For a more powerful DSP kick, the TMDX320005402 DSP starter kit features a 100MHz TMS320C54002 fixed-point DSP processor. The board has standard 3.5mm audio jacks for inputs and outputs from the variable-rate codec.

Both units contain only mono codecs and come with a C compiler, an Assembler, and a Linker.

Motorola

Motorola's Evaluation Module Kit for the DSP56303 contains a 24-bit fixed-point processor running at 80 MHz or 100 MHz, depending on which kit is selected. The board contains a 16-bit stereo codec connected to stereo 1/8" phone jacks.

The unit ships with a windows based development system for writing, compiling, and assembling DSP programs. The program also serves as the program loader to the board. The board connects to the user's PC via an RS-232 serial cable.

The board is available with other 563XX processors depending on the users memory and speed requirements.

Analog Devices

The Analog Devices Sharc® line of DSP processors are 32-bit floating-point processors with varying speeds and features. There are two main development boards that are reasonably priced.

The ADDS-2106X-EZLITE contains a 32-bit floating point ADSP-21061 40MHz DSP processor. A variable-rate stereo codec interfaces the DSP to the analog world. The system will allow line or microphone level inputs with line level outputs. Software development tools as well as a program loader are included. The program loader allows compiled programs to be loaded to the DSP via a serial port from a PC and tested. A C language compiler, assembler and linker are all included as part of the software tools package. Compiled and ready to run sample programs are included as well as the source code for each.

Another Sharc® digital signal processor board option is the ADDS-21065-EZLITE development board. This board contains a 32-bit floating point ADSP-21065 60MHz DSP processor. A variable-rate stereo codec interfaces the DSP to the analog world. The system will allow line level inputs with line level outputs. The main difference between this board and the ADDS-2106x board is the processor. The ADDS-21065L board also comes with a newer AD1819A full-duplex 16-bit codec. The ADDS-21065L board is currently shipping with the newer *VisualDSP++* design tools from Analog Devices. As of this writing, *VisualDSP++* does not support the ADDS-2106x development board, however this should be available in 2001.

A PRACTICAL DESIGN EXAMPLE USING ANALOG DEVICES ADDS-2106x-EZLITE

The only way to truly appreciate the simplicity of DSP algorithm design is to see it in action. This example should demonstrate all of the steps necessary to create a viable algorithm. In order to demonstrate the fact that it does not take a scientist to program a DSP, this example will attempt to use as much source code as possible from other sources. While the final program will be specific to the ADDS-2106x-EZLITE development board, the example will be programmed in C language, making it more portable to another platform.

The Problem

The problem chosen for this example is something that might be used in a real-world broadcast engineering environment. The design problem is a 2-band compressor. Starting with the block diagram shown in Figure 12, it can be observed that this is a typical design, which might be found in commercial hardware. Of course this entire block diagram will be built with software not hardware.

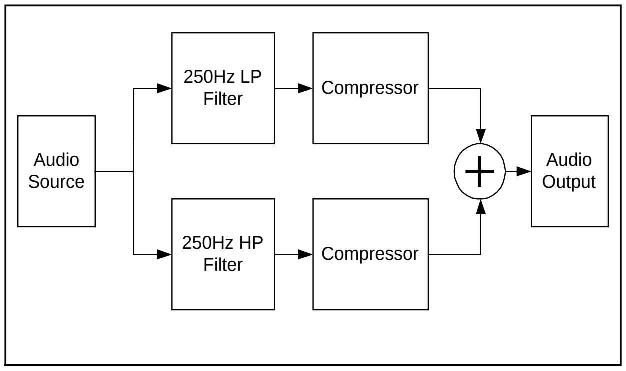


Figure 12. Two-Band Compressor Block Diagram.

The Beginning

The first step in the design process is to find a prewritten section of software code that will take care of the overhead functions of the development board. Overhead functions might include items such as setting up interrupt vectors, initializing the codec, initializing Direct Memory Access (DMA), and other functions not related to the algorithm being developed. In this case, the code segment came with the development board. A program called "tt" came with the development board. Its purpose: Take an input sample and return the sample to the output port. This seems like a useless program, but it is a teaching program used to demonstrate how to change sample rates. Because the program is simple and only needs minor modification (to remove the sample rate change ability) it was chosen as the victim of this experiment. The source code for *tt* is included with the development board. The source code is available in C language or assembly. A look through the directory yielded a source code file named "tt.c."

The key part of the segment to look for is this:

```
void spr0_asserted( int sig_num )
{
          // Copy received data buffers
//to transmit data buffers.
          tx_buf[1] = rx_buf[1];
          tx_buf[2] = rx_buf[2];
}
```

The line "void spr0_asserted…" is an interrupt vector for the serial port receive buffer. Whenever an audio sample is received, this small chunk of code gets executed. Otherwise, the processor typically sets idle (depending on the program). Notice the "tx_buf[1] = rx_buf[1];" line. This line simply takes whatever came in the receive buffer and puts it in the transmit buffer which will be sent to the codec when next transmit interrupt in received. Simply stated, rx_buf[1] is the left input sample, rx_buf[2] is the right input sample, while tx_buf[1] is the left output sample and tx_buf[2] is the right output sample. The code created in this experiment will replace these two lines of the sample code. The starter code will also be modified to sample the audio at 48K. The data sheet for the onboard codec is available online in PDF format form www.analog.com, but a look at the source code for the program *tt.c*, was of greater assistance in determining what settings would yield a 48K sample rate.

The Hardware and Requirements

Taking a look at the abilities of the hardware for a moment, it is observed that the speed of the development board is 40MHz. The number of operations that can be performed per sample needs to be calculated. This is to determine how many operations can be performed on a sample before the processor receives another sample. At 48KHz, the period between the samples can be determined using the calculation from earlier in the paper:

$$T = \frac{1}{Fs} = \frac{1}{48KHz} = 20.83\mu S / Sample$$

With a processor speed of 40,000,000 instructions per second, the following equation cancels out the seconds and yields instructions per sample.

$$\frac{Instruct}{Sample} = 40{,}000{,}000 \frac{Instruct}{S} \times 20.83 \frac{\mu S}{Sample}$$

$$\frac{Instruct}{Sample} = 833$$

This means 833 operations can be performed on a sample before the next one arrives and expects to be processed. This is important since too many operations on a sample would leave too little time to perform operations on the next sample. Eventually, samples will get dropped because they could not get serviced in time. This activity creates an extremely undesirable form of distortion and it is recommended that the programmer take care not to let this happen.

Assuming that 833 operations can be performed on each sample is an optimistic viewpoint. There are many instructions used for overhead in the processor. A more likely 800 instructions per sample is a nice round number, which leaves a little headroom for the processor to handle other potential interrupt requests.

Programming in C has its own disadvantages because the final compiled code is not as efficient as compiled assembly language. Most DSP programmers program in assembly language. While the Analog Devices assembly language is easy to use, it was still thought best to use a common language in this example. That still leaves the question of how many instructions may be executed between each sample. The answer is still 800, but using C language means that those 800 instructions may only perform what might be done in 600 assembly instructions. For this example, however, the count should be well below either.

The Software Tools

The Analog Devices board ships with a command line C compiler. Some of the newer boards are shipping with a graphical Integrated Development Environment (IDE). Either of these will work for this example problem. Both will require the programmer to compile the final code and then link it to its libraries to create an executable for upload to the development board. The documentation for these tools will need to be consulted for operational instructions since that is beyond the scope of this paper.

The Design Process

The skeleton code has been obtained and slightly modified, the software tools are in place, and the design problem is set. It is time to get to the business at hand.

Filters

The first thing needing attention is the design of the filters. An FIR filter will fit the design since it is completely phase linear, can have excellent cutoff, and is extremely stable. A CAD program will be used to design the filter. CAD programs usually have several FIR filter design options available based on different methods of filter approximation. The Parks-McClellan equiripple approximation is an iterative algorithm and will yield the fewest number of taps for any given filter tolerance mask. The fewer the taps, the less delay, and the fewer instructions used in the processor.

To begin the design of the filter, specify a filter mask. A low-pass filter mask with a cutoff of 250Hz, and 12dB/oct cutoff could be drawn as demonstrated in Figure 13.

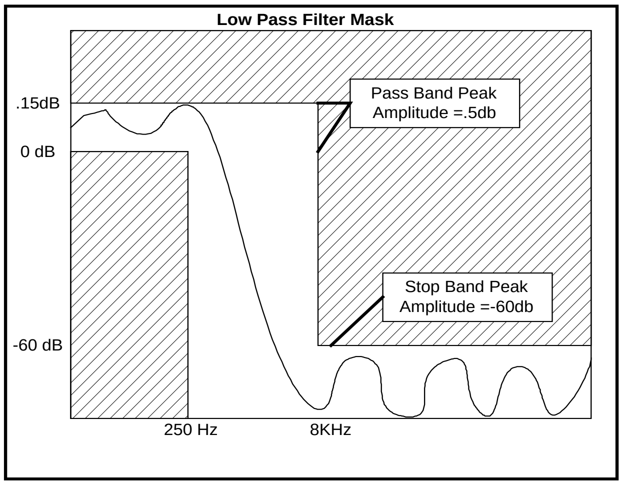


Figure 13. 250Hz Low Pass Filter Mask.

Next, the number of required taps needs to be determined using a formula. There are very complex equations that exist for determining the number of taps in a given equiripple filter. Fortunately, there are some simple formulas that will get an approximate answer. An estimate is usually sufficient, unless a programmer is truly short on instruction cycles. A simple equation for determining the required number of taps in a equiripple filter is as follows:

$$N \cong \frac{-20\log 10(\sqrt{\delta_p \delta_s}) - 13}{2.324\Delta\omega} + 1$$

Where:

N=Number of Taps

$\Delta\omega$=normalized transition bandwidth

δ_p=pass band peak ripple amplitude

δ_s=stop band peak ripple amplitude

For the given mask, these variables can be filled in as such:

$$\Delta\omega = \frac{2\pi\Delta f}{F_{sample}}$$

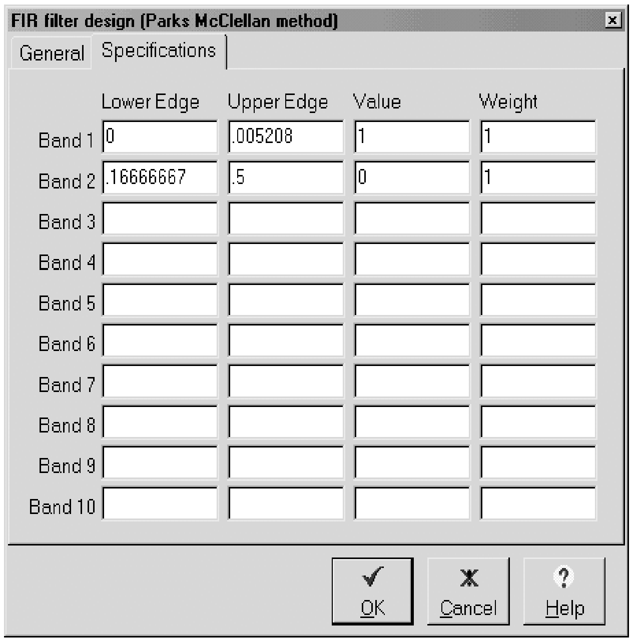
$$\delta_p = 10^{\frac{.15}{20}} = .0174 \qquad \delta_s = 10^{\frac{-60}{20}} = .001$$

A spreadsheet will come in handy here. By inserting the numbers in the appropriate spots, N turns out to be 14.67, which should be rounded up to 15. This is the number of taps required for the Parks-McClellan equiripple approximation that will be performed using a CAD design program.

CAD Design

Using a CAD program makes FIR filter design a cinch. There are many products to choose from including Matlab, PC-DSP and even some shareware programs available on the net. For this simple design problem, the free demo of PC-DSP will be sufficient. The demo version of PC-DSP is available from www.dspsolutions.com.

Selecting the Parks-McClellan approximation from the FIR filter design menu brings the window in Figure 14. Tap information and a name for the filter is entered under the tab labeled "General." The "Specifications" tab is where the information for the frequency response of the filter is entered.



	Lower Edge	Upper Edge	Value	Weight
Band 1	0	.005208	1	1
Band 2	.16666667	.5	0	1
Band 3				
Band 4				
Band 5				
Band 6				
Band 7				
Band 8				
Band 9				
Band 10				

Figure 14. PC-DSP Parks-McClellan FIR Filter Design Window.

The information is entered in the boxes based on the desired shape of the filter. As can be seen above, the part of the filter having "Value" set to 1 is the pass band, and has a lower edge of 0 (0Hz) and an upper edge of .005208. The .005208 is the result of dividing the frequency by the sample rate. This is called "normalized frequency." The normalized frequency is used for all DSP design software and allows the same design algorithm to work for any sample rate. The stop band has the "Value" set to 0 and the frequency ranges from .16666667 (8KHz) to .5 (24KHz). The frequencies above .5 are invalid since this is the maximum frequency any digital system can sample without aliasing. Clicking OK sets the design into action.

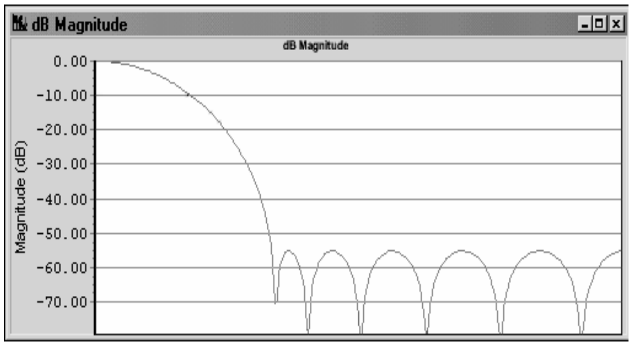


Figure 15. FIR 250 Hz Low Pass Filter Design dB Magnitude Plot.

The analysis plot of the magnitude in dB of the designed filter is shown in Figure 15. Note that the stop band gain is actually about –55dB, not the specified -60dB. This most likely happened due to an insufficient number of taps in the filter. Remember, the design equation used to determine the number of taps was an approximation. However, -55dB is close enough for the purposes of this experiment, so the filter will not be redesigned.

The actual tap coefficients, when plotted, are shown in Figure 16. This is also known as the impulse response of the filter. This filter happens to have all positive coefficients. Typically filter impulse responses will have both positive and negative coefficients.

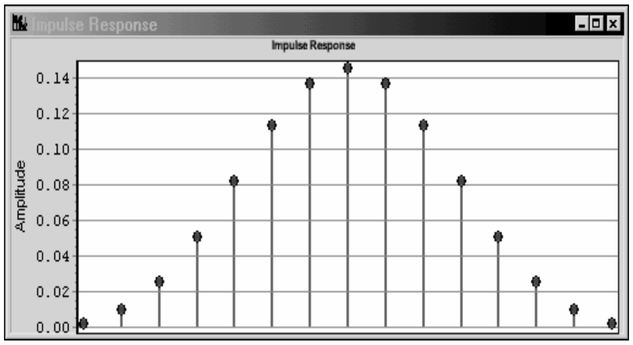


Figure 16. FIR Low Pass Filter Design Impulse Response Plot.

The linear phase so highly praised by audio purists can be seen in Figure 17.

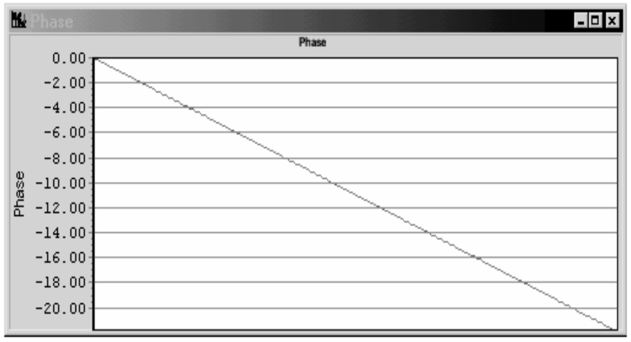


Figure 17. FIR Filter Phase Response.

This is half of the filter work. Since the crossover points in the two-band splitter are at 250Hz, the same design factors can be used. Simply renaming the filter and changing the "Values" in the Parks-McClellan window of PC-DSP will yield a high-pass filter with the same roll-off characteristics and crossover points.

The response of the high-pass filter, overlaid on the response of the low-pass filter is shown in Figure 18.

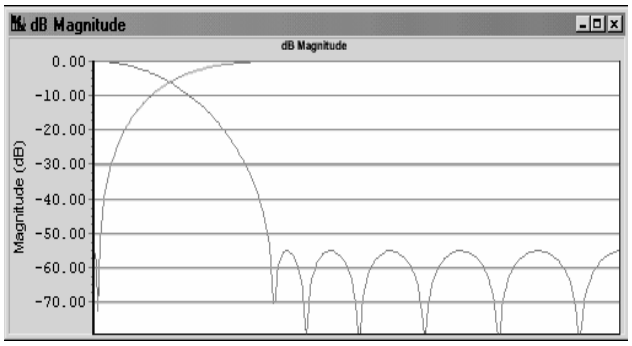
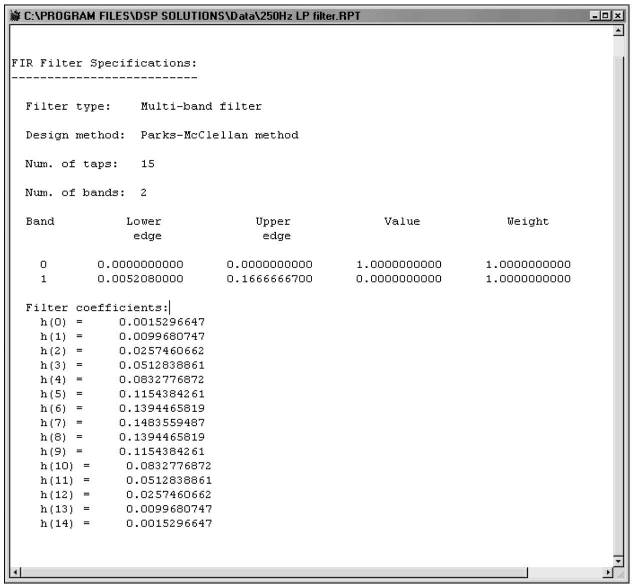


Figure 18. FIR High Pass and Low Pass dB Magnitude Plots.

Now that the two-way crossover has been designed, it is time to write the software code.

The Implementation

To begin, the coefficients are needed. These can be cut and pasted from the demo version of PC-DSP, output to a file by the full version of PC-DSP, or output to a file by some other CAD program. The printout from PC-DSP is shown in Figure 19.



C:\PROGRAM FILES\DSP SOLUTIONS\Data\250Hz LP filter.RPT

```
FIR Filter Specifications:
--------------------------

  Filter type:    Multi-band filter

  Design method:  Parks-McClellan method

  Num. of taps:   15

  Num. of bands:  2

  Band          Lower            Upper            Value            Weight
                 edge             edge

    0       0.0000000000     0.0000000000     1.0000000000     1.0000000000
    1       0.0052080000     0.1666666700     0.0000000000     1.0000000000

  Filter coefficients:
    h(0) =      0.0015296647
    h(1) =      0.0099680747
    h(2) =      0.0257460662
    h(3) =      0.0512838861
    h(4) =      0.0832776872
    h(5) =      0.1154384261
    h(6) =      0.1394465819
    h(7) =      0.1483559487
    h(8) =      0.1394465819
    h(9) =      0.1154384261
    h(10) =      0.0832776872
    h(11) =      0.0512838861
    h(12) =      0.0257460662
    h(13) =      0.0099680747
    h(14) =      0.0015296647
```

Figure 19. FIR Low Pass Filter Report From PC-DSP 2.0 Demo Version.

The filter coefficients were cut and pasted to a text editor and edited to end up with this C language code fragment as the final product:

```
float pm LP_Coeffs[TAPS]=
       {
       0.0015296647,
       0.0099680747,
       0.0257460662,
       0.0512838861,
       0.0832776872,
       0.1154384261,
       0.1394465819,
       0.1483559487,
       0.1394465819,
       0.1154384261,
       0.0832776872,
       0.0512838861,
       0.0257460662,
       0.0099680747,
       0.0015296647
       };
```

This is an array variable declaration and assignment that the DSP program will use to implement the FIR filter. This code fragment will be inserted into the modified Analog Devices code. Another two lines of code are needed to complete the required overhead for a C language implementation of an FIR filter. These are:

```
float dm LP_state1[TAPS+1];
float dm LP_state2[TAPS+1];
```

These two lines set up two separate buffers to be used by the FIR filter function as storage during FIR calculations. It is generally one memory location longer than the number of taps being implemented. Since a particular variable may be stored in this buffer for more than one sample, two are required to keep the stereo signals completely separated. The same process will be repeated in the coding of the high pass filter. A variable will be defined using the coefficients generated from a high pass filter report, and then two more buffers will be set up.

The implementation of the filters in C language will take 4 lines of code. These are:

```
LP_Out_Left=fir((float)Left,&LP_Coeffs[0],&LP_state1[0],(int)TAPS);
LP_Out_Right=fir((float)Right,&LP_Coeffs[0],&LP_state2[0],(int)TAPS);
HP_Out_Left=fir((float)Left,&HP_Coeffs[0],&HP_state1[0],(int)TAPS);
HP_Out_Right=fir((float)Right,&HP_Coeffs[0],&HP_state2[0],(int)TAPS);
```

These four lines implement the FIR filter function 4 times to get a high pass and low pass filter for both the left and right channels. The *fir()* function is a special function included in the analog Devices C language libraries. Refer to the complete code listing at the end of this paper to see how this code is inserted into the program. Notice how the design took much longer than the coding. Building upon programs that have already been written is the basis for almost all programming today. There is a lot of freely available source code from manufacturers and users in the Internet.

Now that the filters are designed and coded, the compressor needs to be designed. An applications document written by John Tomarakos and Dan Ledger was found to be extremely helpful for this part of the design. The publication describes many audio applications including compression, limiting, gating and expanding. The important aspect of the document is that it contains source code for the applications. In addition, the source code can be downloaded from the Internet.

However, the source code to accompany the document is written in assembly language. This is a problem because it was decided use C language to write the code for the example. Luckily, the algorithm is not very complex. Further examination of the assembly code, and the accompanying notes within the code, made it clear how the algorithm worked. The algorithm compares the input sample to a threshold level to determine if any processing action needs to be taken. If so, the threshold is subtracted from the signal and the remaining part is multiplied by the gain reduction ratio. The threshold is then added back to the sample and it is ready for output. The code fragment for one channel of compression in assembly is as follows:

```
compress_left:
   f4 = abs f2;
   comp(f4,f1); /* Is left channel past threshold? */
   if LT jump compress_right; /* If not, check right channel */
    f4 = f4 - f1; /* signal = signal - threshold */
    f4 = f4 * f0; /* signal = signal * ratio */
    f4 = f4 + f1; /* signal = signal + threshold */
    f2 = f4 copysign f2; /* f2 now=compressed left channel*/
```

The author of this code was gracious enough to document it thoroughly. From this documentation, the C language implementation can be written using what might be just as many lines of code. As can be seen, the algorithm requires an input signal, a threshold and gain reduction variables. The input will be the output of the filters designed earlier. What need to be determined and programmed are the threshold and gain reduction levels. The complete example by Tomarakos allowed the user to push a button on the development board to toggle the compression ratio through several levels. That is a little beyond this example's scope so a constant threshold and ratio will be used. Since this was not covered in the design spec for this example, an arbitrary gain reduction will be chosen and set at 5:1. The threshold will be set at .5 (1 being the maximum level before digital clipping). For simplicity, there will be no provisions for attack time and release time.

These will be set as program constants similarly to the way "TAPS" was set in the filter section. The following code will be added at the beginning of the program where the constants are defined:

```
#define RATIO .2
#define THRESH .5
```

If the programmer wants to add variable ratio and threshold in the future, the programmer need only remove the *#define* statements and use the names as variables within the program. This is much simpler than hard coding the ratio and threshold into every line of code that uses them.

Now the C language compressor algorithm can be written as follows:

```
if (abs(LP_Out_Left)>THRESH)
        {
        LP_Out_Left= LP_Out_Left-THRESH;
        LP_Out_Left= LP_Out_Left*RATIO;
        Bass_Comp_Left= LP_Out_Left+THRESH;
        }
```

The compressed low pass left channel is now located in the variable called "Bass_Comp_Left." The code needs to be implemented three more times to cover all the bands for both channels. After the compression has been run in all channels, the results need to be summed properly and sent to the codec for output. The following lines do just that:

```
Left_Mix_Out=Bass_Comp_Left+Hi_Comp_Left;
Right_Mix_Out=Bass_Comp_Right+Hi_Comp_Right;
```

The last required code writing step is to send the data to the transmit buffer for it to be sent to the codec at the next sample cycle. The following lines accomplish this:

```
tx_buf[1]=Left_Mix_Out;
tx_buf[2]=Right_Mix_Out;
```

The intermediate variable named Left_Mix_Out could have been omitted and the transmit buffer would be set directly to the sum of the compressed outputs. While more efficient, it did not clearly illustrate what was happening. There are many examples of this in the code just written. While it works, keep in mind there are more efficient methods of programming. The code is ready to be compiled. The complete code listing for this example is Code Listing 1 at the end of this paper.

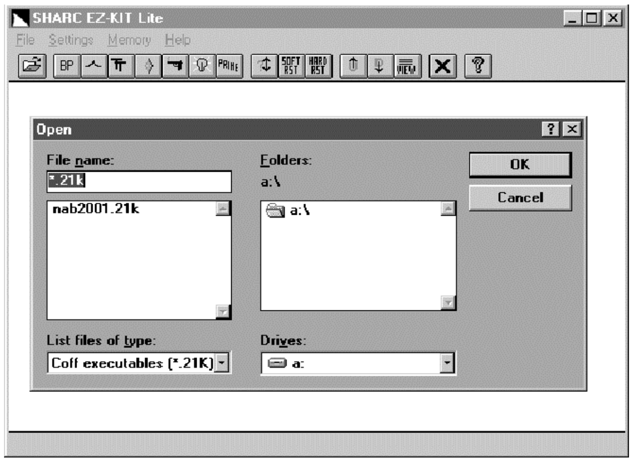


Figure 20. Screenshot of Analog Devices Sharc EZ-KIT Lite Host Interface Software.

Once the code is compiled and linked, the executable can be uploaded to the DSP development board for testing. In the case of the ADDS-2106x-EZLITE, the program is uploaded using the EZ-KIT LITE Host program. This is a windows based program that makes uploading the program extremely simple. The basic screen is shown in Figure 20.

Click the open folder to bring up the file window and select the file that was compiled and linked earlier. In this case the file has a ".21K" extension denoting the fact that it is an executable for a Shark 21000 series DSP processor. Once the file is selected and OK is clicked, the file begins to upload to the development board immediately. As soon as it is uploaded, the program begins to execute. At this point, the program is running, and audio inputs and outputs can be connected to the board to determine if all this effort has paid off.

Subjective listening tests show that this algorithm does indeed work. However, since there were no provisions for attack and decay, the algorithm will distort the audio soon after the audio threshold is reached. Since the algorithm acts on every sample, the attack and decay times are effectively instantaneous. However, this made for an excellent example to demonstrate how some knowledge of filter design and code reuse can make DSP design less of a mystery than it has previously been.

Extras for an Installable Device

For those bold enough to take the project to the next level, here are a few suggestions. First, modify the code for efficiency by doing away with the intermediate variables used for demonstrative purposes.

Second, Add variable attack and decay by using a pushbutton interface or other means of interfacing human control to the DSP program. Some designers have actually used analog potentiometers and sampled the values as a means of control. Today's common DSP based broadcast devices use a general-purpose microcontroller and various menu functions to interface the human to the DSP algorithm.

Third, package the project in some sort of rack mount case. A balanced to unbalanced interface should be a part of this. A well laid out user interface panel would also make the project more appealing.

Do this, and the project might end up being the next Optimod$_{®}$ or Omnia$_{®}$. Who knows?

SUMMARY

Digital signal processing has long been a mystery to the broadcast engineer. Broadcast engineers have had to put their faith in the processing manufacturers, now that the days of twiddle sticks, greenie screwdrivers, soldering irons and a handful of filter capacitors have given way to rolly wheels, LCD displays, and menu functions.

Knowledge is power, especially knowledge of how DSP systems operate. Books on the subject are filled with equations and transforms, but seldom touch on the subject of implementing a system with hardware. The best way for an engineer to learn about DSP is to get their hands dirty by experimenting with self-contained DSP systems or writing simple DSP programs for implementation on development boards. Hands-on DSP is achievable without an advanced degree in calculus. The broadcast engineer need only be resourceful enough, and bold enough, to jump in feet first and experiment.

CODE LISTINGS

Code Listing 1. Complete code for the design project modified from tt.c program included with the Analog Devices ADDS-2106x-EZLITE development board. It includes custom code and code adapted from freely available assembly code from Analog Devices. Code developed in the experiment described in the paper is denoted in bold.

```
/*************************************************************/
/* 2-Band Compressor for Demonstration of filter
/* and compressor design during NAB2001 BEC--April 2001
/* Designed by Michael Gay
/* http://icdweb.cc.purdue.edu/~gay
/* No Warranties are expressed or implied regarding the
/* suitability of this code for any application.
/*************************************************************/

/* ADSP-2106x System Register bit definitions */
#include <def21060.h>
#include <21060.h>
#include <signal.h>
#include <sport.h>
#include <macros.h>
#include <math.h>
#include <filters.h>
#include <stdio.h>
                                     /* DMA Chain pointer bit definitions */
#define CP_PCI 0x20000          /* Program-Controlled Interrupts bit */
#define CP_MAF 0x1ffff          /* Valid memory address field bits   */

#define SetIOP(addr, val)  (* (int *) addr) = (val)
#define GetIOP(addr)       (* (int *) addr)

#define TAPS 15  /*This is where we set the number of taps we will be using*/
#define RATIO .05 /*This is where we set the compression ratio*/
#define THRESH .25 /*This is where we seth the compression threshold*/

/*************************************************************/
/*             Set some Global Variables                      */
/*Not all of these are required, but they should make the */
/*code easier to follow for the Novice                      */
/*************************************************************/

float Left;           /* Left input from buffer */
float Right;          /* Right input from buffer */
float LP_Out_Left;    /* Left low Pass Output */
float LP_Out_Right;   /* Right low pass output */
float HP_Out_Left;    /* Left high pass output */
float HP_Out_Right;   /* Right high pass output */
float Bass_Comp_Left;/* Left bass compressor output */
float Bass_Comp_Right;/* Right bass compressor output */
float Hi_Comp_Left;  /* Left high band compressor output */
float Hi_Comp_Right; /* Right high band compressor output */
float Left_Mix_Out;  /* Left compressed high and bass mix  */
float Right_Mix_Out; /* Right compressed high and bass mix */

/*************************************************************/
/*Crossover Filter Variables Begin Here                    */
/*************************************************************/

/* The coefficients are set in memory below.  The float means set the data as */
/* floating point.  PM means put the data in program memory instead of data   */
/* memory (for dual data buss Harvard architecture). The LP_Coeffs[TAPS] is a */
/* one dimensional array with the length set by  the defined constant "TAPS". */
/* The same operation will be performed for the HP_Coeffs filter              */

float pm LP_Coeffs[TAPS]=
    {
    0.0015296647,
    0.0099680747,
    0.0257460662,
    0.0512838861,
    0.0832776872,
    0.1154384261,
    0.1394465819,
    0.1483559487,
    0.1394465819,
    0.1154384261,
    0.0832776872,
    0.0512838861,
    0.0257460662,
    0.0099680747,
    0.0015296647,
       };
/* The LP_state1 and LP_state2 are temporary buffers used by the C language */
/* implementation of an FIR filter.  Two are needed for each filter for     */
/* stereo operation. Notice these are stored in DM or data memory.          */

float dm LP_state1[TAPS+1];
float dm LP_state2[TAPS+1];

/*Repeat Everything for the High Pass filter      */
float pm HP_Coeffs[TAPS]=
    {
    -0.0015296647,
    -0.0099680747,
    -0.0257460662,
    -0.0512838861,
    -0.0832776872,
    -0.1154384261,
    -0.1394465819,
    0.8516440513,
    -0.1394465819,
    -0.1154384261,
    -0.0832776872,
    -0.0512838861,
    -0.0257460662,
    -0.0099680747,
    -0.0015296647
       };

float dm HP_state1[TAPS+1];
float dm HP_state2[TAPS+1];

/***************************************************************/
#define SZ_regs_1847 16
int regs_1847[SZ_regs_1847] = {
     /* Note that the MCE bit is maintained throughout initial
        programming to hold off premature autocalibration. */
     0xc000,                /* index 0 - left input control */
     0xc100,                /* index 1 - right input control */
     0xc280,                /* index 2 - left aux 1 input control */
     0xc380,                /* index 3 - right aux 1 input control */
     0xc480,                /* index 4 - left aux 2 input control */
     0xc580,                /* index 5 - right aux 2 input control */
     0xc600,                /* index 6 - left dac control */
     0xc700,                /* index 7 - right dac control */
      0xc85c,                /* index 8 - data format & Sample Rate*/
     0xc909,                /* index 9 - interface configuration */
     0xca00,                /* index 10 - pin control */
     0xcb00,                /* index 11 - no register */
     0xcc40,                /* index 12 - miscellaneous information */
     0xcd00,                /* index 13 - digital mix control */
     0xce00,                /* index 14 - no register */
     0x8f00};               /* index 15 - no register */

int rx_buf[3];                          /* receive buffer */
int tx_buf[3] = {0xcc40, 0, 0};         /* transmit buffer */

/* DMA chaining Transfer Control Blocks */
typedef struct {
   unsigned   lpath3;     /* for mesh multiprocessing  */
   unsigned   lpath2;     /* for mesh multiprocessing  */
   unsigned   lpath1;     /* for mesh multiprocessing  */
   unsigned   db;         /* General purpose register  */
   unsigned   gp;         /* General purpose register  */
   unsigned** cp;         /* Chain Pointer to next TCB */
   unsigned   c;          /* Count register            */
   int        im;         /* Index modifier register   */
   unsigned * ii;         /* Index register            */
} _tcb;

_tcb rx_tcb = {0, 0, 0, 0, 0, 0, 3, 1, 0};      /* receive tcb */
_tcb tx_tcb = {0, 0, 0, 0, 0, 0, 3, 1, 0};      /* transmit tcb */
int cmd_blk[8];                                 /* command block */
static int xmit_count;
static int * xmit_ptr;
static int source;
static int filter;
static int old_source;
static int old_filter;

/***************************************************************/
/*                                                             */
/* Periodic timer interrupt                                    */
/*                                                             */
/***************************************************************/
void timer_lo_prior( int sig_num )
{
    sig_num=sig_num;

    // Toggle flag 2 LED for no reason other than to see it flash
     // Actually it lets you know the processor has not locked up
    set_flag(SET_FLAG2, TGL_FLAG);
}
/***************************************************************/
/*                                                             */
/* Serial port transmit DMA complete                           */
/*                                                             */
/***************************************************************/
void spt0_asserted( int sig_num )
{
    // Check if there are more commands left to transmit.
    if( xmit_count )
    {
        // If so, put the command into the transmit buffer and update count.
        tx_buf[0] = *xmit_ptr++;
        xmit_count--;
    }
}
/************************************************************/
/*                                                          */
/* Serial port receive DMA complete                         */
/* Write New Code Here                                      */
/*                                                          */
/************************************************************/
void spr0_asserted( int sig_num )
{
     Left=rx_buf[1];
     Right=rx_buf[2];
     /* Implement Filters */
     LP_Out_Left= fir( (float)Left, &LP_Coeffs[0], &LP_state1[0], (int)TAPS );
     LP_Out_Right= fir( (float)Right, &LP_Coeffs[0], &LP_state2[0], (int)TAPS );
     HP_Out_Left= fir( (float)Left, &HP_Coeffs[0], &HP_state1[0], (int)TAPS );
     HP_Out_Right= fir( (float)Right, &HP_Coeffs[0], &HP_state2[0], (int)TAPS );
     /*Implimment Compressor*/
     if (abs(LP_Out_Left)>THRESH)
             {
             LP_Out_Left= LP_Out_Left-THRESH;
             LP_Out_Left= LP_Out_Left*RATIO;
             Bass_Comp_Left= LP_Out_Left+THRESH;
             }
     if (abs(LP_Out_Right)>THRESH)
             {
             LP_Out_Right= LP_Out_Right-THRESH;
             LP_Out_Right= LP_Out_Right*RATIO;
             Bass_Comp_Right= LP_Out_Right+THRESH;
             }
     if (abs(HP_Out_Left)>THRESH)
             {
             HP_Out_Left= HP_Out_Left-THRESH;
             HP_Out_Left= HP_Out_Left*RATIO;
             Hi_Comp_Left= HP_Out_Left+THRESH;
             }
     if (abs(HP_Out_Right)>THRESH)
             {
             HP_Out_Right= HP_Out_Right-THRESH;
             HP_Out_Right= HP_Out_Right*RATIO;
             Hi_Comp_Right= HP_Out_Right+THRESH;
             }
     /*Mix Outputs*/
     Left_Mix_Out=Bass_Comp_Left+Hi_Comp_Left;
     Right_Mix_Out=Bass_Comp_Right+Hi_Comp_Right;
     /*Send to Codec*/
     tx_buf[1]=Left_Mix_Out;
     tx_buf[2]=Right_Mix_Out;
}
/************************************************************/
/*                                                          */
/*   Some overhead stuff to setup the serial ports          */
/*   to operate.                                            */
/*                                                          */
/************************************************************/
void setup_sports ( void )
{
    /* Configure SHARC serial port SPORT0 */

    /* Multichannel communications setup */
    sport0_iop.mtcs  = 0x00070007;       /* transmit on words 0,1,2,16,17,18 */
    sport0_iop.mrcs  = 0x00070007;       /* receive on words 0,1,2,16,17,18  */
    sport0_iop.mtccs = 0x00000000;       /* no companding on transmit        */
    sport0_iop.mrccs = 0x00000000;       /* no companding on receive         */

    /* TRANSMIT CONTROL REGISTER */
    /* An alternate (and more efficient) way of doing this would be to   */
    /* write the 32-bit register all at once with a statement like this: */
    /*       SetIOP(STCTL0, 0x001c00f2);                                 */
    /* But the following is more descriptive...                          */

    sport0_iop.txc.mdf   = 1;    /* multichannel frame delay (MFD)       */
    sport0_iop.txc.schen = 1;    /* Tx DMA chaining enable               */
    sport0_iop.txc.sden  = 1;    /* Tx DMA enable                        */
    sport0_iop.txc.lafs  = 0;    /* Late TFS (alternate)                 */
    sport0_iop.txc.ltfs  = 0;    /* Active low TFS                       */
    sport0_iop.txc.ditfs = 0;    /* Data independent TFS                 */
    sport0_iop.txc.itfs  = 0;    /* Internally generated TFS             */
    sport0_iop.txc.tfsr  = 0;    /* TFS Required                         */

    sport0_iop.txc.ckre  = 0;    /* Data and FS on clock rising edge     */
    sport0_iop.txc.gclk  = 0;    /* Enable clock only during transmission*/
    sport0_iop.txc.iclk  = 0;    /* Internally generated Tx clock        */
    sport0_iop.txc.pack  = 0;    /* Unpack 32b words into two 16b tx's   */

    sport0_iop.txc.slen  = 15;   /* Data word length minus one           */
    sport0_iop.txc.sendn = 0;    /* Data word endian 1 = LSB first       */
    sport0_iop.txc.dtype = SPORT_DTYPE_RIGHT_JUSTIFY_SIGN_EXTEND;
                                 /* Data type specifier                  */
    sport0_iop.txc.spen  = 0;    /* Enable (clear for MC operation)      */

    /* RECEIVE CONTROL REGISTER */
    /* SRCTL0 <= 0x1f8c20f2     */
    sport0_iop.rxc.nch   = 31;   /* multichannel number of channels - 1  */
    sport0_iop.rxc.mce   = 1;    /* multichannel enable                  */
    sport0_iop.rxc.spl   = 0;    /* Loop back configure (test)           */
    sport0_iop.rxc.d2dma = 0;    /* Enable 2-dimensional DMA array       */
    sport0_iop.rxc.schen = 1;    /* Rx DMA chaining enable               */
    sport0_iop.rxc.sden  = 1;    /* Rx DMA enable                        */
    sport0_iop.rxc.lafs  = 0;     /* Late RFS (alternate)                 */
    sport0_iop.rxc.ltfs  = 0;     /* Active low RFS                       */
    sport0_iop.rxc.irfs  = 0;     /* Internally generated RFS             */
    sport0_iop.rxc.rfsr  = 1;     /* RFS Required                         */
    sport0_iop.rxc.ckre  = 0;     /* Data and FS on clock rising edge     */
    sport0_iop.rxc.gclk  = 0;     /* Enable clock only during transmission*/
    sport0_iop.rxc.iclk  = 0;     /* Internally generated Rx clock        */
    sport0_iop.rxc.pack  = 0;     /* Pack two 16b rx's into 32b word      */

    sport0_iop.rxc.slen  = 15;    /* Data word length minus one           */
    sport0_iop.rxc.sendn = 0;     /* Data word endian 1 = LSB first       */
    sport0_iop.rxc.dtype = SPORT_DTYPE_RIGHT_JUSTIFY_SIGN_EXTEND;
                                  /* Data type specifier                       */
    sport0_iop.rxc.spen = 0;      /* Enable (clear for MC operation)      */

    /* Enable sport0 xmit & rcv irqs (DMA enabled) */
    interrupt(SIG_SPR0I, spr0_asserted);
    interrupt(SIG_SPT0I, spt0_asserted);

    /* Set up Transmit Transfer Control Block for chained DMA */
    tx_tcb.ii = tx_buf;           /* DMA source buffer address              */
    tx_tcb.cp = &tx_tcb.ii;       /* define ptr to next TCB (point to self)   */
    SetIOP(CP2, (((int)&tx_tcb.ii) & CP_MAF) | CP_PCI);
                                  /* define ptr to current TCB (kick off DMA) */
                                  /* (SPORT0 transmit uses DMA ch 2)          */

    /* Set up Receive Transfer Control Block for chained DMA */
    rx_tcb.ii = rx_buf;           /* DMA destination buffer address         */
    rx_tcb.cp = &rx_tcb.ii;       /* define ptr to next TCB (point to self)   */
    SetIOP(CP0, (((int)&rx_tcb.ii) & CP_MAF) | CP_PCI);
                                  /* define ptr to current TCB (kick off DMA) */
                                  /* (SPORT0 receive uses DMA ch 0)           */
}
/*************************************************************/
/*                                                           */
/*  Send commands to the codec to set sample rate,           */
/*  analog gain, etc.                                        */
/*                                                           */
/*************************************************************/
void send_1847_config_cmds( void )
{
    // Set up pointer and counter to transmit commands.
    xmit_ptr   = regs_1847;
    xmit_count = SZ_regs_1847;

    // Wait for all commands to be transmitted.
    while( xmit_count )
        idle();

    // Wait for AD1847 autocal to start.
    while( !(rx_buf[0] & 0x0002) )
        idle();

    // Wait for AD1847 autocal to finish.
    while( rx_buf[0] & 0x0002 )
        idle();

    return;
}

/*************************************************************/
/*                                                           */
/*     Initialization routine                                */
/*                                                           */
/*************************************************************/
void init_21k( void )
{
    // Disable timer and set rate to 4 Hz.
    timer_off();
    timer_set( 10000000, 10000000 );

    // Initialize pointer and counter to transmit commands.
    xmit_count = 0;
    xmit_ptr   = regs_1847;

    // Enable interrupt nesting.
    asm( "#include <def21060.h>" );
    asm( "bit set mode1 NESTM;"  );

    // Enable timer (low priority) interrupt.
    interrupt( SIG_TMZ, timer_lo_prior );

    // Turn flag LEDs off.
    set_flag( SET_FLAG2, SET_FLAG );

    return;
}

/*************************************************************/
/*                                                           */
/*  Main loop of program.  Nothing really happens here       */
/*  since the action happens in the serial port interrupt    */
/*                                                           */
/*************************************************************/
void main ( void )
{
    int i;
    int x;
// Set LP state Arrays to zero
    for (i=0 ; i<TAPS+1 ; i++){
        LP_state1[i]=0.0;
        LP_state2[i]=0.0;
        HP_state1[i]=0.0;
        HP_state2[i]=0.0;
        }

    // Initialize some SHARC registers.
    init_21k();
    // Reset the Codec.
    set_flag( SET_FLAG0, CLR_FLAG );    /* Put CODEC into RESET  */
    for( x=0 ; x<0xffff ; x++ )         /* Hold CODEC in RESET */
        ;
    set_flag( SET_FLAG0, SET_FLAG );    /* Release CODEC from RESET */
    // Configure SHARC serial port.
    setup_sports();
    // Send setup commands to CODEC.
    send_1847_config_cmds();
    // Turn on all LEDs.
    set_flag(SET_FLAG2, CLR_FLAG);
    // Turn on the timer.
    timer_on();
    // Loop forever.
    for(;;)
    {
        idle();  /*processor sits idle until an interrupt is triggered*/
    };
}
/**************************************************************/

// End of modified file tt.c now named NAB2001Demo.c.

/**************************************************************/
```

REFERENCES:

1. Taylor, John T. & Huang Qiuting (1997). **CRC Handbook of Electrical Filters.** Boca Raton, NY: CRC Press.

2. Smith, Steven W. (1999). **The Scientist's and Engineer's Guide to Digital Signal Processing.** San Diego, CA: California Technical Publishing.

3. Oxtoby, Anthony J.A (1997). **Notes on Real-Time Digital signal Processing using the ADSP-2101 16 Bit Fixed Point Processor.** Unpublished Manuscript, Purdue University, West Lafayette, IN.

4. Tomarakos, John & Ledger, Dan (1998). **Using the Low-Cost, high Performance ADSP-21065L Digital signal Processor For Digital Audio Applications.** Norwood, MA: DSP Applications Group, Analog Devices, www.analog.com.

5. (2000) **Symetrix 9022 2x2 DSP Engine User's Guide.** Lynnwood, WA: Symetrix, Inc., www.symetrixaudio.com.

6. Alkin, Oktay (1994). **Digital Signal Processing A Laboratory Approach Using PC-DSP**. Englewood Cliffs, NJ, Prentice Hall.