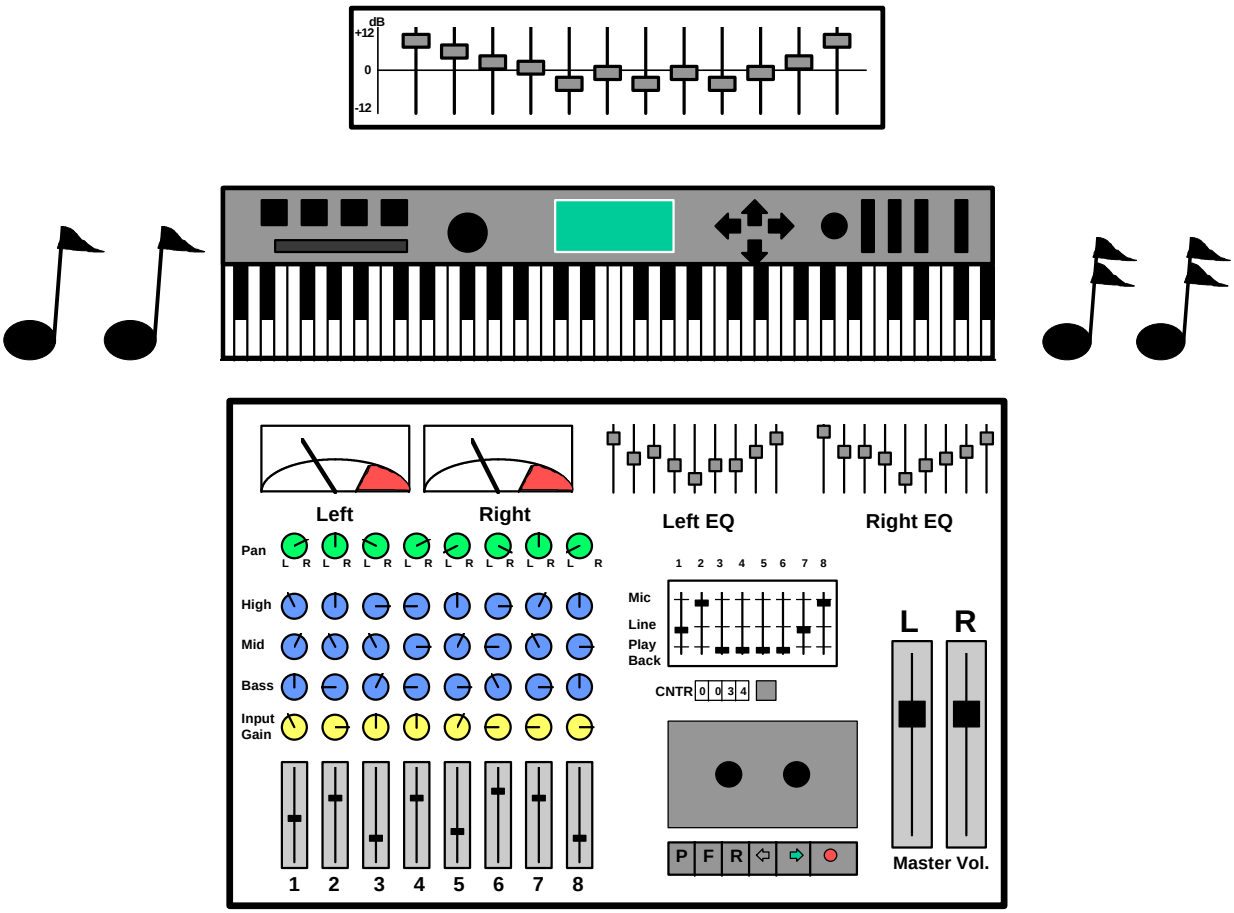
a

Using The Low-Cost, High Performance ADSP-21065L Digital Signal Processor For Digital Audio Applications

Revision 1.0 - 12/4/98



Authors:
John Tomarakos
Dan Ledger
Analog Devices DSP Applications

Using The Low Cost, High Performance ADSP-21065L Digital Signal Processor For Digital Audio Applications

Dan Ledger and John Tomarakos

DSP Applications Group, Analog Devices, Norwood, MA 02062, USA

This document examines desirable DSP features to consider for implementation of real time audio applications, and also offers programming techniques to create DSP algorithms found in today's professional and consumer audio equipment. Part One will begin with a discussion of important audio processor-specific characteristics such as speed, cost, data word length, floating-point vs. fixed-point arithmetic, double-precision vs. single-precision data, I/O capabilities, and dynamic range/SNR capabilities. Comparisions between DSP's and audio decoders that are targeted for consumer/professional audio applications will be shown. Part Two will cover example algorithmic building blocks that can be used to implement many DSP audio algorithms using the ADSP-21065L including: Basic audio signal manipulation, filtering/digital parametric equalization, digital audio effects and sound synthesis techniques.

TABLE OF CONTENTS

0. INTRODUCTION

This document will serve as an introduction for those new to digital signal processing with interests in digital audio. It will first cover important DSP features for use in audio application such as precision, speed, data format and I/O capabilities. Some basic comparative analysis will be shown for DSPs that are targeted for professional and consumer audio applications. Dynamic range requirements for high fidelity audio processing will also be discussed.

Finally, there will be some discussion on various programming techniques that can be used for creating DSP algorithms using the ADSP-21065L. Hardware circular buffering, delay lines usage, and wavetable lookups will be presented with tips on how these building blocks can be used in certain algorithms. Implementation of various digital audio algorithms will be demonstrated, with theoretical equations as well as actual coding implementations shown wherever possible. These include basic audio signal manipulation, filtering techniques, waveform synthesis techniques, digital audio effects and more.

In general, most audio algorithms fall under one of three classes: Professional, Prosumer, and Consumer Audio. For *Professional Audio*, the applications are targeted to a specific consumer base that consists of professional musicians, producers, audio engineers and technicians. *Prosumer Audio* includes many professional applications, but aimed more at lower cost, higher volume equipment sold through local music equipment retailers. *Consumer Audio* applications target a high volume customer base through consumer electronic retailers. Many basic DSP algorithms are used in all three markets segments, while others are used only in the professional or consumer space. Table 1 shows some examples of the types of products and audio algorithms used in the professional and consumer markets to help demonstrate the differentiation between the two markets.

Professional Audio Products	**Algorithms Used**
• Electronic Music Keyboards	Wavetable/FM synthesis, Sample Playback, Physical Modeling
• Digital Audio Effects Processors *(Reverb, Chorus, Flanging, Vibrato Pitch Shifting, Dyn Ran. Compression….)*	Delay-Line Modulation/Interpolation, Digital Filtering (Comb, FIR….)
• Vocal "Harmonizers" / Formant-Corrected Pitch Shifters	STFFT(Phase Vocoder), additive synthesis, frequency-domain interpolation(Lent's Alg), windowing
• Graphic and Parametric Equalizers	Digital FIR/IIR filters
• Digital Mixing Consoles	Filtering, Digital Amplitude Panning, Level Detection, Volume Control
• Digital Recording Studios (DAT) / Multichannel Digital Audio Recorders	Compression techniques: MPEG, ADPCM, AC-3
• Speaker Equalization	Filtering
• Room Equalization	Filtering
Consumer Audio Products	**Algorithms Used**
Karaoke	MPEG, audio effects algorithms
Digital Graphic Equalizers	Digital Filtering
Digital Amplifiers/Speakers	Digital Filtering
Home Theater Systems	AC-3, Dolby Prologic, THX
{Surround-Sound Receivers/Tuners}	DTS, MPEG, Hall/Auditorium Effects
Digital Versitile Disk (DVD) Players	AC-3, MPEG...
Digital Audio Broadcasting Equip.	AC-3, MPEG...
CD Players and Recorders	PCM
CD-I	ADPCM, AC-3, MPEG
Satellite (DBS) Broadcasting	AC-3, MPEG
Satellite Reciever Systems	AC-3,
Digital Camcorders	
Digital Car Audio Systems *(Digital Speakers, Amps, Equalizers Surround-Sound Systems)*	Ex. Circle Surround (RSP Tech.) Digital Filtering...

Computer Audio Multimedia Systems	3D Positioning (HRTFs), ADPCM, MPEG, AC-3 ….

Table 1 : Some Algorithms Used In Professional and Consumer Audio

1. SELECTING AN AUDIO SIGNAL PROCESSOR

The ADSP-21065L contains the following desirable characteristics to perform real-time DSP computations:

- *Fast and Flexible Arithmetic*
  Single-cycle computation for multiplication with accumulation, arbitrary amounts of shifting, and standard arithmetic and logical operations.
- *Extended Dynamic Range for Extended Sum-of Product Calculations*
  Extended sums-of-products, common in DSP algorithms, are supported in multiply/accumulate units. Extended precision of the accumulator provides extra bits for protection against overflow in successive additions to ensure that no loss of data or range occurs.
- *Single-cycle Fetch of Two Operands For Sum-of-Products Calculations*
  In extended sums-of-products calculations, two operations are needed on each cycle to feed the calculation. The DSP should be able to sustain two-operand data throughput, whether the data is stored on-chip or off.
- *Hardware Circular Buffer Support*
  A large class of DSP algorithms, including digital filters, requires circular data buffers. The ADSP-21065L is designed to allow automatic address pointer wraparounds to simplify circular buffer implementation, and thus reducing overhead and improving performance.
- *Efficient Looping and Branching for Repetitive DSP Operations*
  DSP algorithms are repetitive and are most logically expressed as loops. The 21065L's program sequencer allow looping of code with minimal or zero overhead. Also, no overhead penalties for conditional branching instructions.

1.1 General Purpose Digital Signal Processors and Decoders For Audio

There are many tradeoffs which must be considered when selecting the ideal DSP for an application. In any cost sensitive, high volume audio application with high fidelity requirements, designers look for a number of desired features at the lowest available cost. Generally, these are often speed, flexibility, data types, precision, and on-chip memory. There are a handful of DSPs and audio decoders on the market today with architectures targeted for the consumer and professional audio like the Analog Devices ADSP-21065L, Crystal Semiconductor CS4923, Motorola DSP563xx family and Zoran ZR385xx family.

1.2 Processor Speed

Processor speed generally determines how many operations can be performed within a DSP in a set amount of time. There are two units of measurement that are typically used to describe the speed of a chip: Megahertz and MIPS (millions of instructions per second). The clock speed of the chip is measured in Megahertz (MHz), or millions of cycles per second. This is the rate at which the DSP performs it most basic units of work [5]. Most DSPs perform at least one instruction per clock cycle. The second unit of measurement, MIPS describes exactly what it stands for : millions of instructions per second. It is important, however, to understand how specific DSP manufacturers define an instruction. Some manufacturers will count multiple operations executed in one instruction opcode as more than one machine instruction while other maintain the one instruction opcode equals one instruction.

1.3 On-Chip Memory

The ‘on-chip’ memory in a DSP is the memory integrated inside of the DSP which is used to store both program instructions and data. The size of on-chip memory in today’s DSP is increasing due to the changing to meet the memory requirements for evolving DSP algorithms used today. As shown in Section 3, many audio applications generally require large memory buffers. Off-chip memory can add to the system cost and increase PCB real estate, so the trend in recent years has been an increase in 'on-chip' memory integration. In addition, a 'bus bottleneck' can be produced during computationally intensive DSP routines executed off-chip, since it usually takes more than one DSP cycle to execute dual memory fetch instructions. This is because DSP manufacturers will multiplex program and data memory address and data lines together off-chip to save pins on the processor and reduce the package size, thus compromising the performance of Harvard Architecture-based processors.

1.4 I/O Capabilities and Interfaces For Processing Of Audio Samples

Another important consideration in selecting a DSP is determining if the DSP communication with the outside world is fast and efficient enough to handle a particular application's requirements. The designer must determine the transfer rate requirements for any given application in order to determine what type of DMA and peripheral interface would be adequate for the design. Many DSPs include a number of on-chip peripherals that can transmit or receive data in various binary formats between the DSP and the outside world. Many devices require a memory-mapped parallel interface or serial interface, so DSP peripheral support plays a crucial role in what types of devices can be used with the selected DSP.

1.4.1 *ADSP-21065L DMA (Direct Memory Access) Controller*

The ADSP-21065L includes a number of peripherals that can transmit or receive data from the outside world and the DSP core. On-chip DMA circuitry handles transfer of data between the DSP and external device. The ADSP-21065L host interface circuitry allows for an easy interfade to an 8, 16 or 32-bit host processor. The ADSP-21065L's *zero-overhead* DMA controller capable of transferring data between all I/O ports and the DSP core with no processor intervention.

1.4.2 *ADSP-21065L Serial Interface to Audio Converters and other Digital Audio Devices*

The ADSP-21065L has 2 serial ports to allow interface to synchronous devices as well as inter-processor communication. Enhanced modes of operation include multichannel TDM communication as well as support for standard audio protocols such as Philips I^2S, Sony CDP, and AC'97 digital audio protocols.

Synchronous Serial Ports with Time Division Multiplexing

The ADSP-21065L supports a TDM multichannel mode to easily interface to many synchronous serial devices such as Audio Codecs, Audio A/D Converters and Audio D/A Converters. Many codecs can operate in a TDM scheme where control/status information and stereo data are sent in different 'timeslots' in any given serial frame. For example, multichannel mode is often used for interfacing to the Analog Devices AD1847 multichannel SoundPort codec. AC'97 compatible devices such as the Analog Device AD1819A can also be interface to the ADSP-21065L in this mode. For example, the ADSP-21065L EZ-LAB Development board uses the AD1819a, which is a TDM protocol based on the AC-97 1.03 specification.

Philips I^2S Digital Serial Protocol

In consumer and professional audio products of recent years, the analog or digital 'front-end' of the DSP uses a digital audio serial protocol known as I^2S. Audio interfaces between various ICs in the past was hampered because each manufacturer had dedicated audio interfaces that made it extremely difficult to interface these devices to each other. Standardization of audio interfaces was promoted by Philips with the development of the Inter-IC-Sound (I^2S) bus, a serial interface developed for digital audio to enable easy connectivity and ensure successful designs. In short, I^2S is a popular 3 wire serial bus standard protocol developed by Philips for transmission of 2 channel (stereo) Pulse Code Modulation digital data, where each audio sample is sent MSB first. I^2S signals, shown in Figures 1 and 2, consist of a bit-clock, Left/Right Clock and alternating left and right channel data. This protocol can be compared to synchronous serial ports in TDM mode with 2 timeslots (or channels) active. This multiplexed protocol requires only 1 data path to send/receive 2 channels of digital audio information.

Figure 1
I²S Digital Audio Serial Bus Interface Examples

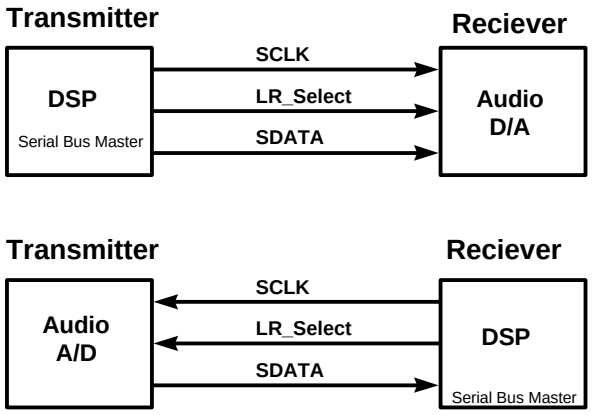
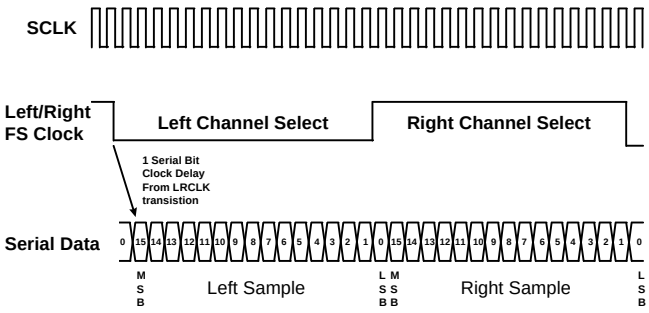


Figure 2. Example I²S Timing Diagram for 16-bit Stereo PCM Audio Data



Audio data word sizes supported by various audio converter manufacturers range can be either 16, 18, 20, or 24 bits

As a result, today many analog and digital audio 'front-end' devices support the I²S protocol. Some of these devices include:

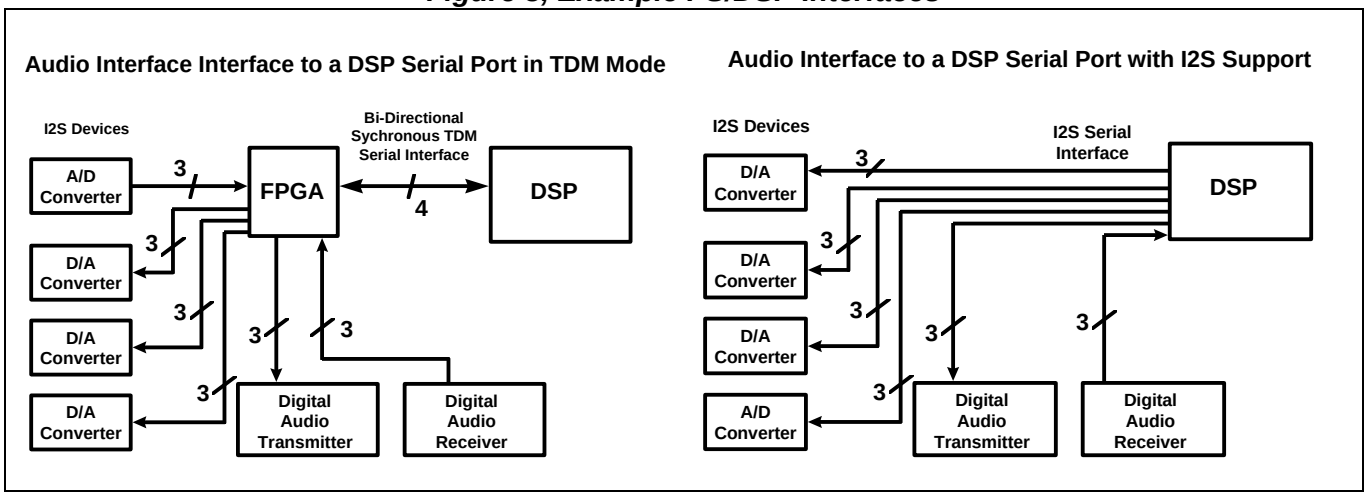
- Audio A/D and D/A converters
- PC Multimedia Audio Controllers
- Digital Audio Transmitters and Receivers that support serial digital audio transmission standards such as AES/EBU, SP/DIF, IEC958, CP-340 and CP-1201.
- Digital Audio Signal Processors
- Dedicated Digital Filter Chips
- Sample Rate Converters

The ADSP-21065L has 4 transmit and I²S serial port support for interfacing to up to 8 commercially available I²S devices Some audio DSPs and decoders also integrate analog and digital audio interfaces on-chip which results in a savings in PCB space, as well as cost savings.

Figure 3 below shows two examples for interfacing I²S devices to a DSP. DSPs without I²S support can still interface to these devices with the use of an FPGA. . This allows a designer to take use multiple I²S devices with many commercially available DSPs that support a serial time-division multiplexed scheme but do not have built in support for I²S. The timings between the devices can be resolved so that data can be aligned to a particular time-slot in the DSP TDM frame .

Thus, the ADSP-21065L's built-in support for the I²S protocol eliminates the need for the FPGA and result in a simple, glueless interface. Standard DSP synchronous serial ports with a TDM mode can still be interfaced to I²S devices, but additional glue logic via an FPGA will be required to synchronize a sample to a particular DSP timeslot.

Figure 3, Example I²S/DSP Interfaces

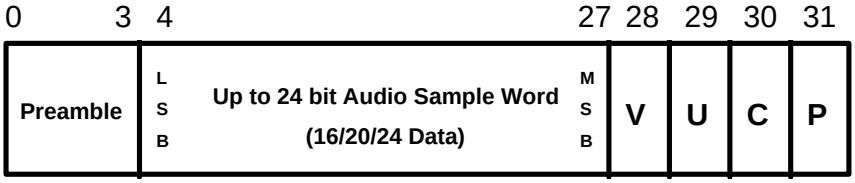


SPD/IF & AES/EBU Digital Audio Transmitters and Receivers

The ADSP-21065L's I²S interface easily allow transmission and reception of audio data using industry standard digital audio serial protocols. These devices act as a 'digital' front-end for the DSP. There are primarily 2 dominant digital protocols used today. One is used for professional audio and the other for consumer audio.

AES/EBU (Audio Engineering Society/European Broadcast Union) is a standardized digital audio bit serial communications protocol for transmitting and receiving two channels of digital audio information through a transmission line (balanced or unbalanced XRL microphone cables and audio coax cable with RCA connectors). This format of transmission is used to transmit digital audio data over distances of 100 meters. Data can be transmitted up to 24 bit resolution, along with control, status and sample rate information embedded in frame[37]. AES/EBU is considered to be the standard protocol for professional audio applications. It is a common interface that is used in interfacing different professional mixing and DAT recording devices together. The AES3-1992 Standard can be obtained from the Audio Engineering Society.

Figure 4. AES3 Frame Format



Audio Engineering Society Recommended Practice:
AES3-1992: Serial Transmission Format for Two-Channel Linearly Represented Digital Audio Data

V = Validity
U = User Data
C = Channel Status
P = Parity Bit

SPD/IF (Sony/Philips Digital Interface Format) is based on the AES/EBU standard in operating in 'consumer' mode. The physical medium is an unbalanced RCA cable. The consumer mode carry less control/status information. Typical applications where this interface can be found is in home theater equipment and CD players.

Digital Audio Receivers typically receive AES/EBU and SP/DIF information and convert the audio information into the I²S (or parallel) format for the ADSP-21065L, as well as provide status information that is received along with the audio data. Digital Audio Transmitters can take an I²S audio stream from the ADSP-21065L and transmit the audio data along with control information in AES/EBU and SPD/IF formats.

1.5 DSP Numeric Data Formats : Fixed/Floating Point Arithmetic

Depending on the complexity of the application, the audio system designer must decide on how much computational accuracy and dynamic range will be needed. The most common native data types are explained briefly in this section. 16- and 24-bit fixed-point DSPs are designed to compute integer or fractional arithmetic. 32-bit DSPs like ADI's 2106x SHARC family were traditionally offered as floating point devices, however, this popular family of DSPs can equally perform both floating point arithmetic and integer/fractional arithmetic.

1.5.1 *1.6.0 16/24/32-Bit Fixed-Point Arithmetic*

DSPs that can perform fixed point operations typically use a two's complement binary notation for representing signals. The representation of the fixed-point format can be signed (twos-complement) or unsigned integer or fractional notation. Most DSP operations are optimized for signed fractional notation. For example, the numeric format in signed fractional notation would correspond to the samples produced from a 5 V A/D converter as shown in figure 4 below. The highest full scale positive fractional number would be 0.999…. while the highest full scale negative number is -1.0.

Figure 4

Signed Two's Complement Representation of Sampled Signals

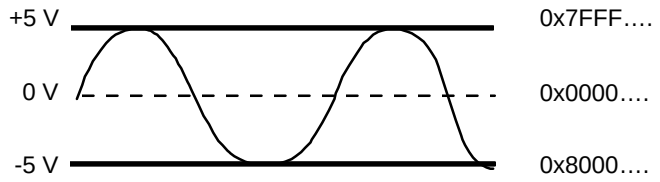
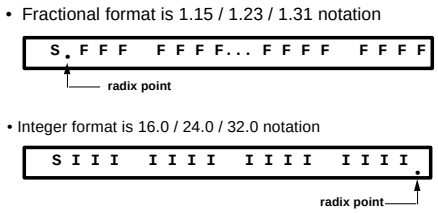


Figure 5

Fractional And Integer Formats



In the fractional format, the binary point is assumed to be to the to the left of the LSB (sign bit). In the integer format, the binary point is to the right of the LSB (figure 5).

1.5.2 *Floating-Point Arithmetic*

The native floating point capability of the ADSP-21065L has data paths that are 32 bits wide., where 24 bits represent the mantissa and 8 bits represent the exponent. The 24 bit mantissa is used for precision while the exponent is for extending the dynamic range. For 40 bit extended precision, 32 bits are used for the mantissa while 8 bits are used to represent the exponent (figures 6 and 7).

Figure 6.

IEEE 754 32-Bit Single Precision Floating-Point Format

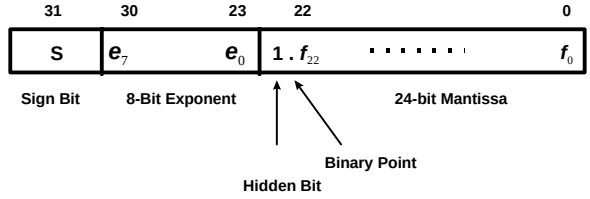
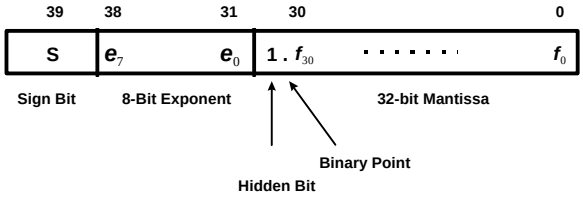


Figure 7.

40-Bit Extended Precision Floating-Point Format



A 32-bit floating point number is represented in decimal as:

$$n = m \times 2^{e-128}$$

It's binary numeric IEEE format representation is stored on the ADSP-21065L as:

$$n = (-1)^S \times 2^{e-128} (1.b_0 b_1 b_2 \text{---} b_{23})$$

It is important to know that the IEEE standard always refers to the mantissa in signed-magnitude format, and not in twos-complement format. So the extra hidden bit effectively improved the precision to 24 bits and also insures any number ranges from 1 (1.0000….00) to 2 (1.1111….11) since the hidden bit is always assumed to be a 1.

Figure 7 shows the 40-bit extended precision format available that is also supported on the ADSP-2106x family of DSPs. With extended precision, the mantissa is extended to 32 bits. In all other respects, it is the same format as the IEEE standard format. 40-bit extended precision binary numeric format representation is stored as:

$$n=(-1)^{s} x\, 2^{e-128}(1.b_0 b_1 b_2 \text{---} b_{30})$$

Floating Point Arithmetic is traditionally used for applications that have high dynamic range requirements. Typically in the past, trade-offs were considered with price vs performance. Until recently, the higher cost made 32-bit floating point DSPs unreasonable for use in audio. Today, designers can achieve high quality audio using 32-bit fixed or floating point processing with the introduction of the ADSP-21065L, at a cost comparable to 16-bit and 24-bit DSPs.

1.6 The Importance Of Dynamic Range In DSP-Audio Processing

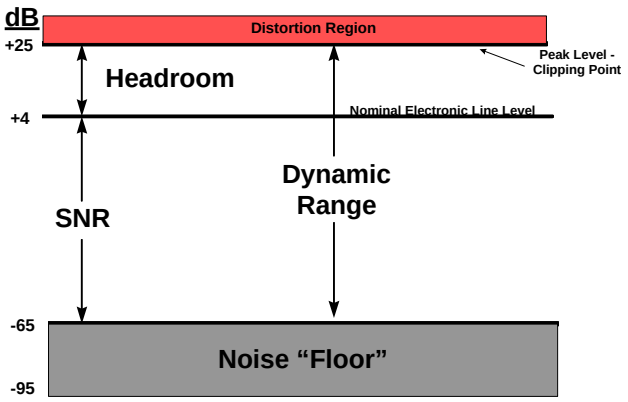
One of the top considerations when designing an audio system is determining acceptable signal quality for the application. Audio equipment retailers and consumers often use the phrase 'CD-quality sound' when referring to high dynamic range audio. Compare sound quality of a CD player to that of an AM radio broadcast. For higher quality CD audio, noise is not audible, especially during quiet passages in music. Lower level signals are heard clearly. But, the AM radio listener can easily hear the low level noise at very audible levels to the point where it can be very distracting. Thus, as an audio signals dynamic ranges, the better distinction one can make for low level audio signals while noise becomes inaudible. The table below shows some comparisons of signal quality for some audio applications, devices and equipment.

Audio Device/Application	Typical Signal Quality
AM Radio	48 dB
Analog Broadcast TV	60 dB
FM Radio	70 dB
Analog Cassette Player	73 dB
Video Camcorder	75 dB
ADI SoundPort Codecs	80 dB
16 Bit Audio Converters	90 to 95 dB
Digital Broadcast TV	85 dB
Mini-Disk Player	90 dB
CD Player	92 to 96 dB
18-bit Audio Converters	104 db
Digital Audio Tape (DAT)	110 dB
20-bit Audio Converters	110 dB
24-bit Audio Converters	110 to 120 dB
Analog Microphone	120 dB

Table 2 : Some Dynamic Range Comparisons

Figure 9.

Audio Signal Level (dBu) Relationship Between Dynamic Range, SNR and Headroom



Important Audio Definitions [Davis & Jones, 17] (See Figure 9 for graphic representation)

- **Decibel -** Used to describe sound level (sound pressure level) ratio, or power and voltage ratios:

  $$dB_{Volts}=20\log(V_o/V_i),\quad dB_{Watts}=10\log(P_o/P_i),\quad dB_{SPL}=20\log(P_o/P_i)$$

- **Dynamic Range** - The difference between the loudest and quietest representable signal level, or if noise is present, the difference between the loudest (maximum level) signal to the noise floor. Measured in dB.

  Dynamic Range = (Peak Level) - (Noise Floor) dB

- **SNR (Signal-To-Noise Ratio, or S/N Ratio)** - The difference between the nominal level and the noise floor. Measured in dB. Other authors define this for analog systems as the ratio of the largest representable signal to the noise floor when no signal is present[6], which more closely parallels SNR for a digital system.
- **Headroom** - The difference between nominal line level and peak level where signal clipping occurs. Measured in dB. The larger the headroom, the better the audio system will handle very loud signal peaks before distortion occurs.
- **Peak Operating Level** - The maximum representable signal level at which point clipping of the signal will occur.
- **Line Level** - Nominal operating level (0 dB, or more precisely between -10 dB and +4 dB)
- **Noise Floor** - The noise floor for human hearing is the average level of 'just audible' white noise. Analog audio equipment can generate noise from components. With a DSP, noise can be generated from quantization errors.

[One can make an assumption that the headroom + S/N ration of an *electrical analog signal* equals the dynamic range (although not entirely accurate since signals can still be audible below the noise floor)].

In undithered DSP-based systems, the SNR definition above is not directly applicable since there is no noise present when there is no signal. In digital terms, dynamic range and SNR (Figure 11) are often both used to describe the ratio of the largest representable signal to the quantization error or noise floor [R. Wilson, 9]. The wordlength for a given processor determines the number of quantization levels that are available. For an ***n-bit*** data word would yield $\mathbf{2^n}$ quantization levels (some examples shown in Table 4 below). The higher number of bits used to represent a signal will result in a better approximation

of the audio signal and a reduction in quantization error (noise), which produces and an increase in the SNR. In theoretical terms**, there is an increase in the signal-to-quantization noise or dynamic range by approximately 6 dB for each bit added to the wordlength of an ADC, DAC or DSP**. For example, figure 10 demonstrates how 32-bit or 24-bit processing can more accurately represent a given value as compared to 16-bit processing. 24-bit processing can more accurately represent a signal 256 times better than 16-bit processing, while the ADSP-21065L's 32-bit processing can more accurately represent signals 65,536 times better than that for 16-bit processing, and 256 times more accurately than that of a 24-bit processor.

N Quantization Levels for n-bit data words ($N = 2^n$ levels)
2^8 = 256
2^{16} = 65,536
2^{20} = 1,048,576
2^{24} = 16,777,216
2^{32} = 4,294,967,296
2^{64} = 18,446,744,073,729,551,616

Table 4: An n-bit data word yields 2^n quantization levels

Figure 10.

Fixed Point DSP Quantization Level Comparisons

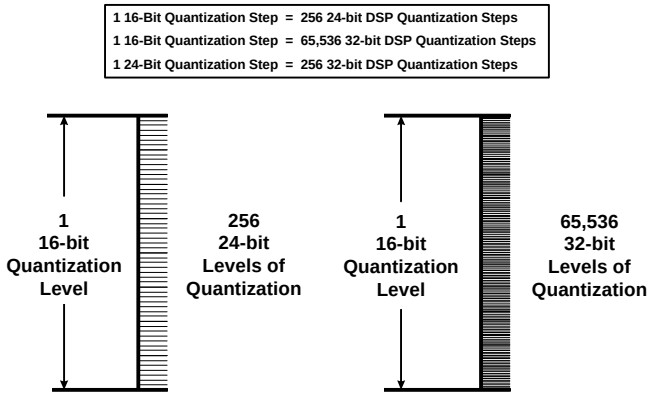
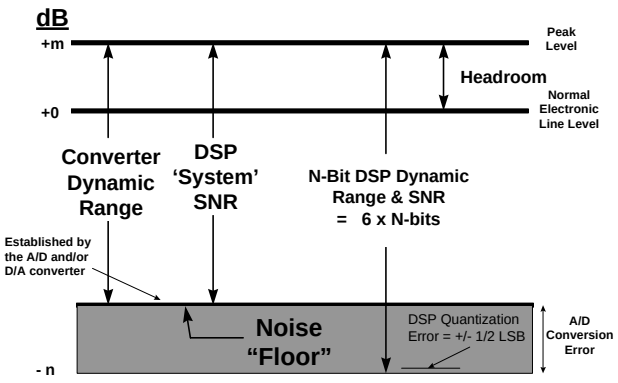


Figure 11.

DSP/Converter SNR and Dynamic Range



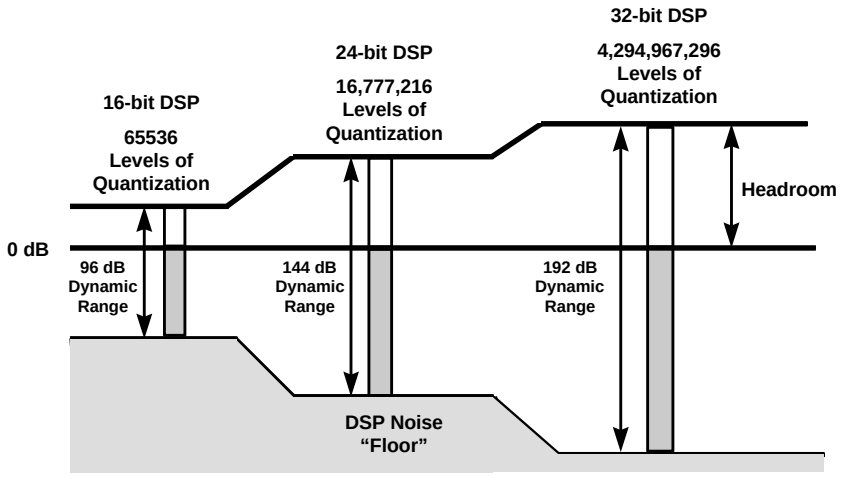
The maximum representable signal amplitude to the maximum quantization error for of an ideal A/D converter or DSP-based digital system is calculated as:

$$\mathbf{SNR_{A/D(RMS)}(dB)} = 6.02\mathbf{n} + 1.76\ \mathbf{dB}$$

$$\mathbf{Dynamic\ Range(dB)} = 6.02\mathbf{n} + 1.76\ \mathbf{dB} \cong 6\mathbf{n}$$

1.76 dB is based on sinusoidal waveform statistics, and would vary for other waveforms [ ], and ***n*** represents the data word length of the converter or the processor.

Figure 12.
Fixed-Point DSP Dynamic Range Comparisons



Fixed Point Dynamic Range per Bit of Resolution = 6dB
16 bit fixed point precision yields 96 dB, 16 x (6 dB per bit) = 96 dB
24 bit fixed point precision yields 144 dB, 24 x (6 dB per bit) = 144 dB
32 bit fixed point precision yields 192 dB, 32 x (6 dB per bit) = 192 dB

Figure 1 above compares the dynamic ranges between commercially available 16, 24 and 32-bit fixed point processors (assuming single-precision arithmetic). As stated earlier, the number of data-word bits used to represent a signal directly affects the SNR and quantization noise introduced during the sample conversions and arithmetic computations.

Additional Fixed Point MAC Unit Dynamic Range for DSP Overflow Prevention

Many DSPs include additional bits in the MAC unit to prevent overflow in intermediate calculations. Extended sums-of-products are common in DSP algorithms are achieved in the MAC unit with single cycle multiply accumulates placed in an efficient loop structure. The extra bits of precision in the accumulator result register provide extended dynamic range for the protection against overflow in successive multiplies and additions, thus ensuring that no loss of data or range occurs. Below is a table comparing the extended dynamic ranges of 16-bit, 24-bit, and 32-bit DSPs. Note that the ADSP-21065L has a much higher extended dynamic range than 16 and 24 bit DSPs when executing fixed point multiplication instructions.

N-bit DSP	N-bit x N-bit Multiply	Additional MAC Result Bits	Precision in MAC Result Register	Additional Dynamic Range Gained	Resulting MAC Dynamic Range
16-bit DSP	32-bits	8-bits	40-bits	48 dB	240 dB
24-bit DSP	48-bits	8-bits	56-bits	48-dB	336 dB
32-bit 21056L	64-bits	16-bits	80-bits	96-dB	480 dB

Developing Audio Algorithms Free From Noise Artifacts

If a digital system produces processing artifacts which are above the noise floor of the input signal, then these artifacts will be audible under certain circumstances e.g. when an input signal is of low intensity or limited frequency content. Therefore, whatever the dynamic range of a high-quality audio input, be it 16, 20 or 24 bit samples, the digital processing which is performed on it should be designed to prevent processing noise from reaching levels at which it may appear above the noise

floor of the input and hence become audible. For a digital filter routine to operate transparently, the resolution of the processing system must be considerably greater than that of the input signal so that any errors introduced by the arithmetic computations are smaller than the precision of the ADC or DAC. In order for the DSP to maintain the SNR established by the A/D converter, all intermediate DSP calculations require the use of higher precision processing [9,15]. The effects of a finite word length that can degrade an audio signal's SNR can be the result of any of the following:

- *A/D Conversion Noise*

Finite precision of an input data word sample will introduce some inaccuracy for the DSP computation as a result of the nonlinearities inherent in the A/D Conversion Process.

- *Quantization Error of Arithmetic Computations From Truncation and Rounding*

DSP Algorithms such as Digital Filters will generate results with must be truncated or rounded up. In IIR filters where feedback is implemented, these errors will tend to accumulate.

- *Computational Overflow*

Whenever the result of an arithmetic computation is larger than the highest positive or negative full scale value, an overflow will occur and the true result will be lost.

- *Coefficient Quantization*

Finite Word Length of a filter coefficient can affect pole/zero placement and affect a digital filters frequency response. The ADSP-21065L enables precise placement of poles/zeros with 32-bit accuracy.

- *Limit Cycles*

Occur in IIR filters from truncation and rounding of multiplication results or addition overflow. These often cause periodic oscillations in the output result, even when the input is zero.

"The overall DSP-based audio system dynamic range is only as good as the weakest link"

Thus, in a DSP-based audio system, this means that any one of the following sources or devices in the audio signal chain will determine the dynamic range of the overall audio system:

- the analog input signal from a microphone or other device
- the ADC word size and conversion errors
- DSP word length effects: DSP quantization errors from truncation and rounding, and filter coefficient quantization
- output DAC
- other connecting equipment used to further process the audio signal

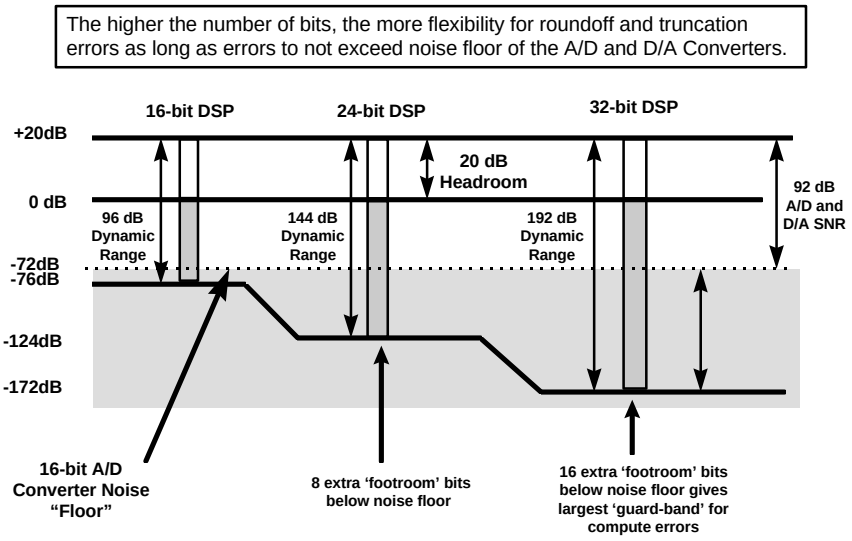
Fielder [38] demonstrates the dynamic range requirements for consumer CD audio requires 16-bit conversion/processing while the minimum requirement for professional audio is 20-bits (based on perceptual tests performed on human auditory capabilities). Dynamic range application requirements for high fidelity audio processing can be categorized into two groups:

'Consumer CD-Quality' Audio System uses 16-bit conversion with typical dynamic ranges between 85-93 dB
'Professional-Quality' Audio System uses 20-24 bit conversion with dynamic Range between 110-122 dB

Maintaining 96 dB 16-bit 'CD-Quality' Audio During DSP Algorithm Calculations

When processing audio signals, the DSP must keep quantization errors introduced by arithmetic calculations lower than the converter noise floor. Consider a 'CD-quality' audio system. If the DSP is to process audio data from a 16 bit A/D converter (ideal case), a 96 dB SNR must be maintained through the algorithmic process in order to maintain a 'CD quality' audio signal (6x16=96dB). Therefore, it is important that all intermediate calculations be performed with higher precision than the 16-bit ADC or DAC resolution. Errors introduced by the arithmetic calculations can be minimized when using higher data-word processing (single or extended double precision) .

Figure 13. Fixed-Point DSP Noise Floor with a typical 16-bit ADC/DAC at 92 dB



As an comparison example, let's take a look at the processing of audio signals from a 16-bit A/D converter that has a dynamic range close to it's theoretical maximum, in this case with a 92 dB dynamic range and SNR (see Figure 13 above). The 16-bit DSP only has 4 dB higher SNR higher than the A/D converter. For moderate to complex audio processing using single precision arithmetic, the 16-bit DSP data path will not be adequate as a result of truncation and round-off errors that can accumulate. As shown in the Figure 15 below, errors produced from the arithmetic computations will be seen by the output D/A converter. The same sample processing algorithm implemented on a higher resolution DSP would ensure these errors are not seen by the D/A converter. The 24-bit DSP has 8 bits below the converter noise floor to allow for errors, while the the ADSP-21065L (32-bit DSP) has 16-bits below the noise floor, allowing for the greatest SNR computation flexibility in developing stable, noise free audio algorithms.

Thus, when using a 16-bit converter for 'CD-quality' audio, the general recommendation is to use a higher resolution processor (24/32-bit) since additional bits of precision gives the DSP the ability to maintain the 96dB SNR of the audio converters [9,15, 28]. Double precision math can still be used for smaller data word DSPs if software overhead is available, although the real performance of the processor can be compromised. A 16-bit DSP using single precision processing would only suffice for low cost audio applications where processing is not too complex and SNR requirements are around 75 dB (audio cassette quality).

Figure 14.
16-bit A/D Samples at 96 dB SNR

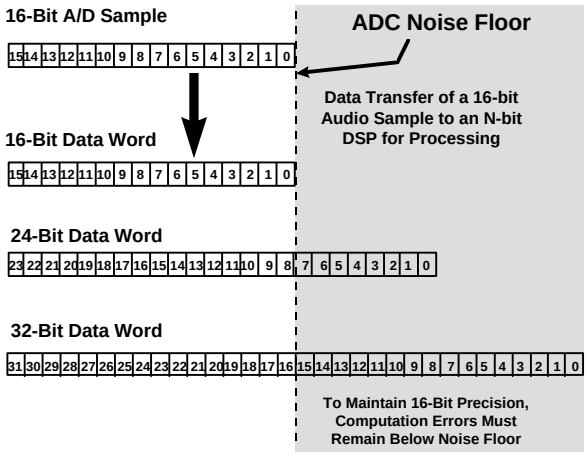
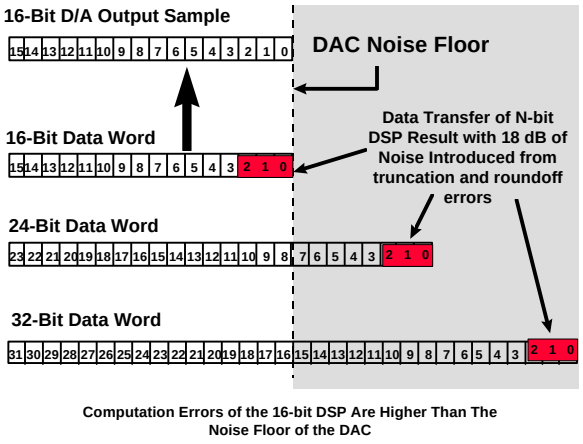


Figure 15.
16-bit D/A Output Samples with Finite Length Effects



Summary of requirements for maintaining 16-bit accuracy, 96 dB SNR:

- All intermediate calculations must be performed using higher precision filter coefficients and higher precision storage of intermediate samples in larger data word computation registers and/or memory to ensure the noise floor of the algorithm/filter is less than the final truncated output result by the D/A converter
- At least 24 bits are required if the quality of 16 bits is to be preserved. However, even with 24-bit processing, it has been demonstrated that care would need to be taken to ensure the noise floor of the digital filter algorithm is not greater than the established noise floor of the 16 bit signal, especially for recursive IIR audio filters. [R. Wilson, 9].
- When processing 16/18/20 bit audio data, the use of 32-bit processing is especially useful for complex recursive processing using IIR filters. For example, parametric/graphic equalizer implementations using cascaded 2nd order IIR filters, and comb/allpass filters for audio are more robust using 32-bit math. The ADSP-21065's 32-bit capability reduces the burden from the DSP programmer to ensure that the quantization error from computations does not go above the ADC/DAC noise floor.
- The ADSP-21065L's 32-bit processing can give an additional 48 dB 'guard' benefit to ensure 16-bit signal quality is not impaired during multistage recursive filter computations and and multiple algorithmic passes before obtaining the final result for the DAC.

Processing 110-120 dB, 20-/24-bit Professional-Quality Audio

When the compact disc was launched in the early 1980s, the digital format of 16-bit words sampled at 44.1 kHz, was chosen for a mixture of technical and commercial reasons. The choice was limited by the quality of available analog-to-digital converters, by the quality and cost of other digital components, and by the density at which digital data could be stored on the medium itself. It was thought that the format would be sufficient to record audio signals with all the fidelity required for the full range of human hearing. However, research has shown that this format is imperfect in some respects.

Firstly, **the sensitivity of the human ear is such that the dynamic range between the quietest sound detectable and the maximum sound which can be experienced without pain is approximately 120dB**. The 16-bit words used for CD allow a maximum dynamic range of 96 dB although with the use of dither this is reduced to about 93 dB. Digital conversion technology has now advanced to the stage where recordings with a dynamic range of 120dB or greater may be made, but compact disc is unable to accurately carry them.

While 16-bit, 44.1 kHz PCM digital audio continues to be the standard for high quality audio in most current applications, such as CD, DAT and high-quality PC audio, recent technological developments and improved knowledge of human hearing have created a demand for greater word lengths in the professional audio sector. 18, 20 and even 24 bit analog-to-digital converters are now available which are capable of exceeding the 96dB dynamic range available using 16 bits. Many recording studios now routinely master their recordings using 20-bit recorders, and quickly moving to 24 bits. These technological developments are now making their way into the consumer and so-called “prosumer” audio markets. The most conspicuous incarnation is DVD which is capable of carrying audio with up to 24-bit resolution. New DVD standards are extending the digital formats to 24-bits at sample rates of 96 kHz and 192 kHz formats. Other products include DAT recorders which can sample at 96kHz. Many professional audio studio manufacturers now offer DAT recorders with 24-bit conversion, 96 kHz sampling rage. In fact, three trends can be identified which have influenced the current generation of digital audio formats which are set to replace CD digital audio, and these may be summarized as follows:

- Higher resolution - 20 or 24 bits per word
- Higher sampling frequency - typically 96 kHz
- More audio channels

With many converter manufacturers introducing 24-bit A/D and D/A converters to meet emerging consumer and professional audio standards, processing of audio signals will require at least 32-bit processing in order to offer sufficient precision to ensure that a filter algorithm's quantization noise artifacts will not exceed the 24-bit input signal.

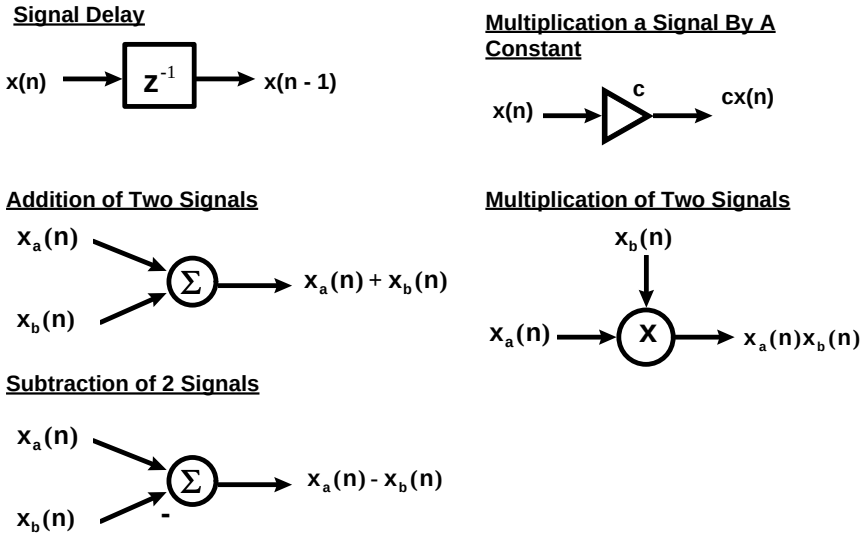
2. USEFUL DSP HARDWARE/SOFTWARE BUILDING BLOCKS FOR AUDIO

This section will briefly review common DSP operations, and show how a DSP programmer can take advantage of the ADSP-21065L processor specific characteristics that allow the designer to easily write DSP algorithms. This DSP was designed to allow efficient coding of real-time signal processing operations such as convolution and vector operations while allowing fast, efficient memory accesses.

2.1 Basic Arithmetic Operations

DSPs have the ability to perform a large range of mathematical operations. All DSPs must be able to perform simple operations like addition, subtraction, absolute value, multiplication, logical operations (AND, OR,..). The ADSP-2106x family with it's floating-point support can perform more advanced functions like divisions, logarithms, square roots and averages very efficiently. Figure 16 below summarizes some common code building blocks:

Figure 16.
Common DSP Building Block Operations



2.2 Implementing Convolution With Zero-Overhead Looping, Multiply/Accumulate Instructions (MAC), and Dual Memory Fetches

A common signal processing operation is to perform a running sum on an input and an impulse response to a system. Convolution involves multiplying two sets of discrete data and summing the outputs as seen in the convolution equation below:

$$y(n) = \sum_{m} x(m)h(n - m)$$

Examining this equation closely shows elements required for implementation. The filter coefficients and input samples need to come from 2 memory arrays. They need to be multiplied together and added to the results of previous iterations. So memory arrays, multipliers, adders, and a loop mechanism are needed for actual implementation. The ADSP-2106x DSPs can fetch two data words from memory(x(n) and h(n-m)), multiply them and accumulating the product (MAC instruction) to a previous results in one instruction cycle. When used in a zero-overhead loop, digital filter implementation becomes extremely optimized since no explicit software decrement, test and jump instructions are required.

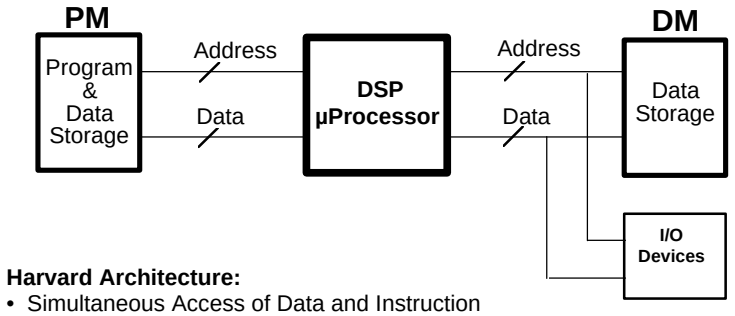
Multiply / Accumulates (MAC)

Many DSP architectures like the SHARC family include a fixed-point MAC in the computational section to allow a multiply and accumulate in 1 instruction cycle. The DSP needs this support in order to multiply an input sample with a filter coefficient, and add the result to the previous accumulator results.

Dual Memory Fetches with a Modified Harvard Architecture

DSP architectural features are designed to perform these computations as quickly as possible, usually within 1 instruction cycle. To perform an operation shown above, a DSP architecture should allow: 1 multiplication with an addition to a previous result, fetch a sample from memory and fetch a coefficient within 1 instruction cycle. To perform the complete convolution operation, an efficient loop hardware should be able to efficiently loop through the number of iterations of the MAC & dual memory fetch instruction.

Figure 17.
The Harvard Architecture



Harvard Architecture:
- Simultaneous Access of Data and Instruction

Variations of Harvard Architecture:
- Single-cycle Access of 2 Data Memories and Instruction (can be from Cache)
- Gives Three Bus Performance with only 2 Busses

The ADSP-21065L uses a Modified Harvard Architecture (Figure 17 above) further to enable 2 data transfers and 1 instruction (such as a MAC) to be executed in 1 instruction cycle due to the fact that there are 2 separate memory spaces (program and data) and either a cache or separate PM data bus. The ability to also store data in the Program Memory Space allows the DSP to execute an instruction and performing 2 memory moves in any given cycle. On-chip memory storage allows the DSP programmer to place arithmetically intensive filter computations in internally to maintain single cycle dual memory fetches.

2.3 Hardware Circular Buffering For Efficient Storage/Retrieval Of Audio Samples

An important feature for repetitive DSP algorithms is the use of circular buffering. A circular buffer is a finite segment of the DSPs memory defined by the programmer that is used to store samples for processing (Figure 18). The ADSP-2106x DSPs have data *address generation units* that automatically generate and increment pointers [18] for memory accesses. When data is stored/retrieved from a circular buffer in consecutive locations, the address generation units will ensure that the indirect pointer to the buffer will automatically wrap to the beginning memory address after exceeding the buffer's endpoint (Figure 19). When circular buffering is implemented in hardware, the DSP programmer does not have to be concerned with additional overhead of testing and resetting of the address pointer so that it does not go out of the boundary of the buffer.

Figure 18.
Hardware Circular Buffering

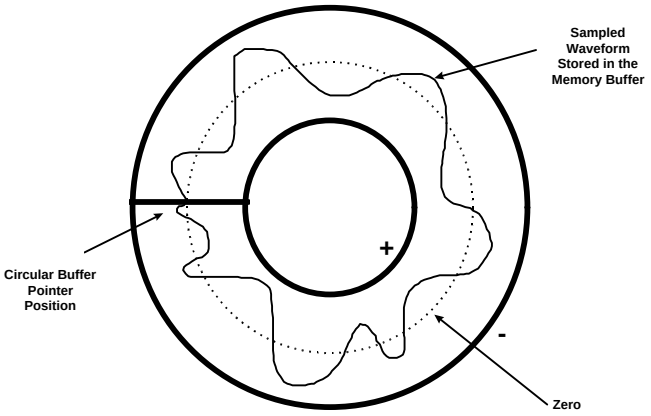
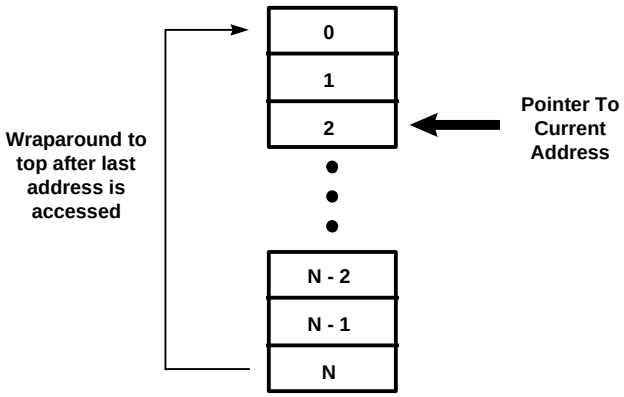


Figure 19.
Circular Buffer of Length N



2.4 Zero-Overhead Looping

The Control Unit in the DSP microcomputer must provide efficient execution of data as quickly as possible. For digital filter routines, a running sum of MAC operations is typically executed in fast loop structures with what is know as *zero overhead*, meaning the branching, loop decrementing, and termination test operations are build into the control unit hardware, saving precious DSP cycles without having to include addition loop construct operation in hardware. Once the loop is initialized, there is no software overhead. The example assembly pseudocode below shows how hardware-controlled loops with zero overhead can produce code that is 3 times faster, after the 1 cycle that is required for setting up the DO Loop Instruction.

• Software Loop Example:

```
                CNT = 10;
Loop1:          Mult/Acc AR, X, Y;
                Decrement CNT;
                JNE     Loop1;
```

• Zero Overhead Hardware Loop Example:

```
                CNT=10;
                Do Loop Until Mult_Acc Done:
Mult_Acc:       Mult/Acc AR, X, Y;
```

2.5 Block Processing vs. Sample Processing

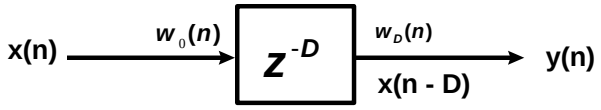
DSP algorithms usually process signals by either block processing or sample processing [2]. For *block processing*, data is transferred to a DSP memory buffer and then processed each time the buffer fills with new data. Examples of such algorithms are fast fourier transforms and fast convolution. The processing time requirement is based on the sample rate times the number of locations in the memory buffer.

In *sample processing* algorithms, each input sample is processed on a sample-by-sample basis through the DSP routine as each sample becomes available. Sampled data is usually passed from a peripheral (such as a serial port) and transferred to an internal register or memory location so it is made available for processing. The is the preferred method when implementing real-time digital filters for infinite duration. For infinite duration sequences, once the DSP is initialized, it will forever process data coming in and output a result as long as the DSP system is powered . So for real-time digital IIR/FIR filters and digital audio effects, sample processing will be the method used for most examples to be covered in this paper. As we will see in the next section, some digital filters and audio effects use sample processing techniques with delay-lines.

2.6 Delay-Lines

The *Delay-Line* is a basic DSP building block which is can be used to filter a signal or produce time-based effects such as chorusing, flanging, reverberation and echo. The basic design for any time-delay effect is to simply delay an incoming signal and output the result by some fixed or variable length of time (See general delay line structure in Figure 20). The DSP delay-line can be implemented by the following technique [17]: Using an ADC, an input analog signal is converted to it's equivalent binary numeric representation. These discrete samples are then placed in the DSP's internal (or external) RAM. To move through the delay-line, the DSP uses addressing generation/modification methods to automatically increment (or decrement) an address pointer after each input sample is stored so the other samples are stored in consecutive memory locations. At the same time, previously stored samples are sent to a DAC from another 'tapped' address location in the memory buffer. The DAC converts the digital result back to its analog equivalent. Figure 20 below shows the DSP structure of the delay-line:

Figure 20. Delay Line with buffer size D



$$\text{Delay(sec)} = D_{\text{Buff. Size}} \times T_{\text{Samp. Rate}}$$

$$\text{Delay(sec)} = \frac{D_{\text{Buff. Size}}}{f_{\text{Samp. Rate}}}$$

The delay time of an DSP-processed audio signal is determined by:

1. Delay Line Buffer Size - number of words (address locations) defined for the buffer.
2. Sampling Rate - determined usually by the audio converters. This also corresponds with the rate at which data is received, processed and returned by the DSP (usually within an interrupt service routine). The address in the buffer is incremented every time samples are stored/retrieved.

The I/O difference equation is simply:

$$y(n) = x(n - D)$$

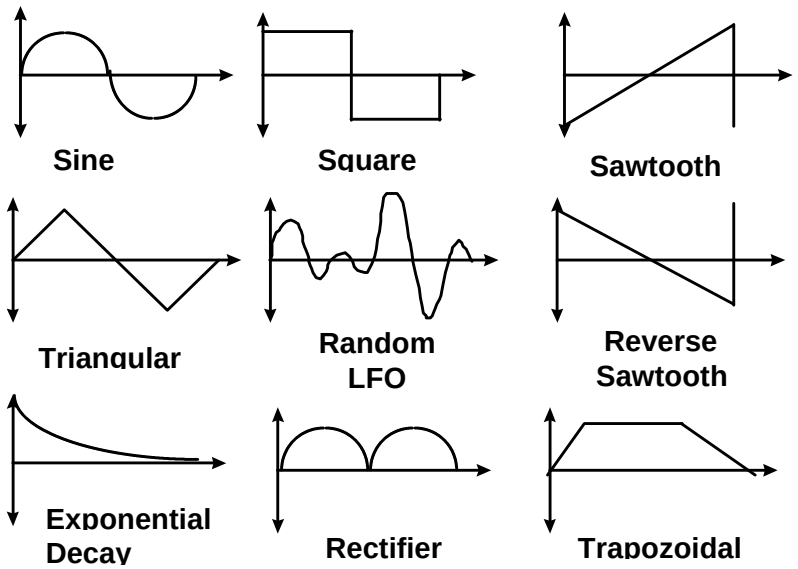
Usually, the sampling rate of the A/D or D/A converter is related to the rate at which the DSP's interrupt service routine is called for data processing. The DSP interrupt rate usually depends on the AD/DA converters since the converters are connected to the DSP's serial ports or are using a hardware interrupt pin to notify the DSP when data is being transmitted or received.

To increase the delay of a signal, either the buffer size must be increased to store more samples, or the sampling rate can be decreased to increase the delay. Tradeoffs must be considered when choosing longer delay times. Sometimes a DSP only has a limited amount of memory available. The higher the bandwidth requirement of the incoming signal, the more memory storage required by the DSP. But, by decreasing the sampling rate, the bandwidth is reduced. In some cases this is not a problem. For example, human voices or stringed instruments have a bandwidth of only up to 6 kHz. In such cases, a smaller sampling rate will not limit the with the frequency range of the instrument.

2.7 Signal Generation With Look-Up Tables

Methods of signal generation for wavetable synthesis, delay-line modulation and tremolo effects can be produced by using a periodic lookup of a signal stored in the DSP's data memory. Wavetable Generators can be used to implement many time-delay modulation effects an amplitude effects such as the chorus, flanger, vibrato, and tremolo. The figure below shows some of the more common signals that can be easily stored in memory for use in audio applications.

Figure 21.
Example Wavetable Storage Signals Useful For Audio Algorithms



Most high level languages such a C/C++ have build in support to generate trigonometric functions. Real-time Embedded System Software Engineers who program DSP algorithms mostly in assembly do not have the flexibility of a high level language when generating signals. Various methods proposed by Crenshaw [8], Orfanidis [2] and Chrysafis [39] can be used for generating sinusoidal/random signals in a DSP. Signal generation can be achieved by:

1. Making a subroutine call to a Taylor Series function approximation for trigonometric signals, uniform/Gaussian random number generator routine for random white noise generation.
2. Using a table lookup
3. Using hold/linear interpolation operations between consecutive locations in the wavetable to increase the resolution of the stored signal.

The advantage of using a wavetable to generate a signal is that it is simple to generate signal simply by performing a memory read from the buffer, therefore saving DSP cycle overhead. The wavetable can be implemented as a circular buffer so that the signal stored is regenerated over and over. The larger the buffer, the purer the signal that can be generated. With larger internal memory sizes integrated on many DSPs or the use of low cost commodity SDRAM, the option of using a look-up table is more easily achievable than in the past. To save memory storage, the size of the table can be reduced by a factor of 2, and as suggested above, the DSP can interpolate between 2 consecutive values. For example, a wavetable buffer can contain 4000 locations to represent 1 period of a sine wave, and the DSP can interpolate in between every value to produce 8000 elements to construct the signal. This is not a bad approximation for generating a decent sounding tone

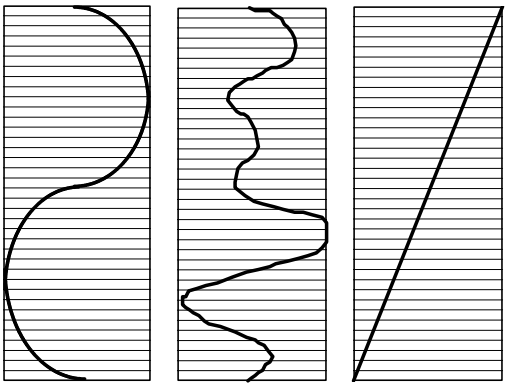
What is the best way to progress through the table? The general recommendation for accessing data from the table would be to declare the wavetable in the DSP program as a circular buffer instead of as a linear buffer (see some examples in Figure 22 below). This will allow the signal to be replayed over and over without the program having to check to see if the pointer needs to be reset. Two methods can be used to progress through the lookup table:

1. ***Sample-Rate Dependent Update:*** On method for updating a wavetable pointer is sample-rate dependent update, where a new lookup value is generated every time the sample processing algorithm is entered (typically via an interrupt service routine). This synchronization with the sample rate will not introduce possible aliasing artifacts in implementing delay line modulation.

2. ***DSP Timer Expire Update:*** Another method, would be to update the value in the table using the DSP's on chip programmable timer. Every time the timer expires and resets itself, the timer ISR can update the pointer to the wavetable buffer. This method allow movement through a table that is not relative to the converter's sampling rate, allowing for more flexible and precise timing of signal generation or delay-line modulation.

For certain digital audio effects such as flanging/chorusing/pitch shifting, lookup table updates can be easily achieved using the programmable timer as well as via the audio processing ISR. Delay-line modulation value can be easily updated by using the programmable timer or an interrupt counter to process the parameter used to determine how far back in the delay-line buffer the DSP's data addressing unit needs to fetch a previously stored sample. A sine wavetable can be used to implement many time delay modulation effects an amplitude effects such as the chorus, flanger, vibrato, and tremolo. *Random Low-frequency oscillator* (LFO) Tables can be used to implement realistic chorus effects [2]. Using a sawtooth wavetable will be useful for shifting the pitch of a signal [16]. We will look at these examples in more detail in subsequent sections.

Figure 22.



Many methods exist for generating wavetable data files for inclusion into a DSP program. An easy way is to use a mathematical software packages such a MATLAB to create data files. Signal tables can even be created using Microsoft Excel. C source also exists on the Internet for generating ASCII files with Hex data for creating simple periodic signals.

3. IMPLEMENTING DSP AUDIO ALGORITHMS

Now that some techniques for creating components of an algorithm have been proposed, let's examine some basic to moderately complex audio algorithms that are often used in prosumer equipment. Many of the DSP Techniques we will discuss can be used to implement many of the features found in digital mixing consoles and digital recorders. We will provide some example processing routines for various effects/filters that were implemented using a low cost DSP evaluation platform.

Figure 23.
Typical 8 Channel Mixer/Recorder

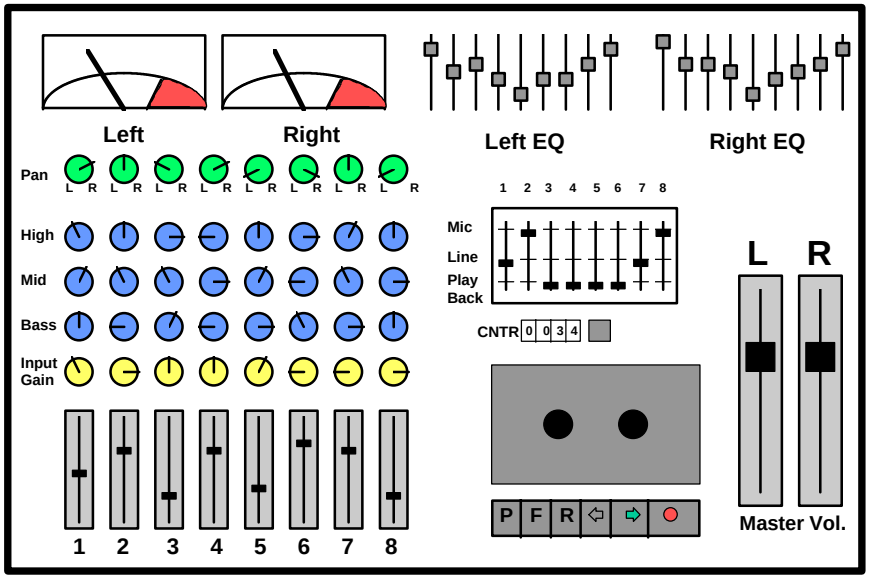


Figure 23 is an example 8 channel mixing console / recorder. Some of the features that are commonly found in mixers and multi-track recorders can be implemented with DSP instructions to perform functions that are often found in mixer equipment as the 8 track recorder shown above:

- Channel and Master Gain/Attenuation
- Mixing Multiple Channels
- Panning multiple signals to a Stereo Field
- High, Low and Mid Digital Filters Per Channel
- Graphic/Parametric Equalizers
- Signal Level Detection
- Effects Send/Return Loop for further processing of channels with signal processing equipment

Many of these audio filters and effects have been implemented using the ADSP-21065L EZ-LAB development platform, as we will demonstrate in this section with some assembly code examples.. The ability to perform all of the above functions is only constrained by the DSP MIPs. The 21056L's dual multiprocessor system capability can also be used for computationally intensive audio applications. For example, in a digital mixing console application, audio manufacturers typically will use multiple processors to split up DSP tasks or assign different processors to handle a certain number of channels. In the following section, we will model our effects and filter algorithms to cover many of the features that are found in the above digital mixer diagram, and show how easy it is to develop such a system using the ADSP-21065L EZ-LAB.

3.1 Basic Audio Signal Manipulation

The attractive alternative of choosing to use a DSP is because of the easiness at which a designer has the ability to add, multiply, and attenuate various signals, as well a filtering the signal to produce a more pleasing musical response. In this section we will review some techniques to amplify or attenuate signals, pan signals to the left and right of a stereo field, mixing multiple signals, and pre-scaling inputs to prevent overflow when mixing or filtering signals using fixed point instructions with the ADSP-21065L.

3.1.1 **Volume Control**

One of the simplest operations that can be performed in a DSP on an audio signal is volume gain and attenuation. For fixed-point math, this operation can be performed by multiplying each incoming sample by a fractional value number between 0x0000…. and 0x7FFF…. or using a shifter to multiply or divide the sample by a power of 2. When increasing the gain of a signal, the programmer must be aware of overflow, underflow, saturation, and quantization noise effects.

```
.VAR   DRY_GAIN_LEFT = 0x6AAAAAAA;      /* Gain Control for left channel */
                                        /* scale between 0x00000000 and 0x7FFFFFFF */
.VAR   DRY_GAIN_RIGHT = 0x40000000;     /* Gain Control for right channel */
                                        /* scale between 0x00000000 and 0x7FFFFFFF */

/* modify volume of left channel */
   r10 = DM(Left_Channel);              /* get current left input sample    */
   r11 = DM(DRY_GAIN_LEFT);             /* scale between 0x0 and 0x7FFFFFFF */
   r10 = r10 * r11(ssf);                /* x(n) *(G_left)             */

/* modify volume of right channel */
   r10 = DM(Right_Channel);             /* get current right input sample    */
   r11 = DM(DRY_GAIN_RIGHT);            /* scale between 0x0 and 0x7FFFFFFF */
   r10 = r10 * r11(ssf);                /* x(n) *(G_right)            */
```

3.1.2 **Mixing Multiple Audio Signal Channels**

Adding multiple audio signals with a DSP is easy to do. Instead of using op-amp adder circuits, mixing a number of signals together in a DSP is easily accomplished with an ALU's adder circuit and/or Multiply/Accumulator. First signals are multiplied by a constant number so that the signals do not overflow when added together. The easiest way to ensure signals are equally mixed is by choosing a fractional value equal to the inverse of the number of signals to be added.

For example, to mix 5 audio channels together at equal strength, the difference equation (assuming fractional fixed point math) would be:

$$\mathbf{y(n)} = \frac{1}{5}\mathbf{x}_1\mathbf{(n)} + \frac{1}{5}\mathbf{x}_2\mathbf{(n)} + \frac{1}{5}\mathbf{x}_3\mathbf{(n)} + \frac{1}{5}\mathbf{x}_4\mathbf{(n)} + \frac{1}{5}\mathbf{x}_5\mathbf{(n)}$$

The general mixing equation is:

$$\mathbf{y(n)} = \frac{1}{\mathbf{N}}\left[\mathbf{x}_1\mathbf{(n)} + \mathbf{x}_2\mathbf{(n)} + \mathbf{.....x_N(n)}\right]$$

Choosing ***N*** to equal the number of signals will guarantee that no overflow will occur if all signals were at full scale positive or negative values at a particular value of ***n***. Each signal can also be attenuated with different scaling values to provide individual volume control for each channel which compensates for differences in input signal levels:

$$\mathbf{y(n) = c_1x_1(n) + c_2x_2(n) + \ldots c_Nx_N(n)}$$

An example of mixing 5 channels with different volume adjustments can be:

$$y(n) = \frac{1}{5}x_1(n) + \frac{1}{10}x_2(n) + \frac{3}{10}x_3(n) + \frac{1}{20}x_4(n) + \frac{9}{20}x_5(n)$$

As in the equal mix equation, the sum of all of the gain coefficients should be less than 1 so no overflow would occur if this equation was implemented using fractional fixed point arithmetic. An example implementation of the above difference equation is shown below.

5-Channel Digital Mixer Example With Custom Volume Control Using The ADSP-21065L

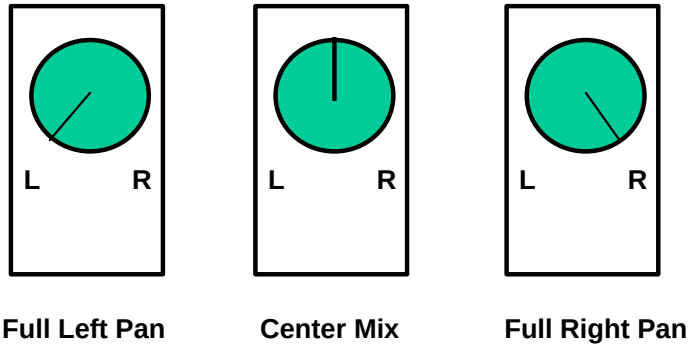
```
#define     c1          0x19999999              /* c1 = 0.2, 1.31 fract. format */
#define     c2          0x0CCCCCCC              /* c2 = 0.1 */
#define     c3          0x26666666              /* c3 = 0.3 */
#define     c4          0x06666666              /* c4 = 0.05 */
#define     c5          0x39999999              /* c5 = 0.45 */

------------------------------------------------------------------------------
/* Serial Port 0 Receive Interrupt Service Routine */

5_channel_digital_mixer:

/* get input samples from data holders */
   r1 = dm(channel_1);              {audio channel 1 input sample}
   r2 = dm(channel_2);              {audio channel 2 input sample}
   r3 = dm(channel_3);              {audio channel 2 input sample}
   r4 = dm(channel_4);              {audio channel 2 input sample}
   r5 = dm(channel_5);              {audio channel 2 input sample}

   r6 = c1;
   mrf = r6 * r1 (ssf);             {mrf = c1*x1}

   r7 = c2;
   mrf = mrf + r7 * r2 (ssf);       {mrf = c1*x1 + c2*x2}

   r8 = c3;
   mrf = mrf + r4 * r2 (ssfr);      {mrf = c1*x1 + c2*x2 + c3*x3}

   r9 = c4;
   mrf = mrf + r4 * r2 (ssfr);      {mrf = c1*x1 + c2*x2 + c3*x3 + c4*x4}

   r10 = c5;
   mrf = mrf + r4 * r2 (ssfr);      {mrf = y= c1*x1 + c2*x2 + c3*x3 + c4*x4 + c5*x5}
   mrf = sat mrf;

{----------------- write output samples to stereo D/A converter -------------------}
   r0 = mr1f;
   dm(left_output)  = r0;           {left output sample}
   dm(right_output) = r0;           {right output sample}
```

3.1.3 Amplitude Panning of Signals to a Left or Right Stereo Field

In many applications, the DSP may need to process two (or more) channels of incoming data, typically from a stereo A/D converter. Two-channel recording and playback is still the dominant method in consumer and professional audio and can be found in mixers and home audio equipment. V. Pulkki [22] demonstrated placement of a signal in a stereo field (see Figure 4 below) using Vector Base Amplitude Panning. The formulas presented in Pulkki's paper for a two-dimensional trigonometric and vector panning will be shown for reference.

Normally, the stereo signal will contain an exact duplicate of the sampled input signal, although it can be split up to represent two different mono sources. Also, the DSP can also take a mono source and create signals to be sent out to a stereo D/A converter. Typical audio mixing consoles and multichannel recorders will mix down multiple signal channels down to a stereo output field to match the standardized configuration found in many home stereo systems. Figure 25 is a representation of what a typical panning control 'pot' looks like on a mixing console or 8-track home recording device, along with some typical pan settings:

Figure 25. Three Typical Pan Control Settings of a Mono Source To A Stereo Output Field



Many 4/8/12 track analog and digital studios contain a knob to pan an input source entirely to the left or right channel, or played back through both channels at an equal mix (with the pan control centered in the middle). To give the listener a sense of location within the output stereo filed, the DSP can simply perform a multiplication of the algorithmic result on both the left and right channel so that it is perceived from coming from a phantom source.

Figure 26. Panning of Two-Channel Stereophonic Audio Derived by Blumlein, Bauer and Bernfeld [26]

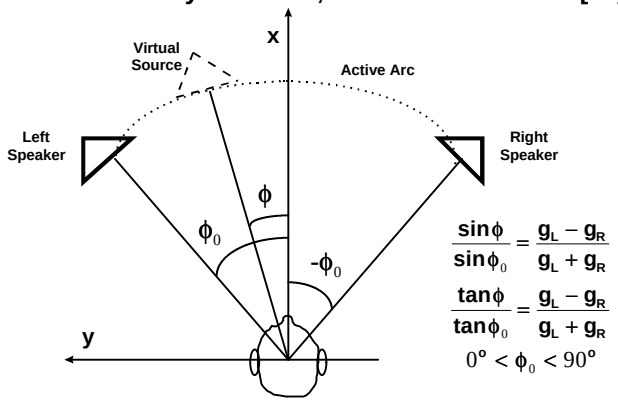
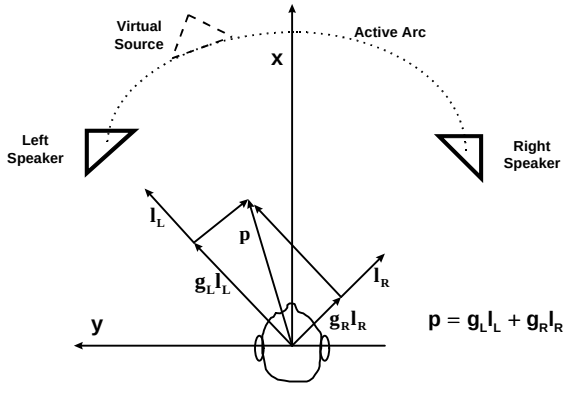


Figure 27. Pulkki's Method [26] For Vector Panning of Two-Channel Audio



$$\boxed{\begin{aligned} y_L(n) &= g_L x_L(n) \\ y_R(n) &= g_R x_R(n) \end{aligned}}$$

To create a panning effect of an audio channel to a particular position in the stereo output field, the programmer can use the Stereophonic Law of Sines, or the Tangent Law equation (Pulkki, Blumlein and Bauer[22].. see Figure 26) where g_L and g_R are the respective gains of the left and right channels.

Stereophonic Law of Sines (proposed by Blumlein and Bauer [22])

$$\frac{\sin\phi}{\sin\phi_0} = \frac{g_L - g_R}{g_L + g_R} \quad \text{where } 0^o < \phi_0 < 90^o,\ -\phi_0 < \phi < \phi_0, \text{ and } g_L, g_R \in [0,1]$$

This is valid if the listener's head is pointing straight ahead. If the listener turns the head to follow the virtual source, the Tangent Law equation as described by Pulkki [derived by Bernfeld, 26] is modified as:

$$\boxed{\frac{\tan\phi}{\tan\phi_0} = \frac{g_L - g_R}{g_L + g_R}} \quad \text{where } 0^o < \phi_0 < 90^o,\ -\phi_0 < \phi < \phi_0, \text{ and } g_L, g_R \in [0,1]$$

Assuming fixed point signed fractional arithmetic where signals are represented between 0 (0x0000…) and 0.99999 (0x7FFF…), the DSP programmer will simply multiply each signal by the calculated gain.

Using Pulkki's Vector Base Amplitude Panning method as shown in the Figure 27, the position ***p*** of the phantom sound source is calculated from the linear combination of both speaker vectors:

$$\mathbf{p} = g_L \mathbf{l}_L + g_R \mathbf{l}_R$$

The output difference I/O equations for each channel are simply:

$$y_L(n) = g_L x_L(n) \quad \text{and} \quad y_R(n) = g_R x_R(n)$$

Vector-Base Amplitude Panning Summary:

1) ***Left Pan:*** If the virtual source is panned completely to the left channel, the signal only comes out of the left channel and the right channel is zero. When the gain is 1, then the signal is simply passed through to the output channel.

$$G_L = 1$$
$$G_R = 0$$

1) ***Right Pan:*** If the virtual source is panned completely to the right channel, the signal only comes out of the right channel and the left channel is zero. When the gain is 1, then the signal is simply passed through to the output channel.

$$G_R = 1$$
$$G_L = 0$$

1) ***Center Pan:*** If the phantom source is panned to the center, the gain in both speakers are equal.

$$G_L = G_R$$

4) ***Arbitrary Virtual Positioning:*** If the phantom source is between both speakers, the tangent law applies. The resulting stereo mix that is perceived by the listener would be off-scale left/right from the center of both speakers. Some useful design equations [26] are shown below:

$$g_L = \frac{\cos\phi \sin\phi_0 + \sin\phi \cos\phi_0}{2\cos\phi_0 \sin\phi} \qquad g_R = \frac{\cos\phi \sin\phi_0 - \sin\phi \cos\phi_0}{2\cos\phi_0 \sin\phi} \qquad \phi = \arctan\left[\frac{\sin\phi_0}{\cos\phi_0}\left(\frac{g_L - g_R}{g_L + g_R}\right)\right]$$

Lookup tables can be used to determine the sine and cosine values quickly to determine the gain factors for the left and right channels if the speaker placement is known and a desired virtual angle is determined. The arctangent approximation can be computed to determine the placement of the virtual source if the speaker placement is known and gain values are arbitrarily selected. Otherwise the programmer can create a panning table with pre-computed left and right gains for a number of panning angles. Two memory storage buffers can be created, one for the left and one for the right channel. To create a virtual pan on-the-fly, the DSP can simply look up the values from memory and perform the multiplication with the output sample prior to the conversion of the sample by the D/A converter. Table 5 below shows left and right channel gains required for the desired panning angle:

Table 5. Left/Right Channel Gains for Desired Virtual Panning Placement

ϕ_0	ϕ	g_L	g_R	g_{L-norm}	g_{L-norm}
45º	45º	1	0	1	0
45º	30º	1.366	0.366	1	0.2679
45º	20º	1.8737	0.8737	1	0.4663
45º	10º	3.3356	2.3356	1	0.7002
45º	0º	1	1	1	1
45º	–10º	–2.3356	–3.3356	0.7002	1
45º	–20º	–0.8737	–1.8737	0.4663	1
45º	–30º	–0.366	–1.366	0.2679	1
45º	–45º	0	1	0	1

After nomalization of the larger gain coefficient to unity, the other channel is a fractional value between 0x0000…. and 0x7FFF…. (unity approximation). From the chart shown above we see that we can use a sine lookup table to calculate the fractional number that can be uses as a gain while the other channel is kept at unity gain (or simply passed from the input through to the desired stereo output channel). Moving through the table periodically to pan left for positive numbers and pan right for negative numbers can create a tremolo/ping-pong effect. At the zero crossing, the phase is reversed as a result of of negative pan values taken from the lookup sine table buffer. Thus, a modified version of the tremolo effect described in section 3.3.3 can be based on the vector based stereo panning concept.

30º Amplitude Panning to a Stereo Output Example (ADSP-21065L DSP)

```
{This example assumes speaker placement of 45 degrees, shown in above table}
#define     Left_Gain   0x7FFFFFFF          /* gL = 0.999999, 1.31 fract. format */
#define     Right_Gain  0x224A8C15          /* gR = 0.2679 */
-------------------------------------------------------------------------------

/* Serial Port 0 Receive Interrupt Service Routine */

Mono_To_Stereo_Panning_Controller:

   r1 = dm(audio_data);             { get audio input sample from codec }

{Note, the left channel can be passed through at unity gain, or multiplied below
with a constant close to unity, in this case 0.99999 which is an approximation of 1}

   r2 = Left_Gain;
   mrf = r2 * r1 (ssfr);            {mrf = gL * xLeft}

   r3 = Right_Gain;
   mrf = r3 * r1 (ssfr);            {mrf = gR * xRight}
```

```
{----------------- write output samples to stereo D/A converter ------------------}
   r0 = mr1f;
   dm(left_output)  = r0;          {left output sample}
   dm(right_output) = r0;          {right output sample}
```

3.2 Filtering Techniques and Applications

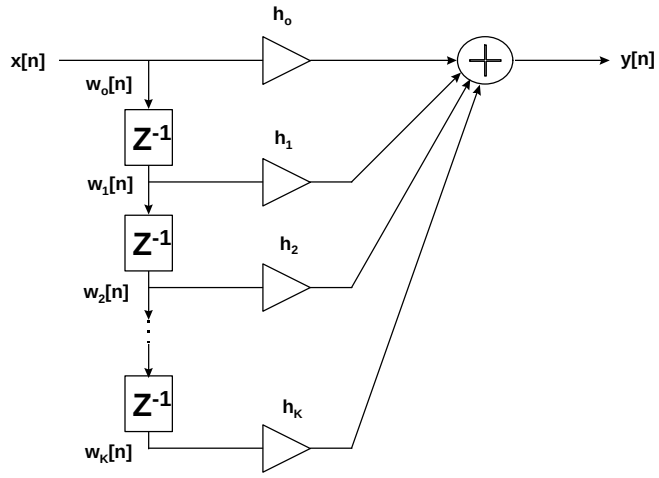
One of the most common DSP algorithms implemented in audio applications is the digital filter. Digital filters are used to increase and decrease the amplitude of a signal at certain frequencies similar to an equalizer on a stereo. These filters, just like analog filters, are generally categorized in to one of four types : high-pass, low-pass, band-pass and notch and are commonly implemented in one of two forms: the IIR (Infinite Impulse Response) filter and the FIR (Finite Impulse Response) filter. Using these two basic filter types in different configurations, we can create digital equivalents to common analog filter configurations such as parametric equalizers, graphic equalizers, and comb filters.

Digital filters work by convolving an impulse response (h[n]) with discrete, contiguous time domain samples (x[n]). The impulse response can be generated with a program like MATLAB and is commonly referred to as a set of filter coefficients. The FIR and IIR examples for the ADSP-21065L include both fixed and floating point equivalent routines.

3.2.1 **The FIR Filter**

The FIR (Finite Impulse Response) filter has an impulse response which is finite in length as implied by its name. The output values (y[n]) are calculated using previous values of x[n] as seen in the figure and difference equation below.

Figure 28.



$$y[n] = \sum_{k=0}^{\infty} h[k]x[n-k]$$

Floating Point FIR Filter Implementation on an Analog Devices' ADSP21065L

```
/* FIR Filter

Calling Parameters:
       f0 = input sample x[n]
       b0 = base address of delay line
```

```
        m0 = 1 (modify value)
        l0 = length of delay line buffer
        b8 = base address of coefficients buffer containing h[n]
        m8 = 1
        l8 = length of coefficient buffer

Return Value:
        f0 = output y[n]

Cycle Count:
        6 + number of taps + 2 cache misses
*/

FIR:    r12 = r12 xor r12, dm(i1,0) = r2;     // set r12/f12=0,store input sample in line
        r8=r8 xor r8, f0 = dm(i0,m0), f4 = pm(i8,m8); // r8=0, get data and coeffs
        lcntr = FIRLen-1, do macloop until lce; // set to loop FIR length - 1
macloop:       f12 = f0*f4, f8 = f8+f12, f0 = dm(i1,m1), f4 = pm(i8,m8); // MAC
        rts (db);                                       // delayed return from subroutine
        f12 = f0*f4, f8 = f8+f12;                       // perform last multiply
        f0=f8+f12;                                      // perform last accumulate
```

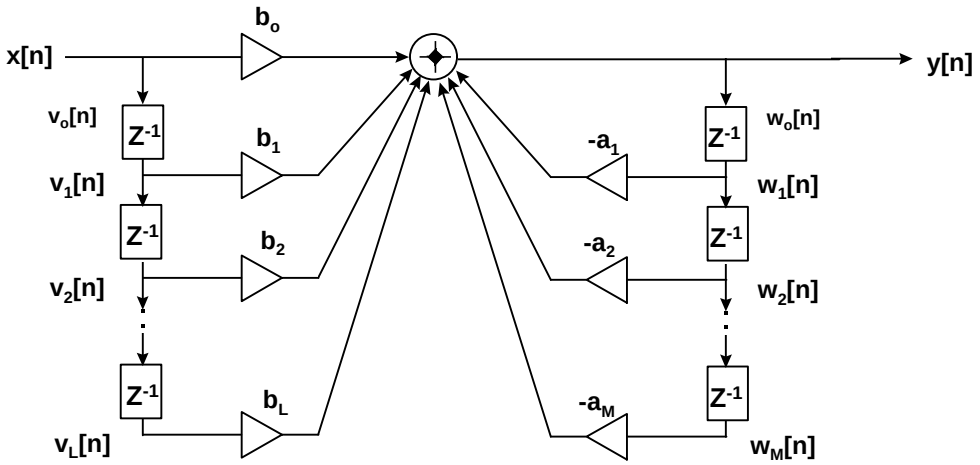
Fixed Point FIR Filter Implementation on an Analog Devices' ADSP21065L

```
/* Fixed Point FIR Filter

Calling Parameters:
        R10 = input sample x[n]
        b0 = base address of delay line
        m0 = 1 (modify value)
        l0 = length of delay line buffer
        b7 = base address of coefficients buffer containing h[n]
        i7 = pointer to coeffs buffer
        m7 = 1
        l7 = length of coefficient buffer

Return Value:
        MR1F = output y[n]
*/

fir_filter:
        M8 = 1;
        B7 = FIR_Gain_Coeffs;
        L7 = @ FIR_Gain_Coeffs;

        DM(I7,1) = R10;                         /* write current sample to buffer */

        R1 = DM(I0,1);                /* get first delay tap length */
        M7 = R1; MODIFY(I7,M7);                 /* buffer pointer now points to first tap */

        R1 = DM(I0,1);                /* get next tap length        */
        M7 = R1;
        R3 = DM(I7,M7), R4 = PM(I8,M8);         /* get first sample and first tap gain for MAC */

        LCNTR = FIRLen-1, DO er_sop UNTIL LCE;
                R1 = DM(I0,1);                          /* get next tap length        */
                M7 = R1;                                        /* put tap length in M7 */
FIR_sop:        MRF = MRF + R3*R4 (SSF),R3 = DM(I7,M7),         R4 = PM(I8,M8);
               /* compute sum of products,  get next sample,  get next tap gain */

        MRF = MRF + R3*R4 (SSFR);                               /* last sample to be computed */
        MRF = SAT MRF;
```

3.2.2 **The IIR Filter**

The IIR (Infinite Impulse Response) has an impulse response which is infinite in length. The output (y[n]) is calculated using both previous values of x[n] and previous values of y[n] as seen in the figure and difference equation below. For this reason, IIR filters are often referred to as recursive filters.

Figure 29.



$$y[n] = \sum_{i=1}^{M} \left(-a_i\, y[n-i]\right) + \sum_{j=0}^{L} \left(b_i\, x[n-j]\right)$$

Floating Point Biquad IIR Filter Implementation on an Analog Devices' ADSP21065L

```
/* BIQUAD IIR Filter

Calling Parameters
f8 = input sample x(n)
r0 = number of biquad sections
b0 = address of DELAY LINE BUFFER
b8 = address of COEFFICENT BUFFER
m1 = 1, modify value for delay line buffer
m8 = 1, modify value for coefficient buffer
l0 = 0
l1 = 0
l8 = 0

Return Values
f8 = output sample y(n)

Registers Affected
f2, f3, f4, f8, f12
i0, b1, i1, i8

Cycle Count : 6 + 4*(number of biquad sections) + 5 cache misses

# PM Locations
10 instruction words
4 * (number of biquad sections) locations for coefficents

# DM Locations
2 * (number of biquad sections) locations for the delay line
*************************************************************************/

cascaded_biquad:              /*Call this for every sample to be filtered*/
b1=b0;                 *I1 used to update delay line with new values*/
f12=f12-f12, f2=dm(i0,m1), f4=pm(i8,m8); /*set f12=0,get a2 coefficient,get w(n-2)*/
```

```
lcntr=r0, do quads until lce;
             /*execute quads loop once for ea biquad section */
       f12=f2*f4, f8=f8+f12, f3=dm(i0,m1), f4=pm(i8,m8);
            /* a2*w(n-2),x(n)+0 or y(n) for a section, get w(n-1), get a1*/
       f12=f3*f4, f8=f8+f12, dm(i1,m1)=f3, f4=pm(i8,m8);
            /*a1*w(n-1), x(n)+[a2*w(n-2)], store new w(n-2), get b2*/
       f12=f2*f4, f8=f8+f12, f2=dm(i0,m1), f4=pm(i8,m8);
             /*b2*w(n-2), new w(n), get w(n-2) for next section, get b1*/
quads: f12=f3*f4, f8=f8+f12, dm(i1,m1)=f8, f4=pm(i8,m8);
       /*b1*w(n-1), w(n)+[b2*w(n-1)], store new w(n-1), get a2 for next
```

Fixed-Point Direct-Form-I IIR Filter Implementation on an Analog Devices' ADSP21065L

```
/* ************************************************************************************
      Direct-Form-I IIR filter of order 2 using hardware circular buffers
      32-bit fixed-point arithmetic, assuming fractional 1.31 format

      The input may need to be scaled down further to avoid overflows, and the delay-line
      pointer i2 is updated by a -1 decrement

      The filter coefficients must be stored consecutively in the order:

            [a0, a1, a2,..., aM, b0, b1,..., bM]

      and i8 is points to this double-length buffer. The a,b coefficients used in the program
      are related to the true a,b coefficients by the scale factors, defined by the
      exponents ea, eb:

            a = a_true / Ga,     Ga = 2^exp_a = scale factor
            b = b_true / Gb,     Gb = 2^exp_b = scale factor

      (because a0_true = 1, it follows that a0 = 1/Ga. This coefficient is redundant and not
      really used in the computation; it always gets multiplied by zero.)

      The common double-length circular buffer I8 should be declared as:

            .var  a[M+1], b[M+1];  <-- PM variables
            B8 = a; L8 = 2*(M+1);

      Program assumes that both numerator and denominator have order M. The y- and x-delay-line
      buffers must be declared as follows:

            .var w[M+1];  <-- DM variables
            .var v[M+1];  <--
            B2 = w; L2 = @w;
            B3 = v; L3 = @v;

************************************************************************************** */

.GLOBAL        IIR_filter;
.GLOBAL        init_IIR_filter_buffers;

#define exp_a   1                    /* scaling exponent for a - divide by 2 */
#define exp_b   1                    /* scaling exponent for b  - scale by 2 */

/* filter coeff in 2.30 fractional format */
#define b2     0x3ec474bf            /* b2  = 0.980740725, scaled by 2 */
#define b0     0x3ec474bf            /* b0  = 0.980740725, scaled by 2 */
#define a1_b1  0x82771682            /* a1 and b1 = -2*b0 = -1.96148145, scaled by 2 */
#define a2     0x3d88e97e            /* a2  = 2*b0 - 1 = 0.96148145, scaled by 2 */
#define a0     0x40000000            /* a0  = 1, scaled by 2 */


/* ------------- DATA MEMORY FILTER BUFFERS ---------------------------------*/
.segment /dm   dm_IIR;

.var           w[3];                 /* y-delay-line buffer in DM */
.var           v[3];                 /* x-delay-line buffer in DM */
```

```
.endseg;

/* -------------PROGRAM MEMORY FILTER BUFFERS -------------------------------*/
.segment /pm    pm_IIR;

.var           a[3] = a0, 0, a2;      /* a coeffs in PM, initial denominator coefficients */
.var           b[3] = b0, 0, b2;      /* b coeffs in PM, initial numerator coefficients */

.endseg;



.segment /pm pm_code;


/* -------------------  IIR Digital Filter Delay Line Initialization  ------------------- */

init_IIR_filter_buffers:

       B2 = w; L2 = @w;                    /* y-delay-line buffer pointer and length */
       B3 = v; L3 = @v;                    /* x-delay-line buffer pointer and length */
       B8 = a; L8 = 6;                     /* double-length a,b coefficients */
       m2 = 1;
       m3 = 1;

       LCNTR = L2;                         /* clear y-delay line buffer to zero */
        DO clr_y_Dline UNTIL LCE;
clr_y_Dline:         dm(i2, m2) = 0;

       LCNTR = L3;                         /* clear x-delay line buffer to zero */
        DO clr_x_Dline UNTIL LCE;
clr_x_Dline:         dm(i3, m3) = 0;

       call init_wavetable;

       RTS;

/*************************************************************************************************/
/*                                                                                               */
/*                          IIR Digital Filter Routine - Direct Form 1                           */
/*                                                                                               */
/*      Input Sample x(n) = R15                                                                  */
/*      Filtered Result y(n) = R9                                                                */
/*                                                                                               */
/*************************************************************************************************/

IIR_filter:
       /*r15 = scaled input sample x, put input sample into tap-0 of x delay line w[] */
       dm(i3, 0) = r15;

       /*put zero into tap-0 of y delay line v[], where s0 = 0*/
       r8 = 0;  dm(i2, 0) = r8;   /* because a0_true = 1, it follows that a0 = 1/Ga.
                                     This coefficient is redundant and not really used in the
                                     computation; it always gets multiplied by zero. */

       m8 = 1; m2 = 1; m3 = 1;

       /*dot product of y delay line buffer w[3] with a-coeffs of length 2 + 1*/
       mrf = 0, r0 = dm(i2, m2), r1 = pm(i8, m8);
       LCNTR = 2;
       DO pole_loop UNTIL LCE;
pole_loop:             mrf = mrf + r0 * r1 (SSF), r0 = dm(i2, m2), r1 = pm(i8, m8);
       mrf = mrf + r0 * r1 (SSFR);
       mrf = SAT mrf;
       r3 = mr1f;
       r12 = ashift r3 by exp_a;                          {Ga * dot product(2nd order a coeff)}

       /*dot product of x delay line buffer v[3] with b-coeffs of length 2 + 1*/
       mrf = 0, r0 = dm(i3, m3), r1 = pm(i8, m8);
```

```
        LCNTR = 2;
        DO zero_loop UNTIL LCE;
zero_loop:              mrf = mrf + r0 * r1 (SSF), r0 = dm(i3, m3), r1 = pm(i8, m8);
        mrf = mrf + r0 * r1 (SSFR);
        mrf = SAT mrf;
        r8 = mr1f;
        r13 = ashift r8 by exp_b;                              {Gb * dot product(2nd order b coeff)}

        /*compute output y, where y(n) = b's * x's - a's * y's  */
        r9 = r13 - r12;                                                 /* output y in r9 */

        /*put output sample into tap-0 of y delay line w[] */       /*  s0 = y  */
        /* and backshift circular y[ ] delay-line buffer pointer */
        dm(i2, -1) = r9;

        /*backshift pointer & update circular x[ ]delay-line buffer, output in r9*/
        modify(i3, -1);

        rts;
.endseg;
```

3.2.3 **Parametric Filters**

Parametric filters are used quite widely in the audio industry because of their ability to amplify or dampen specific frequency components of a signal. Traditionally, these filters have been designed using analog components, however, their digital counterparts are very simple and efficient to implement. The filter's bandwidth, center frequency and gain can be calculated using a few basic formulas to calculate the four required coefficients.

Parametric filters are really just second order IIR filters and have a frequency response containing a single peak or notch at a given frequency ϖ_o. The gain at all other frequencies is roughly unity. These filters require less computational power than higher order FIR and IIR filters and the amount of computational power required to calculate coefficients is minimal compared to higher order filters.

Here's how the filter works : a conjugate pair of poles and a conjugate pair of zeros are arranged along a straight line from the origin of the Z-plane as shown in figure x. If the poles are closer to the origin than the zeros (i.e. R < r), the resulting filter will be a notch filter. On the other hand, the zeros are closer to the origin, the resulting filter will be a peak filter. Figure y contains the frequency responses of both cases. The strength of the boost or cut of the respective peak and notch is determined by the closeness of r and R. Also, the width of the peak or cut is determined by the closeness or r and R to the unit circle.

Figure 30.

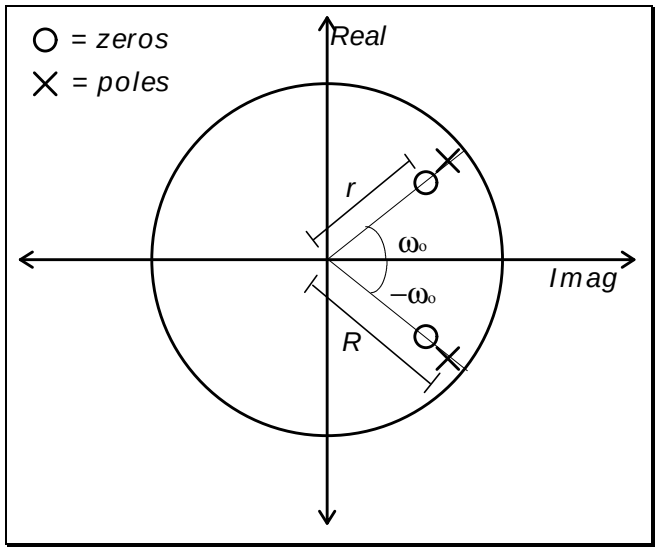


Figure 31.

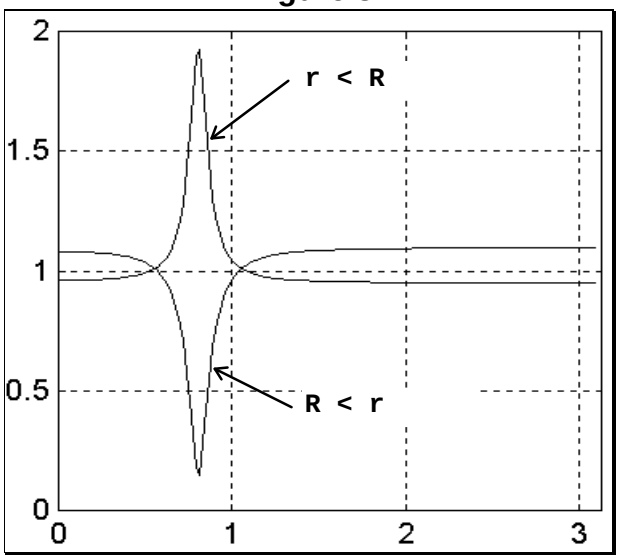


Figure 1 : Pole-Zero Plot of parametric Filter *Figure 2 : Frequency response of parametric filter*

Below are the transfer function and difference equation for the parametric filter.

$$\mathbf{H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 - a_1 z^{-1} - a_2 z^{-2}}} \qquad \mathbf{y(n) = b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) + a_1 y(n-1) + a_2 y(n-2)}$$

This difference equation is implemented in code example below. The following equations (Orphanidis,1996) are used to calculate the coefficient values based on a known value of ϖ_o, r and R:

$$a_0 = 1 \qquad a_1 = -2R\cos(w_o) \qquad a_2 = R^2$$
$$b_0 = 1 \qquad b_1 = -2r\cos(w_o) \qquad b_2 = r^2$$

These calculations are implemented in code example **YYYYY.**

Parametric Filter Implementation on an Analog Devices' ADSP-21065L

```
/*
Parametric Filter Implementation

inputs:
f0 = input sample
i0 = pointer to x[] values - circular buffer 3 elements long
i1 = pointer to y[] values - circular buffer 3 elements long
i8 = pointer to coefficients

outputs:
f1 = output sample
*/
Filter:
      r12 = r12 xor r12, dm(i0,m1) = f0;
      f8  = f0+f3, f0 = dm(i1,m1), f4 = pm(i8,m8);
      f12 = f0*f4, f8 = f8+f12, f0 = dm(i1,m1), f4 = pm(i8,m8);
      f12 = f0*f4, f8 = f8-f12, f0 = dm(i0,m1), f4 = pm(i8,m8);
      f12 = f0*f4, f8 = f8-f12, f0 = dm(i0,m0), f4 = pm(i8,m8);
      f12 = f0*f4, f8 = f8+f12;
      f1 = f8 + f12;
      rts;
```

Parametric Filter Coefficient Calculations Implementation on an Analog Devices' ADSP21065L

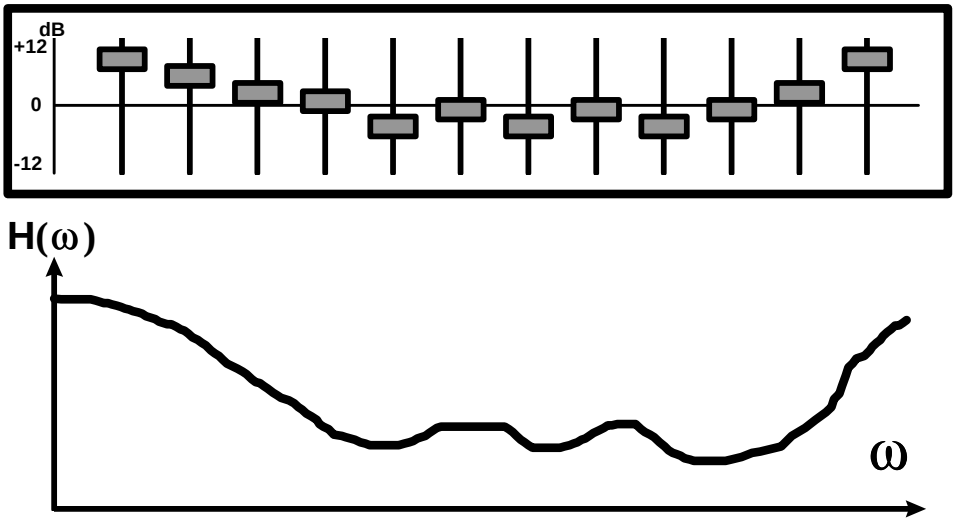
```
/*
Parametric Filter Coefficient Calculations

inputs:
      f0 = v_op     v_o = frequency
      f1 = R
      f3 = r
      i9 = pointer to where coefficients are to be stored
outputs:
      a1, a2, b1, b2 coefficients
```

```
*/
ParametricCalc:
      call cosine (db);       // returns f0 as the cos(f0)
      nop;
      f4 = -2.0;
      f0 = f0 * f4;           /* -2*cos(v_o)   */
      f4 = f0 * f3;           /* f4=-2r*cos(v_o) */
      pm(2,i9)=f4;            /* store b1 */
      f4 = f0 * f1;           /* f4=-2R*cos(v_o) */
      pm(0,i9)=f4;            /* store a1 */
      f3=f3*f3;
      pm(3,i9)=f3;            /* store b2 */
      rts (db);
      f1=f1*f1;
      pm(1,i9)=f1;            /* store a2 */
      rts;
```

3.2.4 Graphic Equalizers

Professional and Consumer use equalizers to adjust the amplitude of a signal within selected frequency ranges. In a Graphic Equalizer, the frequency spectrum is broken up into several bands using band-pass filters. Setting the different gain sliders to a desired setting gives a 'visual graph' (Figure 32) of the overall frequency response of the equalizer unit. The more bands in the implementation yields a more accurate desired response.

Figure 32. Typical 12 Band Analog Graphic Equalizer

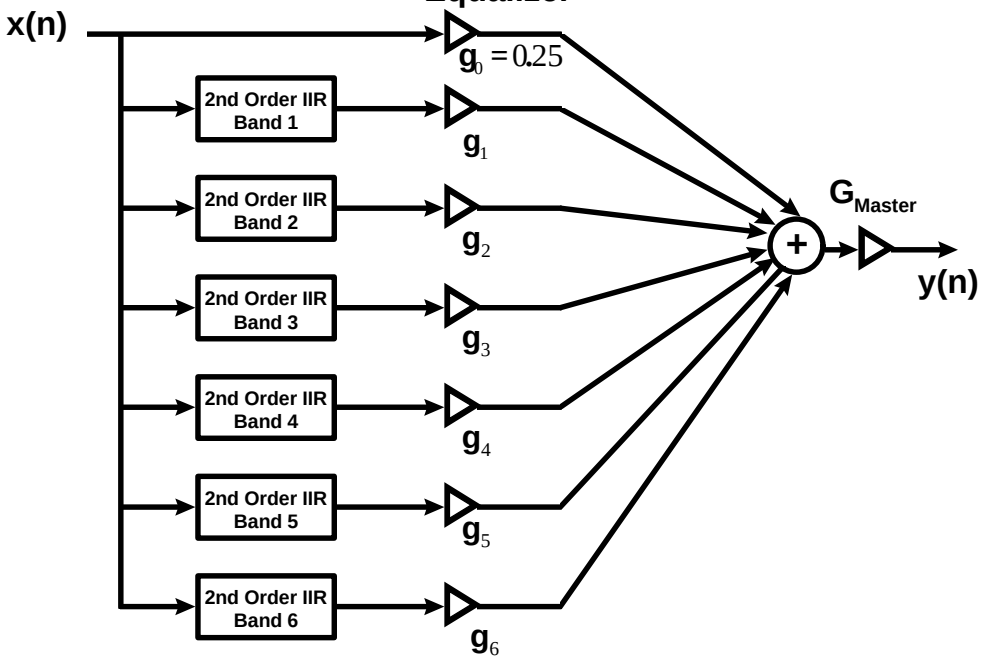


Analog equalizers typically uses passive and active components. Increasing the number of bands results in a large board design. When implementing the same system in a DSP, however, the number of bands is only limited by the speed of the DSP (MIPs) while board space remains the same. Resisters and capacitors are replaced by discrete-time filter coefficients, which are stored in a memory and can be easily modified.

Figure 33 shows and example DSP structure for implementing a 6 band graphic equalizer using second order IIR filters. The feedforward path is a fixed gain of 0.25, while each filter band can be multiplied by a variable gain for gain/attenuation. There are many methods of implementation for the second order filter, such as using ladder structures or biquad filters. Filter coefficients can be generated by a commercially available filter design package, where A and B coefficients can be generated in for the following 2nd order transfer function and equivalent I/O difference equations:

$$\mathbf{H(z)} = \frac{\mathbf{Y(z)}}{\mathbf{X(z)}} = \frac{\mathbf{b}_0 + \mathbf{b}_1\mathbf{z}^{-1} + \mathbf{b}_2\mathbf{z}^{-2}}{1 - \mathbf{a}_1\mathbf{z}^{-1} - \mathbf{a}_2\mathbf{z}^{-2}} \qquad \mathbf{y(n)} = \mathbf{b}_0\mathbf{x(n)} + \mathbf{b}_1\mathbf{x(n-1)} + \mathbf{b}_2\mathbf{x(n-2)} + \mathbf{a}_1\mathbf{y(n-1)} + \mathbf{a}_2\mathbf{y(n-2)}$$

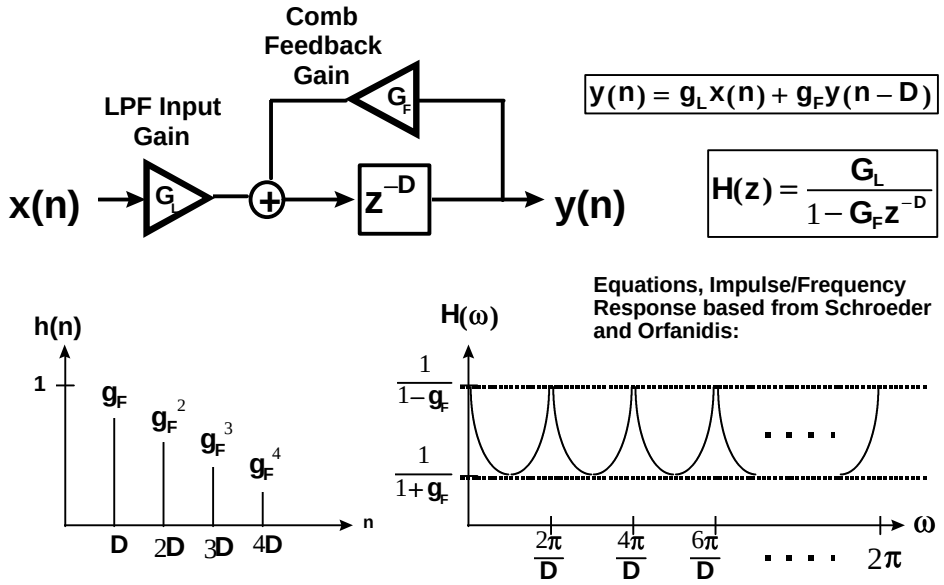
Figure 33. DSP Implementation of a 6-Band Digital Graphic Equalizer



3.2.5 Comb Filters

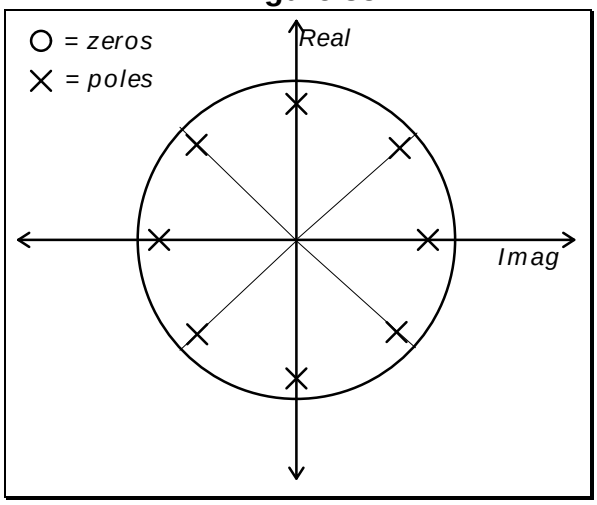
Comb filters are used for noise reduction of periodic signals, signal enhancement, averaging signals, and are inherent in digital audio effects such as delays, chorus and flange effects. Comb filters work by adding a signal with a delayed and scaled version of the same signal. This causes some frequencies to become attenuated and others to become amplified through signal addition and subtraction. Comb filters essentially simulate multiple reflections of sound waves, as we will see in the section on digital reverb. As the name implies, the frequency response looks like the teeth of a comb as seen in the figure below. Orfanidis [2] covers Comb Filters and their applications in much detail and serves as an excellent reference. In his text [2], he shows the derivation of the IIR and FIR Digital Comb Filters with the equations shown below:

Figure 34.
IIR Comb Filter



Below is the corresponding pole-zero diagram. The distance from the poles to the origin is determined by the value of G_F. The number of poles is equal to the length of the delay element, or the value of D. In the case below, D = 8 and G_F equals about 0.9.

Figure 35.



Orfanidis [2] covers Comb Filters and their applications in much detail and serves as an excellent reference. In his text [2], he shows the derivation of the IIR and FIR Digital Comb Filters with the following equations:

$$\mathbf{H_{IIRcomb}(z) = b\frac{1+z^{-D}}{1-az^{-D}}}, \quad \mathbf{H_{FIRcomb}(z) = \frac{1}{N}(1+z^{-D}+z^{-2D}+\ldots.z^{-(N-1)D}) = \frac{1}{N}\frac{1-z^{-ND}}{1-z^{-D}}}$$

For the FIR version, it can be used to adding a delayed versions of an input signal to produces delay effects.

And, in a simplified case for **D=1**, we have an FIR smoother, or Moving Average Filter:

$$\mathbf{H_{FIR\_MA}(z) = \frac{1}{N}(1 + z^{-1} + z^{-2} + \ldots . z^{-(N-1)}) = \frac{1}{N}\frac{1 - z^{-N}}{1 - z^{-1}}}$$

Recursive Comb Filters are can be used for constructing Reverb Algorithms [2]:

$$\mathbf{H_{COMB}(z) = \frac{1}{1 - az^{-D}}}$$

These make excellent building blocks because the input signal is repeated at multiples of **D**.

21065L IIR-Comb Filter Example

```
/*****************************************************************************************
        Transfer function and I/O difference equations:
                    1
         H(z) = -----------          y(n) = ay(n - D) + x(n)
                1 - az^(-D)


        IIR Comb Filter Structure:

         x(n) --------->O------------------------------> y(n)
                        ^ +                    |
                        |                   ____|____
                        |                  |         |
                        |                  | z^(-D)  |
                        |                  |_________|
                        |                       |
                        |          a            |
                        |---------<|-----------|



***************************************************************************************** */

/* ADSP-21065L System Register bit definitions */
#include       "def21065l.h"
#include       "new65Ldefs.h"

.GLOBAL        Init_Comb_Filter_Buffer;
.GLOBAL        IIR_comb_filter;
.GLOBAL        change_reverb_settings;
.EXTERN        Left_Channel;
.EXTERN        Right_Channel;

/* ------------- DATA MEMORY FILTER BUFFERS ---------------------------------*/
.segment /dm   dm_revrb;

#define        D      18000           /* TD = D/fs = 18000/48000 = 0.375 sec */

.var           IRQ1_counter = 0x00000004;
.var           Decay_Time = D + 1;

/*  define comb filter variable, and buffer */
.var           a = 0x40000000;        /* comb feedback coefficient a = 0.50 */
.var           w[D + 1];              /* left delay-line buffer */

.endseg;

/* -------------------------------- PROGRAM MEMORY CODE ------------------------------------ */

.segment /pm pm_code;

Init_Comb_Filter_Buffer:

       /* Initialize comb filter buffer pointer */
```

```
        B2 = w; L2 = @w;              /* delay-line buffer pointer and length */
        M2 = 1;

        /* clear delay line buffers to zero */
        LCNTR = L2;
         DO clrDline_1 UNTIL LCE;
clrDline_1:    dm(i2, m2) = 0;

        RTS;

/*****************************************************************************************/
/*                                                                                       */
/*                                  IIR COMB FILTER                                      */
/*                                                                                       */
/*      Can be used as a building block for a more complex stereo reverb response        */
/*      This example processes the left channel and creates a stereo response            */
/*                                                                                       */
/*****************************************************************************************/

IIR_comb_filter:

        /* get input samples from data holders */
        r0 = dm(Left_Channel);               /* left input sample */
        r1 = dm(Right_Channel);              /* right input sample */

        /* process left channel */
        mrf = 0; mr1f = r0;                   /* mrf = x = left input sample */

        r2 = dm(a);

        /* tap outputs of circular delay line, where r2 = sD = D-th tap */
        m2 =  D;  modify(i2, m2);            /* point to d-th tap */
        m2 = -D;  r3 = dm(i2, m2);           /* put d-th tap in data register */

        mrf = mrf + r2 * r3 (ssfr);          /* mrf = y = xL + a * sD = output */
        r12 = mr1f;

        /* put y result into tap-0 of delay line */ ;
        r12 = mr1f; dm(i2, 0) = r12;

        /* backshift pointer & update delay-line buffer */
        modify(i2, -1);

        /* send comb filter result to left/right channel outputs */
        dm(Left_Channel) = r12;              /* left output sample */
        dm(Right_Channel) = r12;             /* right output sample */

        rts;
```

3.2.6 **Scaling to Prevent Overflow**

Overflow/underflow is a hardware limitation that occurs when the numerical result of the fixed point computation exceeds the largest or smallest number that can be represented by the DSP. To prevent overflow, the input signal needs to be properly scaled down (attenuated) before it is passed through the digital filter routine.

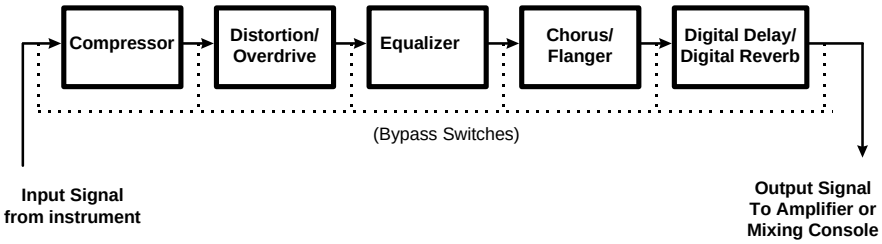
In addition to preventing fixed point overflow when doing simple addition of signals, scaling of input samples when performing fixed point digital filtering, or when going between consecutive filter stages of a higher order filter implementation.

The disadvantage to scaling is the dynamic range of the filter is reduced. Precision in the lower bits can be lost if the downshifted scaling factor causes precision in the LSBs to be lost. The SNR of the final result is lower, since the output would have to be scaled back up, introducing quantization noise into the output converter with ‘zeros’ introduced in the lower LSBs. To prevent this, a DSP data word width would have to be much larger than the precision of the A/D and D/A converters, or double precision math can be used. The high dynamic range capability of 32 or 40-bit floating point processing that the ADSP-21065L offers can virtually eliminate the need for scaling input samples to prevent overflow.

3.3 Time-Delay Digital Audio Effects

In this section some background theory and basic implementation of a variety of time-based digital audio effects will be examined. The figure below shows some algorithms that can be found in digital audio effects processor. We will first look at implementing signal and multiple reflection delay effects using delay lines, and then discuss more intricate effects such as chorusing, flanging, pitch shifting and reverberation. These effects are found in many commercial audio effects processors that use a DSP to process real time audio signals (Figure 36).

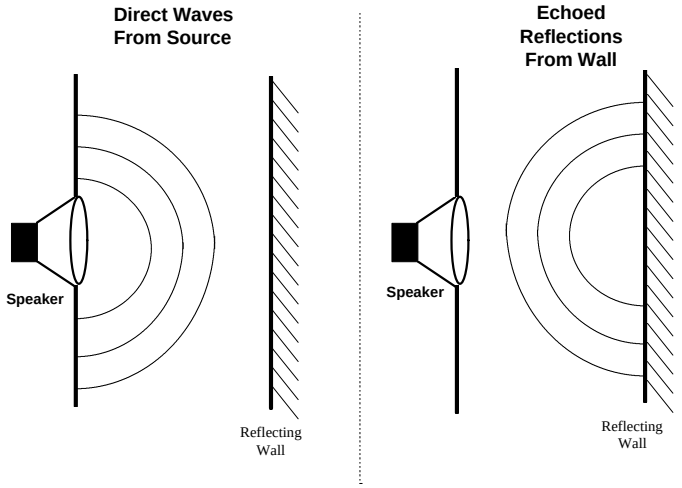
Figure 36.
Typical Signal Chain for Audio Multi-Effects Processors



3.3.1 Digital Delay - (Echo, Single Delay, Multi-tap Delays and ADT)

The Digital Delay is the simplest of all *time delay* audio effects. The delay effect is often the basis to produce more intricate effects such as flanging and chorusing, which vary the delay time on-the-fly. It is also used in reverberation algorithms to produce early reflections and recursive delays.

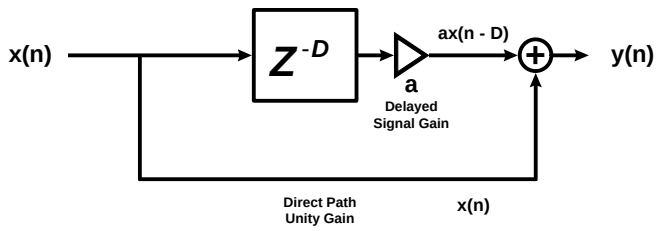
Figure 37. Echo Between The Source And Reflecting Wall



When a sound source is reflected from a distant, hard surface, a delayed version of the original signal is heard at a later time (see Figure 37). Before the introduction of DSPs in audio, the first delay units were created by using tape delay with multiple moving recording heads, while other units then produced the delay with analog circuitry. To recreate this reflection digitally, DSP delay effects units encode the input signal and store it digitally in a delay-line buffer until it is required at the later time where it is decoded back to analog form [17]. The DSP can produce delays in a variety of ways. Delay units can produce stereo results and multiple-tapped delayed results [7]. Many effects processors implement a delay and use it as a basis for

producing multi-tap and reverb effects. Multi-tapped signals can be panned to the right, left or mixed together to give the listener the impression of the stereo echo bouncing from one side to the other.

Figure 38. Implementation of a Digital Delay with a Single Tap



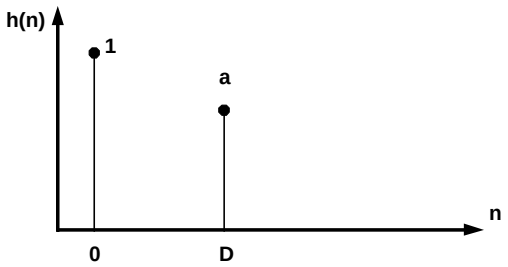


Figure 39. Delay (or Echo) with a Single Reflection

Single Refection Delay

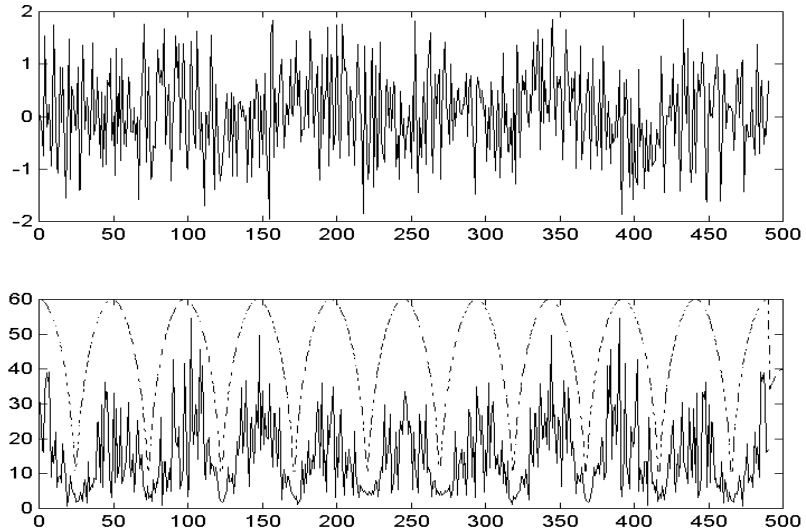
To create a *single reflection* (Figures 38 and 39) of an input signal, the implementation shown above is represented in the following difference equation [2]:

$$\boxed{y(n) = x(n) + ax(n-D)}$$

and it's transfer function is: $H(z) = 1 + az^{-D}$

Notice that the input x(n) is added to a delayed copy of the input. The signal can be attenuated by a factor that is less that 1, because reflecting surfaces, as well as air, contain a loss constant ***a*** due to absorption of the energy of the source wave. The delay ***D*** represents the total time it takes for the signal to return from a reflecting wall. ***D*** is created by using a delay-line buffer of a specified length in DSP memory. The frequency response results in a FIR comb filter [2] where peaks in the frequency response occur at multiples of the fundamental frequency. Comb filters (see section 3.2) result whenever a direct input signal is combined with delayed copies of the direct input (see Figure 40 for an example response).

Figure 40.
Example FIR Comb Filter Result of Adding An Input Signal To a Delayed Replica



The DSP can subtract the delay instead of adding it:

$$y(n) = x(n) - ax(n - D)$$

An example implementation for adding an input to a delayed replica is:

$$y(n) = \frac{1}{2}x(n) + \frac{1}{2}x(n-D)$$

Single Reflection Digital Delay Processing Routine Implemented on the ADSP-21065L

```
#define           a0          0x40000000        /* a0 = 0.50 */
#define           a1          0x40000000        /* a1 = 0.50 */
#define           DelayLine         2000        /* TD = D/fs=2000/8000 Hz= 25msec */

Digital_Delay_Effect:               /* process right channel input only */

/* get input samples from data holders */
   r0 = dm(left_input);             {left input sample}
   r1 = dm(right_input);            {right input sample}

/* tap output of circular delay line */
   m2 =  - DelayLine;               {load delay line tap value }
   r3 = dm(m2, i2);                 {point to d-th tap and put in data register}
                                    {pre-modify address with no update}

   r2 = a0;
   mrf = r1 * r2 (ssf);             {mrf = a0 * x}
   r2 = a1;
   mrf = mrf + r3 * r2 (ssf);       {mrf = a0 * x + a1 * sDelay}
   mrf = sat mrf;

{--- write output samples to codec ------------------------------------}
   r10 = mr1f;
   dm(left_output)  = r10;          {left output sample}
   dm(right_output) = r10;          {right output sample}

/* put input sample into tap-0 of delay line, post-modify address after storage */
   m2 = 1;
   dm(i2, m2) = r0;                 {put value from register r0 into delay line}
                                    {and increment address by 1}

   rti;                             { Return from Interrupt }
```

Automatic Double Tracking (ADT) and Slapback Echo

One popular use of the digital delay is to quickly repeat the input signal with a single reflection at unity gain. By making the delay an input signal around 15-40 milliseconds, the resulting output produces a "*slapback*" or "*doubling*" effect (see Figure 41). The slight differences in the delay create the effect of the two parts being played in unison. This effect can also be set up to playback the original "dry" signal in one stereo channel and the delayed signal in the other channel (Figure 42). This creates the impression of a stereo effect using a single mono source. The same technique is used for a mono result, except both signals are added together. With short delays, slapback can thicken the sound of an instrument or voice when mixed for a mono result, although cancellations can occur from comb filtering side effects when the delay is under 10 ms, which will result in a hollow, resonant sound [2], [26].

Figure 41.
Slapback Echo Effect

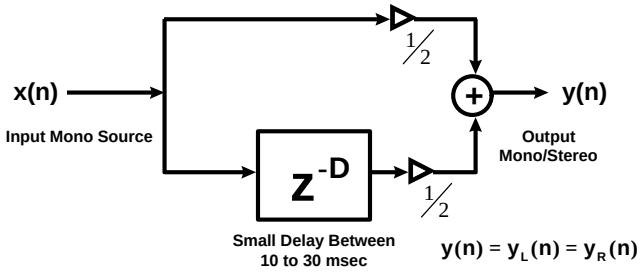
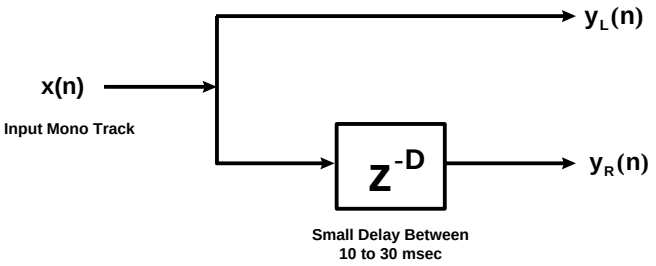


Figure 42.
Automatic Double Tracking / 'Stereo Doubling'



Example Slapback and Stereo Doubling Routines for the ADSP-21065L

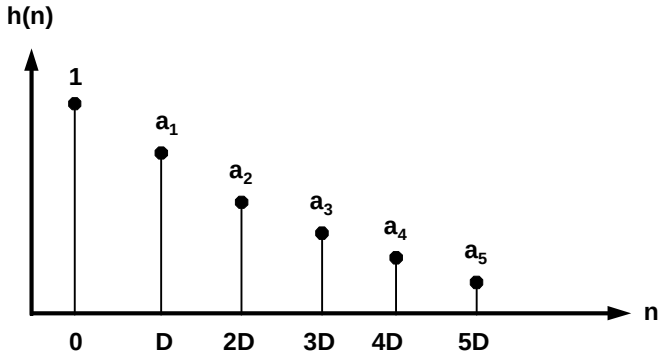
```
/* ------------------------------------------------------------------------------------- */
/*     Slapback Echo - Mono Doubling Audio Effect (ADT) using a digital delay-line       */
/*     Digital Delay Effect to create a mono echo effect                                 */
/*                                                                                       */
/*     This routines scales & mixes both input channels into 1 audio stream              */
/* ------------------------------------------------------------------------------------- */

Slapback_Echo:                          /* process both channel inputs */

       /* get input samples from data holders */
       r0 = dm(Left_Channel);           /* left input sample */
       r1 = dm(Right_Channel);          /* right input sample */

       r1 = ashift r1 by -1;            /* scale signal by 1/2 for equal mix */
       r2 = ashift r2 by -1;            /* scale signal by 1/2 for equal mix */
       r1 = r2 + r1;                    /* 1/2xL(n) + 1/2 xR(n) */


       L6 = dm(delay_time);
       /* tap output of circular delay line */
       r3 = dm(i6, 0);                  /* point to d-th tap and put in data register */
                                        /* fetch address with no update */

       /* add delayed signal together with original signal */
        r1 = ashift r1 by -1;           /* scale input signal by 1/2 for equal mix */
       r3 = ashift r3 by -1;            /* scale delayed signal by 1/2 for equal mix */
       r4 = r3 + r1;                    /* 1/2xL(n) + 1/2 xR(n) */

       /* write output samples to AD1819 Master Codec channels */
       r4 = ashift r4 by 1;             /* turn up the volume a little */
       dm(Left_Channel)  = r4;          /* left output sample */
       dm(Right_Channel) = r4;          /* right output sample */

       /* put input sample into tap-0 of delay line, post-modify address after storage of input */
       dm(i6, -1) = r1;                 /* put value from register r1 into delay line */
                                        /* and decrement address by -1 */

       rts;                             /* Return from Subroutine */



/* ------------------------------------------------------------------------------------- */
/*     Stereo Automatic Double Tracking - ADT Audio Effect using a digital delay-line    */
/*     Digital Delay Effect to create a stereo field effect                              */
/*     Also called 'Stereo Doubling'                                                     */
/* ------------------------------------------------------------------------------------- */

Stereo_Double_Tracking:                 /* process right channel input only */

       /* get input samples from data holders */
       r0 = dm(Left_Channel);           /* left input sample */
       r1 = dm(Right_Channel);          /* right input sample */

       L6 = dm(delay_time);
```

```
/* tap output of circular delay line */
r3 = dm(i6, 0);                     /* point to d-th tap and put in data register */
                                    /* fetch address with no update */

/* write output samples to AD1819 Master Codec channels */
dm(Left_Channel)  = r0;             /* left output sample */
dm(Right_Channel) = r3;             /* right output sample */

/* put input sample into tap-0 of delay line, post-modify address after storage of input */
dm(i6, -1) = r1;                    /* put value from register r1 into delay line */
                                    /* and decrement address by -1 */

rts;                                /* Return from Subroutine */
```

Multitap Delays

Multiple delayed values of an input signal can be combined easily to produce multiple reflections of the input. This can be done by having multiple taps pointing to different previous inputs stored into the delay line, or by having separate memory buffers at different sizes where input samples are stored.

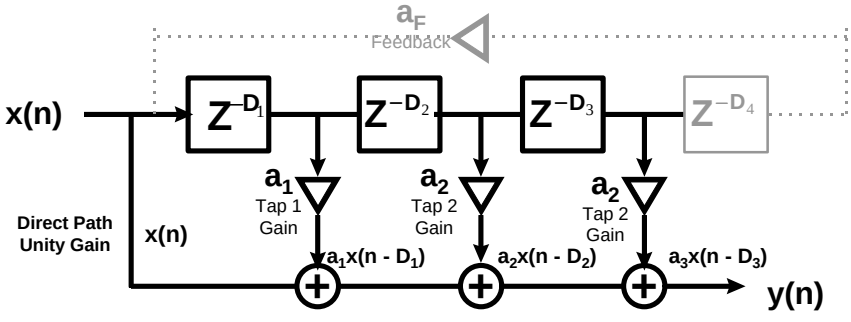
Figure 43.
Typical Impulse Response of Multiple Delay Effect



The difference equation is a simple modification of the single delay case. To 5 delays of the input (see Figure 43), the DSP processing algorithm would perform the following difference equation operation:

$$y(n) = x(n) + a_1x(n - D1) + a_2x(n - D2) + a_3x(n - D3) + a_4x(n - D4) + a_5x(n - D5)$$

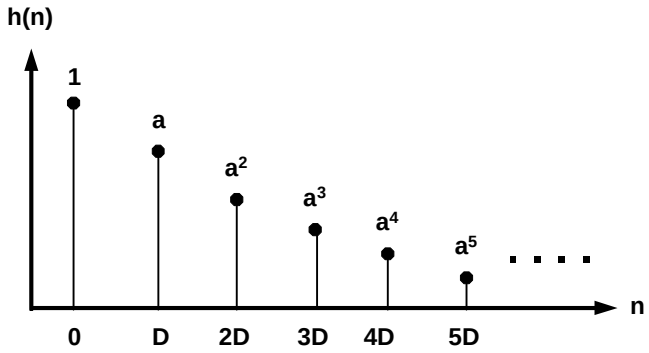
Figure 44. Multiple Delay (3-Tap) Example



$$y(n) = x(n) + a_1x(n - D1) + a_2x(n - D2) + a_3x(n - D3)$$

The above structure uses 3 delay-line tap points for fetching samples. In addition, feedback can be used to take the output of the system delay and feed it back to the input.

Figure 45.
Impulse Response of Multiple Delays Decaying Exponentially



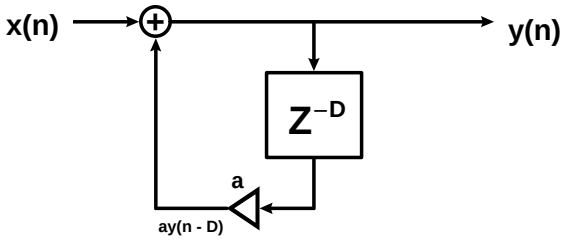


Figure 46.

Adding an infinite # of delays will create a rudimentary reverb effect by simulating reflections in a room (Figure 46). The difference equation then becomes and IIR comb filter (Figure 46):

$$y(n) = x(n) + ax(n - D) * a^2x(n - 2D) + a^3x(n - 3D) + \ldots\ldots$$

The transfer function is then:

$$H(z) = 1 + az^{-D} + a^2z^{-2D} + a^3z^{-3D} + \ldots$$

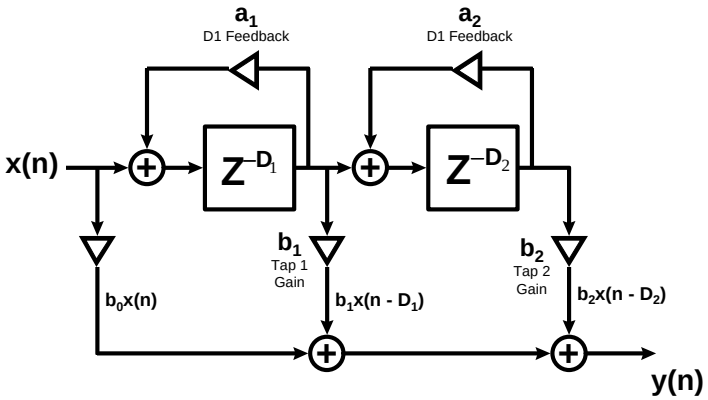
Represented as a geometric series, the transfer function becomes a recursive IIR comb filter:

$$H(z) = \frac{1}{1 - az^{-D}} \qquad y(n) = ay(n - D) + x(n)$$

Since an infinite number of delays are created as the response becomes and IIR response, this computationally simple IIR comb filter is used as the basis for constructing more intricate reverberation effects.

Figure 47 is another variation of implementing multiple delays using 2 delay lines. Feedback is introduced to produced more reflections.

Figure 47.
2 Tap Multi-delay Effect Implementation Described by Orfanidis [Intro. to Signal Processing]



The Z-transform for the above equation described by Orfanidis [2] is:

$$H(z) = b0 + b1\left[\frac{z^{-D1}}{1 - a_1 z^{-D1}}\right] + b2\left[\frac{z^{-D1}}{1 - a_1 z^{-D1}}\right]\left[\frac{z^{-D2}}{1 - a_2 z^{-D2}}\right]$$

Example DSP source code of the interrupt processing routine for the above figure is shown below.

ADSP-21065L Multi-Tap Delay Effect Implementation Using 2 Delay Lines

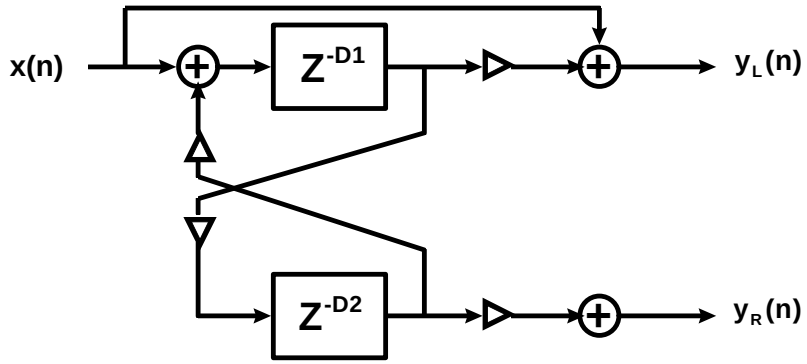
```
#define        a1              0x40000000              /* a1 = 0.50 */
#define        a2              0x33333333              /* a2 = 0.40 */
#define        b0              0x7fffffff              /* b0 = 0.9999 */
#define        b1              0x66666666              /* b1 = 0.80 */
#define        b2              0x4ccd0000              /* b2 = 0.60 */
#define        DelayLn1        3000
#define        DelayLn2        4000

------------------------------------------------------------------------------
/* Serial Port 0 Receive Interrupt Service Routine */
multitap_delay_effect:

/* get input samples from data holders */
   r0 = dm(left_input);              {left input sample}
   r1 = dm(right_input);             {right input sample}

/* tap outputs of circular delay lines */

   m2 =  D1;                         {load delay line 1 tap value}
   r3 = dm(m2, i2);                  {point to d-th tap and put in data register}
                                     {pre-modify address with no update}
   m3 =  D2;                         {load delay line 2 tap value}
   r4 = dm(m3, i3);                  {point to d-th tap and put in data register}

   r2 = b0;
   mrf = r1 * r2 (ssf);              {mrf = b0 * x}

   r2 = b1;
   mrf = mrf + r3 * r2 (ssf);        {mrf = b0 * x + b1 * s1D}

   r2 = b2;
   mrf = mrf + r4 * r2 (ssfr);       {mrf = y = b0 * x + b1 * s1D + b2 * s2D}
   mrf = sat mrf;

{--- write output samples to codec ---------------------------------------}
   r10 = mr1f;
   dm(left_output)  = r10;           {left output sample}
   r10 = mr1f;
   dm(right_output) = r10;           {right output sample}

{--- sample processing algorithm (continued) -----------------------------}
   mrf = 0;
   mr1f = r3;                        {mrf = s1D}
   r5 = a2;
   mrf = mrf + r4 * r5 (ssfr);       {mrf = s20 = s1D + a2 * s2D}
   mrf = sat mrf;

/* put input sample into tap-0 of delay line #2, postmodify address after storage */
   r12 = mr1f; dm(i3, -1) = r12;

   mrf = 0;
   mr1f = r1;                        {mrf = x}
   r5 = a1;
   mrf = mrf + r3 * r5 (ssfr);       {mrf = s10 = x + a1 * s1D}
   mrf = sat mrf;
```

```
/* put input sample into tap-0 of delay line #1, postmodify address after storage */
   r12 = mr1f; dm(i2, -1) = r12;
   rts;
```

Multitap Stereo Ping-Pong Delay

Another interesting delay effect (shown below) consists of alternating delayed replicas of the mono input source between the left and right output speakers using 2 delay-line buffers. Each delay line output is fed into the input of the other delay line.

Figure 48.
Multi-Tap 'Ping-Pong' Delay of a Mono Source to a Stereo Output Field



ADSP-21065L 'Ping-Pong' Delay Effect Implementation Using 2 Delay Lines

```
/* ********************************************************************************************
       STEREO_DELAYS.ASM - 'Ping-Pong' stereo delay effects algorithm with cross-feedback

       Based on Fig.8.4.1 of Introduction to Signal Processing.
       By S. J. Orfanidis - 1996, Prentice-Hall

       Assuming the feedback filters are plain multipliers, GL(z)=aL, GR(z)=aR,
       the sample processing algorithm for each pair xL,xR do:

       What the stereo delay effect does?
       Creating multiple delay typte offects can be obtained from implementing
       simple low-order FIR or IIR filters.  Many DSP audio effects processors widely use
       variations of this effect created with multiple delay lines.

       Stereo delay effects can be implemented by coupling the left and right channels.
       The left and right channels are coupled by introducing cross-feedback coefficients, so
       that the delayed output of one channel is fed into the input of the other channel.

       The I/O equation in the z-domain is:
              YL(z) = HLL(z)*XL(x) + HLR(z)*XR(z)
              YR(z) = HRL(z)*XL(z) + HRR(z)*XR(z)

   ******************************************************************************************** */

/* ADSP-21060 System Register bit definitions */
#include       "def21065l.h"
#include       "new65Ldefs.h"

.GLOBAL        stereo_delay_effect;
.GLOBAL        Init_Delay_Buffers;
.EXTERN        Left_Channel;
.EXTERN        Right_Channel;

/* ------------- DATA MEMORY FILTER BUFFERS ---------------------------------*/
.segment /dm   dm_delay;

/* 32 bit filter coefficients are in 1.31 fractional format */
```

```
#define aL     0x00000000            /*aL = 0 - left self-feedback*/
#define aR     0x00000000            /*aR = 0 - right self-feedback*/

#define bL     0x66666666            /*bL = 0.8 - left delay gain*/
#define bR     0x66666666            /*bR = 0.8 - right delay gain*/

#define cL     0x40000000            /*cL = 0.5 - left direct path gain*/
#define cR     0x40000000            /*cR = 0.5 - right direct path gain*/

#define dL     0x40000000            /*dL = 0.5 - cross feedback from L to R*/
#define dR     0x40000000            /*dR = 0.5 - cross feedback from R to L*/

#define L_2    24000                 /*TL = L/fs = 18000/48000 = 0.375 sec*/
#define R_2    24000                 /*TR = R/fs = 18000/48000 = 0.375 sec*/

/* Stereo Early Reflection Settings for a Reverb Processor */
#define L      3000
#define R      3000

.var    wL[L+1];              /* left delay-line buffer */
.var    wR[R+1];              /* right delay-line buffer */

.endseg;

/* ---------------------- PROGRAM MEMORY CODE --------------------------------*/

.segment /pm pm_code;

Init_Delay_Buffers:

       B2 = wL; L2 = @wL;              /* delay-line buffer pointer and length */
       B3 = wR; L3 = @wR;              /* delay-line buffer pointer and length */
       m2 = 1; m3 = 1;

       LCNTR = L2;                   /* clear left delay line buffer to zero */
        DO clrDline_L UNTIL LCE;
clrDline_L:    dm(i2, m2) = 0;

       LCNTR = L3;                   /* clear right delay line buffer to zero */
        DO clrDline_R UNTIL LCE;
clrDline_R:    dm(i3, m3) = 0;

       RTS;

/* ------------------------------------------------------------------------------------------- */
/*                                                                                             */
/*     stereo delay processing algorithm - processing on both left and right channels          */
/*                                                                                             */
/* ------------------------------------------------------------------------------------------- */

stereo_delay_effect:

       /* get input samples from data holders */
       r0 = dm(Left_Channel);              /* left channel input sample */
       r1 = dm(Right_Channel);             /* right channel input sample */
       r1 = 0x00000000;                    /* for true stereo delay, remove this instruction*/

       /* tap outputs of circular delay lines */

       /* r3 = sLL = L-th tap of left delay */
       m2 =  L;  modify(i2, m2);                    {point to d-th tap}
       m2 = -L;  r3 = dm(i2, m2);                   {put d-th tap in data register}

       /* r4 = sRR = R-th tap of right delay */
       m3 =  R;  modify(i3, m3);                    {point to d-th tap}
       m3 = -R;  r4 = dm(i3, m3);                   {put d-th tap in data register}

       mrf = 0;
       mr1f = r3;                                   {mr1f = sLL}
       r2 = cL;
       mrf = mrf + r0 * r2 (ssfr);                  {mrf = yL = cL * xL + sLL = left output}
       mrf = sat mrf;

       r10 = mr1f;
       dm(Left_Channel) = r10;                      {send result to left output channel}
```

```
mrf = 0;
mr1f = r4;                                  {mr1f = sRR}
r2 = cR;
mrf = mrf + r1 * r2 (ssfr);                 {mrf = yR = cR * xR + sRR = right output}
mrf = sat mrf;

r10 = mr1f;
dm(Right_Channel) = r10;                    {send result to right output channel}

r2 = bL;
mrf = r0 * r2 (ssf);                        {mr = bL * xL}
r2 = aL;
mrf = mrf + r3 * r2 (ssf);                  {mr = bL * xL + aL * sLL}
r2 = dR;
mrf = mrf + r4 * r2 (ssfr);                 {mr = bL * xL + aL * sLL + dR * sRR}
mrf = sat mrf;

/* put value from MAC result into tap-0 of left delay-line */
m2 = 0;
r12 = mr1f; dm(i2, m2) = r12;

/* backshift pointer, update left circular delay-line buffer */
m2 = -1;  modify(i2, m2);

r2 = bR;
mrf = r1 * r2 (ssf);                        {mr = bR * xR}
r2 = aR;
mrf = mrf + r4 * r2 (ssf);                  {mr = bR * xR + aR * sRR}
r2 = dL;
mrf = mrf + r3 * r2 (ssfr);                 {mr = bR * xR + aR * sRR + dL * sLL}
mrf = sat mrf;

/* put value from MAC into delay line into tap-0 of right delay-line */
m3 = 0;
r12 = mr1f; dm(i3, m3) = r12;

/* backshift pointer, update right circular delay-line buffer */
m3 = -1;  modify(i3, m3);
rts;
```

3.3.2 **Delay Modulation Effects**

Delay Modulation Effects are some of the more interesting type of audio effects but are not computationally complex. The technique used is often called Delay-Line Interpolation [6], where the delay-line center tap is modified, usually by some low frequency waveform. The result of interpolating/decimating samples within the delay line results in a slight pitch change of the input signal. Thus, one type of pitch shift algorithm can fall under this category, although there are other DSP methods for pitch shifting. Below is a listing of some effects that fall under Delay-Line Modulation:

Chorus - Simulation of multiple instruments/voices
Flanger - 'Swooshing Jet Sound'
Doppler - Pitch Change increase./decrease of an object moving towards/away from listener.
Pitch Shifting - Changing frequency of an input source
Detune - Very slight pitch change added with the input to simulate 2 voices.
Doubling - Adding a small delay/pitch change with an input source.
Leslie Rotating Speaker Emulation - Combination of Vibrato and Tremolo.

Figure 49 summarize some common types of modulators used for moving the center tap of a delay-line [2, 16, 26].

Figure 49. Common Methods of Modulation

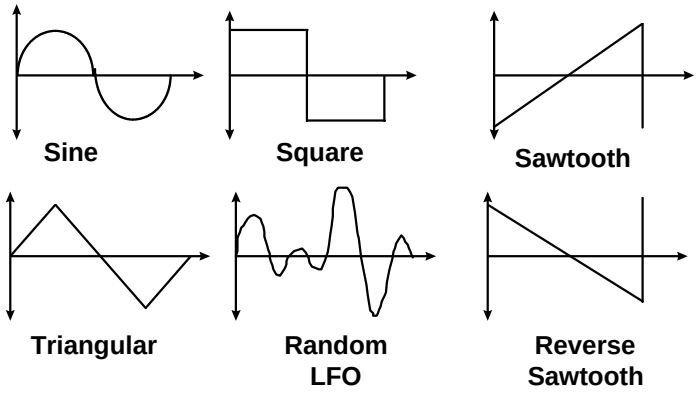


Figure 50. Delay-Line Modulation General Structure (J. Dattorro)

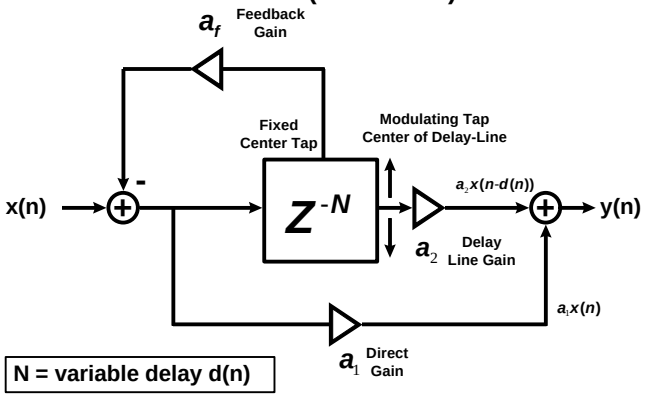
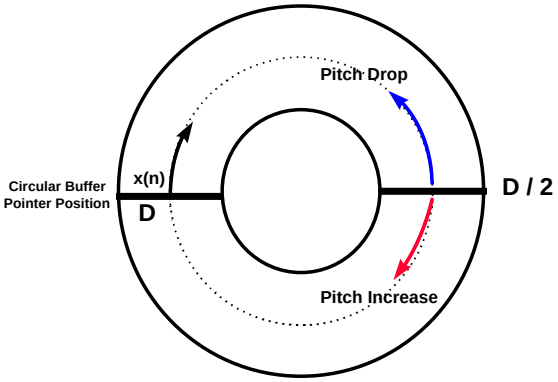


Figure 51. Visualizing The Result Of A Rotating Center Tap



The above general structure (Figure 50) described by J. Dattorro [6] will allow the creation of many different types of delay modulation effects. Each input sample is stored into the delay line, while the moving output tap will retrieved from a different location in the buffer rotating from the tap center (Figure 51). When the small delay variations are mixed with the direct sound, a *time-varying comb filter* results [2, 6].

The General Delay Line Equation for the above structure is:

$$y(n) = a_1 x(n) + a_2 x(n - d(n)) - a_f (n - D_{fixed})$$

and,

d(n) rotates around tap center of delay line D

As we will see, the above general structure will allow the creation of many different types of delay modulation effects. Each input sample is stored into the delay line, while the moving output tap will retrieved from a different location in the buffer rotating from the tap center. If a delay of an input signal is very small (around 10 msec), the echo mixed with the direct sound will cause certain frequencies to be enhanced or canceled (due to the comb filtering). This will cause the output frequency response to change. By varying the amount of delay time when mixing the direct and delayed signals together, the variable delay lines create some amazing sound effects such as chorusing and flanging.

3.3.2.1 Flanger Effect

Flanging was coined by the way it was accidentally discovered. As legend has it, a recording engineer was recording a signal onto 2 reel-to-reek tapedecks and monitored from both playback heads of the 2 tapedecks at the same time. While trying to simulate the ADT or doubling effect, it was discovered that small changes in the tape speed between the 2 decks created a 'swooshing' jet sound. This effect was further enhanced by repeatedly leaning on the flanges of one of the tape reels slightly to slow the taped down. Thus the flanger was born.

It is very easy to recreate this effect using a DSP. Flanging can be implemented in a DSP by varying the input signal with a small, variable time delay at a very low frequency between 0.25 to 25 milliseconds and adding the delayed replica with the original input signal (Figure 52). When the time delay offset is varied by rotating the delay-line center tap, the in-phase and out-of-phase frequencies as a result of the comb filtering sweep up and down the frequency spectrum (Figure 53). The "swooshing" jet engine effect created as a result is referred to as flanging.

Figure 52.
Implementation of a Flanger Effect

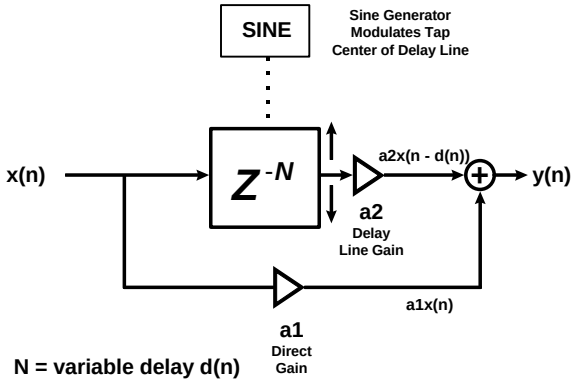
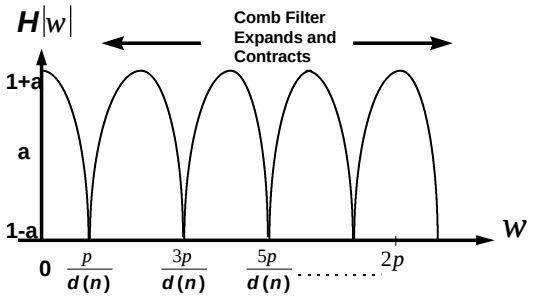


Figure 53.
Frequency Response of the Flanger



The Frequency Response of the Flanger results in a Comb Filter. As the Delay Increase, the number of peaks increases. Changing the Delay modifies the comb filter, which in turn affects the frequencies that are enhanced or cancelled.

By modifying the single reflection echo equation, the flanger can be implemented as follows:

$$y(n) = x(n) + ax(n - d(n))$$

scaling each signal equally by ½ to prevent overflow:

$$y(n) = \frac{1}{2}[\, x(n) + x(n - d(n)\,]$$

Flanging is created by periodically varying delay d(n). The variations of the delay time (or delay buffer size) can easily be controlled in the DSP using a low-frequency oscillator sine wave lookup table (see Figures 54 and 55) that calculates the variation of the delay time, and the update of the delay is determined on a sample basis or by the DSP's on-chip timer. To sinusoidally vary the delay between 0< d(n) < D, the on chip timer interrupt service routine should calculate the following equation described by Orfanidis [2]:

$$d(n) = \frac{D}{2}\left[\, 1 - \cos(\, 2p\, nf_{cycle}\,)\,\right]$$

Figure 54.
Sine Wavetable Circular Buffer Storage

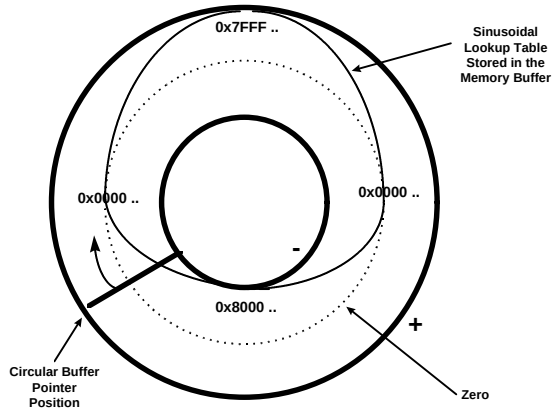


Figure 55.
Example 4 K-word Sinusoidal Wavetable

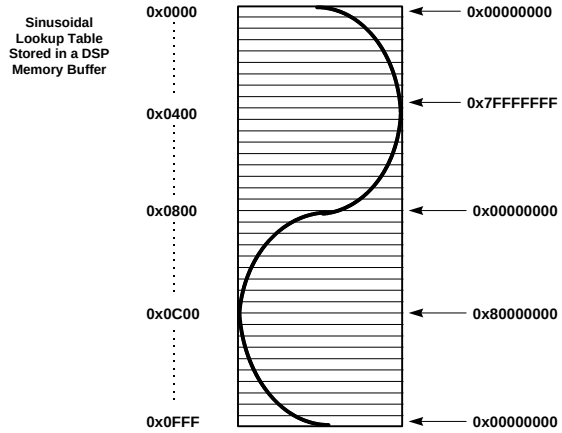
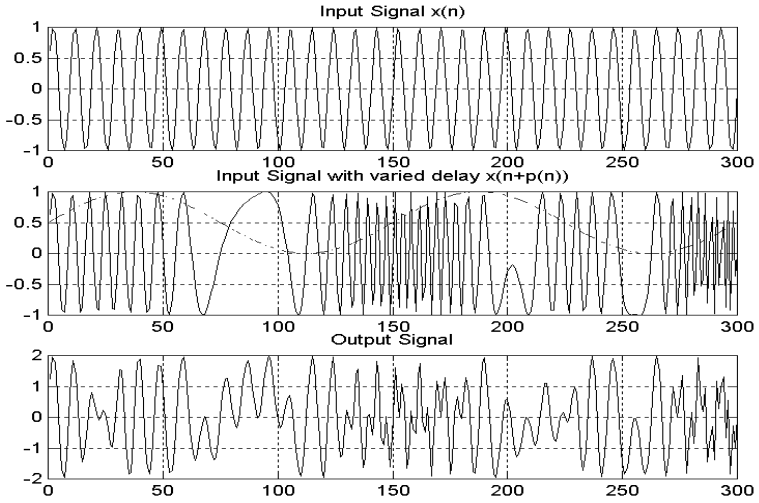


Figure 66.
"Flanger" Result of Adding a Variable Delayed Signal To It's Original



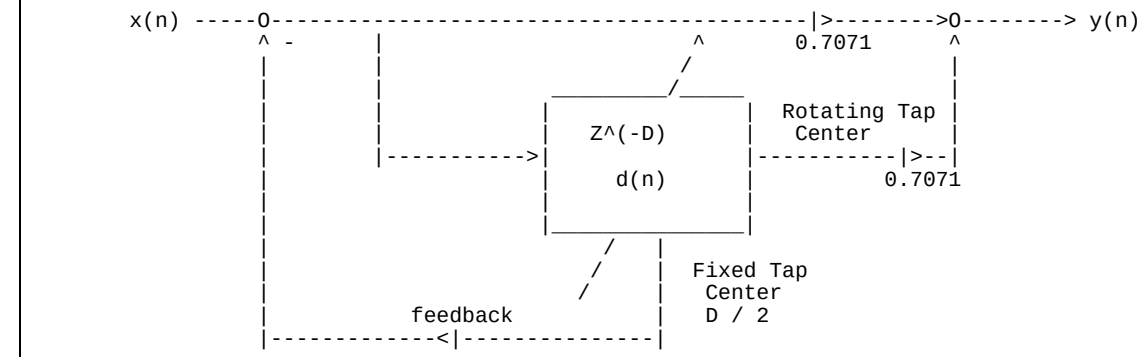
This sinusoidal LFO's frequency is usually controlled by the ***'sweep rate'*** parameter. The LFO can easily be implemented on the DSP by creating a sine wavetable lookup that determines the variation of the delay time. The determination of the delay time can be updated periodically using the on chip programmable timer.

Another control parameter called the '***sweep depth'*** *D* will determine how much the time will change during a sinusoidal cycle. The larger the size of the delay-line buffer, the farther the phase cancellations and reinforcements will move up and down the frequency spectrum.

Example Implementation of the Flanger Routine Using the ADSP-21065L:

```
/* ************************************************************************************************
       FLANGER.ASM - flanging effect - "swooshing jet/doppler" sound

       Flanger Effect as Described by Jon Dattorro in "Effect Design Part 2 -
       Delay-Line Modulation and Chorus," J. Audio Eng. Socienty, Vol. 45, No. 10,
       October 1997
```

```
        Delay calculation Based on Eqs.(8.2.18-8.2.19) of Introduction to Signal Processing.
        By S. J. Orfanidis - 1996, Prentice-Hall
        ISBN 0-13-209172-0

        This version uses Linear Interpolation (versus Allpass Interpolation described by Dattorro)

        I/O equation:
                     y(n) = 0.7071 * x(n) + 0.7071*x(n - d(n)) - 0.7071*x(n - D/2)
```




```
        What the flanging effect does?
        Flanging consists of modulating a delayed replica of an input by a few milliseconds,
        and adding this delayed signal together with the input, which will then cause phase shifting
        and spreading of the audio signal as a result of comb filtering. The delay is modulated using
        a low frequency sinusoid.  The effect works best on drums, guitars, keyboards, and some vocals.

        For each input sample, the sample processing algorithm does the following:
                     store input sample s0 to the flanger delay line buffer - *p = s0 = xinput
                     generated variable delay, d = (D - D * sin(2*pi*fc*t)) / 2
                     s1 = delayed sample = tap(D, w, p, d)
                     y = a0 * s0 + a1 * s1

        Developed for the 21065L EZ-LAB Evaluation Board

************************************************************************************************ */

/* ADSP-21060 System Register bit definitions */
#include       "def21065l.h"
#include       "new65Ldefs.h"

.GLOBAL        Flanger_Effect;
.GLOBAL        Init_Flange_Buffers;
.GLOBAL        Timer0_Initialization;
.GLOBAL        change_depth_rate_width;
.GLOBAL        select_flange_feedback_gain;
.GLOBAL        wavetable_gen;
.EXTERN        Left_Channel;
.EXTERN        Right_Channel;

.segment /dm    dmflange;


/* Flanger Control Knobs */
#define D             345                  /* TD = D/fs = 400/8000 = 50 msec */
#define D2            D/2                  /* Nominal Tap Center of Delay Line D */
#define WaveSize      4000                 /* sinusoidal wavetable */
#define modulation_rate       80000
#define a0  0x5A82799A                     /* a0 = 0.707106781, nominal input gain */
#define a1  0x5A82799A                     /* a1 = 0.707106781, nominal output gain of tapped delay
line */

.var    IRQ1_counter = 0x00000003;
.var    IRQ2_counter = 0x00000003;

.var    DRY_GAIN = 0x7FFFFFFF;
.var    WET_GAIN = 0x7FFFFFFF;             /* For inverted phase, set to 0x80000000 */
.var    feedback_gain = 0xA57D8666;       /* FEEDBACK = -0.707106781, inverted o/p of fixed center tap */
                    /* 0x5A82799A */       /* For positive FEEDBACK, 0.707106781 */
```

```
.var    sweep_rate = 1;                     /* controls M register, signal frequency fc = c*Mreg*ftimer
*/
.var    sweep_width = 0;                    /* controls width of sweep, ranges from 0 to - 15 */
.var    w[D + 1];                           /* delay-line buffer, max delay = D */
.var    sine_value;                         /* wavetable update via timer for delay calcuation */
.var    sine[WaveSize] = "sinetbl.dat";     /* min frequency f1 = fs/Ds = 8/4 = 2 Hz */
                                            /* load one period of the wavetable */
.endseg;


/* -------------PROGRAM MEMORY CODE--------------------------------*/
.segment /pm pm_code;

Init_Flange_Buffers:
       B2 = w;  L2 = @w;                    /* delay-line buffer pointer and length */
       m2 = 1;

       LCNTR = L2;                          /* clear delay line buffer to zero */
        DO clrDline UNTIL LCE;
clrDline:         dm(i2, m2) = 0;

       B6 = sine;                           /* pointer for signal generator */
       L6 = @sine;                          /* get size of sine table lookup */

       RTS;

/*-----------------------------------------------------------------*/

/* Set up timer for the Chorus Effects wavetable generator */

Timer0_Initialization:
       bit clr mode2 TIMEN0;                /* timer off initially */
       bit set mode2 PWMOUT0 | PERIOD_CNT0 | INT_HI0;      /* latch timer0 to high priority timer int */

       r0 = modulation_rate;
       DM(TPERIOD0) = r0;
        DM(TCOUNT0) = r0;                   /* assuming 16.7 nSec cycle @ 60 MIPs */
       r0 = 10;
       DM(TPWIDTH0) =r0;

       bit set imask TMZHI;                 /* timer high priority  */
       bit set mode2 TIMEN0;                /* timer on */

       rts;

/* ------------------------------------------------------------------------------------------- */
/*                                                                                             */
/*                    Wavetable Generator used for Flange Delay Line Modulation                */
/*                                                                                             */
/* ------------------------------------------------------------------------------------------- */
/* High Priority Timer Interrupt Service Routine for Delay Line Modulation of Flange Buffer */
/* This routine is a wavetable generator, where r3 = where r3 = D2 * sin(2*pi*fc*t)   */
/* and it modulates the delay line around rotating tap center */

wavetable_gen:
       bit set mode1 SRRFL;          /* enable secondary registers r0 - r8 */
       nop;                          /* 1 cycle latency writing to Mode1 register */

       m6 = dm(sweep_rate);          /* desired increment c - frequency f = c x fs / WaveSize */
       r1 = D2;                      /* Nominal Center Tap Delay Time */
       r2 = dm(i6, m6);              /* get next value in wavetable */
       r4 = dm(sweep_width);         /* store to memory for chorus routine */
       r2 = ashift r2 by r4;         /* control amount of variable delay via sweep_width */
       r3 = r1 * r2 (SSFR);          /* scale Nominal Delay Time by a fractional value */
       dm(sine_value) = r3;          /* save for flange routine */
       rti(db);
       bit clr mode1 SRRFL;          /* disable secondary register set */
       nop;                          /* 1 cycle latency to write to mode1 register */

/* ------------------------------------------------------------------------------------------- */
/*                                                                                             */
/*                    Digital Flanger - process right channel only                             */
/*                                                                                             */
/* ------------------------------------------------------------------------------------------- */
```

```
Flanger_Effect:
       r15 = DM(Right_Channel);      /* get x-input, right channel */

       r3 = dm(sine_value);          /* calculate time-varing delay deviation */
       r14 = D2;                     /* nominal tap center delay */
       r4 = r14 - r3;                /* r4 = d = D2 - D2 * sin(2*pi*fc*t) */

       /* r4 now will be used to set m2 register to fetch time varying delayed sample */
       m2 = r4;                      /* tap outputs of circular delay line */
       modify(i2, m2);               /* go to delayed sample */
       r4 = -r4;                     /* negate to get back to where we were */
       m2 = r4;                      /* used to post modify back to current sample */
       r6 = dm(i2, m2);              /* get time-varying delayed sample */

       /* r8 will be used to get Nominal Tap Center delayed sample for flange feedback */
       m2 = D2;                      /* tap outputs of circular delay line */
       modify(i2, m2);               /* go to delayed sample */
       m2 = -D2;                     /* negate to get back to where we were */
       r8 = dm(i2, m2);              /* get nominal delayed sample, postmodify back to current sample */

       /* crank out difference equation */
       r5 = a0;                      /* input gain */
       mrf = r15 * r5 (SSF);         /*   mrf = a0 * x    */
       r9 = dm(feedback_gain);       /* gain for feedback of nominal tap center*/
       mrf = mrf + r8 * r9 (SSF);    /*   mrf = a0 * x - af * sNominal */
       r12 = mr1f;                   /* save for input to flanger delay line */

       r7 = a1;                      /* delay line gain */
       mrf = mrf + r7 * r6 (SSFR);   /*   mrf = a0 * x + a * s1 - af * sNominal  */
       mrf = SAT mrf;                /* saturate if necessary */
       r10 = mr1f;                   /* flanged result in r10 */

       /* put 'input minus feedback' sample from r12 into tap-0 of delay line */
       /* and backshift circular delay-line buffer pointer */
       dm(i2, -1) = r12;

       /* send flanged result to both left and right output channels */
       DM(Left_Channel)=r10;
       DM(Right_Channel)=r10;

       rts;

/* ------------------------------------------------------------------------------ */
/*                                                                                */
/*                     IRQ1 Pushbutton Interrupt Service Routine                  */
/*                                                                                */
/*     This routine allows the user to modify flanger width and rate presets.     */
/*                                                                                */
/*     Default before 1st IRQ push:                                               */
/*     1st Pushbutton Press:                                                      */
/*     2nd Pushbutton Press:                                                      */
/*     3rd Pushbutton Press:                                                      */
/*     4th Pushbutton Press:                                                      */
/*     5th Pushbutton Press:                                                      */
/*     6th Pushbutton Press:                                                      */
/*     7th Pushbutton Press:  Reverts back to 1st Pushbutton Press                */
/*                                                                                */
/*     The pushbutton setting is shown by the active LED setting, all others are  */
/* ------------------------------------------------------------------------------*/

change_depth_rate_width:
       bit set mode1 SRRFH;                /* enable background register file */
       NOP;                                /* 1 CYCLE LATENCY FOR WRITING TO MODE1 REGISER!!   */

       r13 = 6;                            /* number of presets */
       r15 = DM(IRQ1_counter);             /* get preset count */
       r15 = r15 + 1;                      /* increment preset */
       comp (r15, r13);
       if ge r15 = r15 - r15;              /* reset to zero */
       DM(IRQ1_counter) = r15;             /* save preset count */

       r10 = pass r15;                     /* get preset mode */
       if eq jump delay_settings_2;        /* check for count == 0 */
       r10 = r10 - 1;
       if eq jump delay_settings_3;        /* check for count == 1 */
```

```
        r10 = r10 - 1;
        if eq jump delay_settings_4;         /* check for count == 3 */
        r10 = r10 - 1;
        if eq jump delay_settings_5;         /* check for count == 4 */
        r10 = r10 - 1;
        if eq jump delay_settings_6;         /* check for count == 5 */

delay_settings_1:                            /* count therefore, is == 6 if you are here */
        r14 = 1;              DM(sweep_rate) = r14;
        r14 = 0;              DM(sweep_width) = r14;
        bit set ustat1 0x3E;                        /* turn on Flag4 LED */
        bit clr ustat1 0x01;
        dm(IOSTAT)=ustat1;
        jump done;

delay_settings_2:
        r14 = 2;              DM(sweep_rate) = r14;
        r14 = 0;              DM(sweep_width) = r14;
        bit set ustat1 0x3D;                        /* turn on Flag5 LED */
        bit clr ustat1 0x02;
        dm(IOSTAT)=ustat1;
        jump done;

delay_settings_3:
        r14 = 3;              DM(sweep_rate) = r14;
        r14 = -1;             DM(sweep_width) = r14;
        bit set ustat1 0x3B;                        /* turn on Flag6 LED */
        bit clr ustat1 0x04;
        dm(IOSTAT)=ustat1;
        jump done;

delay_settings_4:
        r14 = 4;              DM(sweep_rate) = r14;
        r14 = -1;             DM(sweep_width) = r14;
        bit set ustat1 0x37;                        /* turn on Flag7 LED */
        bit clr ustat1 0x08;
        dm(IOSTAT)=ustat1;
        jump done;

delay_settings_5:
        r14 = 5;              DM(sweep_rate) = r14;
        r14 = -2;             DM(sweep_width) = r14;
        bit set ustat1 0x2F;                        /* turn on Flag8 LED */
        bit clr ustat1 0x10;
        dm(IOSTAT)=ustat1;
        jump done;

delay_settings_6:
        r14 = 6;              DM(sweep_rate) = r14;
        r14 = 0;              DM(sweep_width) = r14;
        bit set ustat1 0x1F;                        /* turn on Flag9 LED */
        bit clr ustat1 0x20;
        dm(IOSTAT)=ustat1;

done:
        rti(db);
        bit clr mode1 SRRFH;                        /* switch back to primary register set */
        nop;

/* ------------------------------------------------------------------------------------ */
/*                    IRQ2 Pushbutton Interrupt Service Routine                         */
/*                                                                                      */
/*      Intensifies the effect of the flanged sound to sound more metallic.             */
/*      Negative Feedback subtracts (inverts) the output of the fixed tap center output */
/*      Positive Feedback adds the fixed tap center output of the flange delay line     */
/*                                                                                      */
/*      Default before 1st IRQ push:  Delay = 20.83 msec                                */
/*      1st Pushbutton Press:  Feedback Setting #1 -                                    */
/*      2nd Pushbutton Press:  Feedback Setting #2 -                                    */
/*      3rd Pushbutton Press:  Feedback Setting #3 -                                    */
/*      4th Pushbutton Press:  Feedback Setting #4 -                                    */
/*      5th Pushbutton Press:  Feedback Setting #4 -                                    */
/*      6th Pushbutton Press:  Feedback Setting #4 -                                    */
/*      7th Pushbutton Press:  Reverts back to 1st Pushbutton Press                     */
/*                                                                                      */
```

```
/*      The pushbutton setting is shown by the inactive LED setting, all others are on     */
/*      (reverse of IRQ1 LED settings, which show lit LED for  setting #)                  */
/* ------------------------------------------------------------------------------------- */

select_flange_feedback_gain:
       bit set mode1 SRRFH;                 /* enable background register file */
       NOP;                                 /* 1 CYCLE LATENCY FOR WRITING TO MODE1 REGISER!!   */

       r13 = 6;                             /* number of presets */
       r15 = DM(IRQ2_counter);              /* get preset count */
       r15 = r15 + 1;                       /* increment preset */
       comp (r15, r13);
       if ge r15 = r15 - r15;               /* reset to zero */
       DM(IRQ2_counter) = r15;              /* save preset count */

       r10 = pass r15;                      /* get preset mode */
       if eq jump feedback_settings_2;      /* check for count == 0 */
       r10 = r10 - 1;
       if eq jump feedback_settings_3;      /* check for count == 1 */
       r10 = r10 - 1;
       if eq jump feedback_settings_4;      /* check for count == 3 */
       r10 = r10 - 1;
       if eq jump feedback_settings_5;      /* check for count == 4 */
       r10 = r10 - 1;
       if eq jump feedback_settings_6;      /* check for count == 4 */

feedback_settings_1:                        /* count therefore, is == 5 if you are here */
       /* no feedback */
       r14 = 0x00000000;     DM(feedback_gain) = r14;
       bit clr ustat1 0x3E;
       bit set ustat1 0x01;                 /* turn off Flag4 LED */
       dm(IOSTAT)=ustat1;
       jump exit;

feedback_settings_2:
       /* add some small positive feedback */
       r14 = 0x20000000;     DM(feedback_gain) = r14;
       bit clr ustat1 0x3D;
       bit set ustat1 0x02;                 /* turn off Flag5 LED */
       dm(IOSTAT)=ustat1;
       jump exit;

feedback_settings_3:
       /* add some small negative feedback */
       r14 = 0x5A82799A;     DM(feedback_gain) = r14;
       bit clr ustat1 0x3B;
       bit set ustat1 0x04;                 /* turn off Flag6 LED */
       dm(IOSTAT)=ustat1;
       jump exit;

feedback_settings_4:
       /* add a medium amount of positive feedback */
       r14 = 0xA57D8666;     DM(feedback_gain) = r14;
       bit clr ustat1 0x37;
       bit set ustat1 0x08;                 /* turn off Flag7 LED */
       dm(IOSTAT)=ustat1;
       jump exit;

feedback_settings_5:
       r14 = 0x67FFFFFF;     DM(feedback_gain) = r14;
       bit clr ustat1 0x2F;
       bit set ustat1 0x10;                 /* turn off Flag8 LED */
       dm(IOSTAT)=ustat1;
       jump exit;

feedback_settings_6:
       r14 = 0x90000000;     DM(feedback_gain) = r14;
       bit clr ustat1 0x1F;
       bit set ustat1 0x20;
       dm(IOSTAT)=ustat1;                   /* turn off Flag9 LED */
exit:
       rti(db);
       bit clr mode1 SRRFH;                 /* switch back to primary register set */
       nop;
.endseg;
```

3.3.2.2 Chorus Effect

Chorusing is used to "thicken" sounds. This time delay algorithm (between 15 and 35 milliseconds) is designed to duplicate the effect that occurs when many musicians play the same instrument and same music part simultaneously. Musicians are usually synchronized with one another, but there are always slight differences in timing, volume, and pitch between each instrument playing the same musical notes. This chorus effect can be re-created digitally with a variable delay line rotating around the tap center, adding the time-varying delayed result together with the input signal.

Using this digitally recreated effect, a 6-string guitar can also be 'chorused' to sound more like a 12-string guitar. Vocals can be thickened to sound like more than one musician is singing.

The chorus algorithm is similar to flanging, using the same difference equation, except the delay time is longer. With a longer delay-line, the comb filtering is brought down to the fundamental frequency and lower order harmonics (Figure 70). Figure 67 shows the structure of a chorus effect simulating 2 instruments [2, 6].

Figure 67. Implementation of a Chorus Effect Simulating 2 Instruments

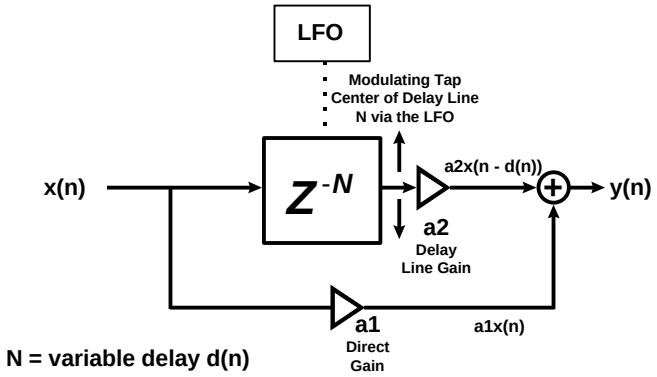
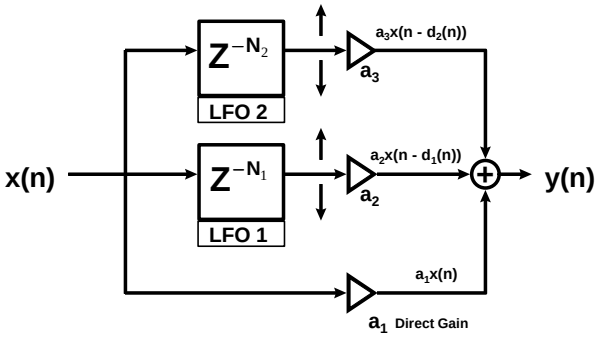


Figure 68. Chorus Effect Simulating 3 Instruments



The difference equation is for Figure 67 is:

$$y(n) = a_1 x(n) + a_2 x(n - d(n))$$

Some example difference equations for simulating 2 or 3 musicians are shown below.

An example fixed point fractional implementation is:

$$y(n) = \frac{1}{2}x(n) + \frac{1}{2}x(n - d(n))$$

Scaling each signal by ½ will equally mix both signal to around the same volume while ensuring no overflow when the signals are added.

To implement a chorus of 3 instruments, 2 variable delay lines can be used (Figure 68). Use a scaling factor of 1/3 to prevent overflow with fixed point math while mixing all three signals with equivalent gain.

$$y(n) = \frac{1}{3}x(n) + \frac{1}{3}x(n - d_1(n)) + \frac{1}{3}x(n - d_2(n))$$

Another Implementation Example as described by Orfanidis[2] is:

$$y(n) = \frac{1}{3}\left[x(n) + a_1(n)x(n - d1(n)) + a_2(n)x(n - d2(n)) \right]$$

where the gain coefficients a1(n) and a2(n) can be a low-frequency random number with unity mean.

The small variations in the delay time can be introduced by a *random LFO* at around 3 Hz. A low frequency random LFO lookup table (for example, Figure 69) can be used to recreate the random variations of the musicians, although the circular buffer will still be periodic.

Figure 69.
Example 4K Random LFO Wavetable Storage

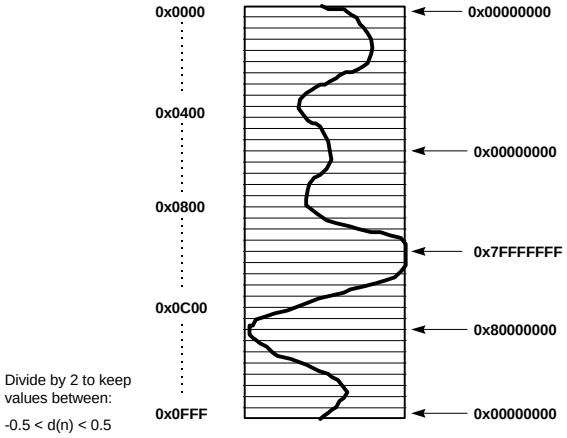
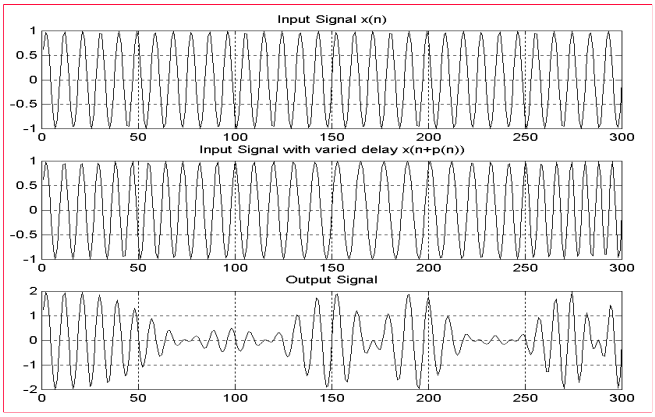


Figure 70.
Chorus Result of Adding a Variable Delayed Signal To It's Original

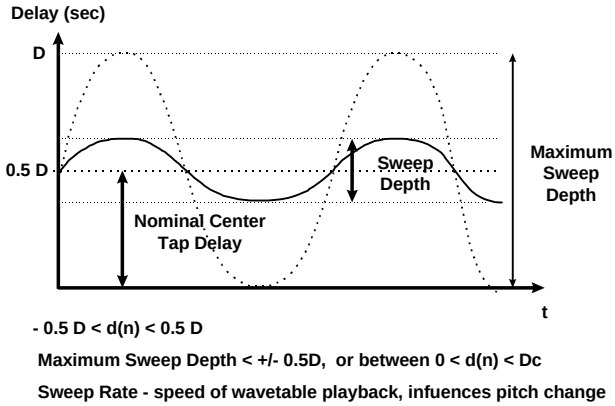


A result of an increasing slope in the LFO will cause the pitch to be lower. A negative slop will result in a pitch increase. The LFO value in the table can be updated on a sample basis via the chorus processing routine, or the wavetable look-up can be modified using the DSP's on-chip programmable timer. The varying delay d(n) will be updated using the following equation:

$$\mathbf{d(n) = D(0.5 + LFO(n))}, \text{ or } \mathbf{d(n) = D(0.5 + v(n))}$$

The signal v(n) is described by Orfanidis [2] as a zero-mean low-frequency random signal varying between [-0.5,0.5]. An easy technique to ensure fixed point signals stay within this range would be to take a lookup table with fractional numbers ranging from -1 to 0.9999 and dividing each lookup value by 2.

Figure 71.
Chorus Delay Parameters



Chorus Parameters

Like the flanger, most units offer "*Modulation*" *(or Rate)* and "*Depth*" controls.

- *Depth (or Delay)*

controls the length of the delay line, allowing a user to change the length on-the-fly.

- *Sweep Depth*

Determines how much the time offset changes during an LFO cycle. It combined with the delay line value for a total delay used to process the signal.

- *Modulation*

The variations in delay time will be introduced by a **low-frequency oscillator (LFO)**. This frequency can usually be controlled with the "*Sweep Rate*" parameter. Usually, the LFO consists of a low frequency random signal. When the waveform is at the largest value, variable delay that results will be the maximum delay possible. A result of an increasing slope in the LFO will cause the pitch to be lower. A negative slop will result in a pitch increase.

Sine and Triangle waves can be used to vary the delay time. One easy method for generating the modulation value is through a wavetable lookup. The value in the table can be modified on a sample basis via the chorus routine, or the lookup can be determined using the DSP's on-chip programmable timer. When the timer count expired and the DSP vectors off to the Timer Interrupt Service Routine, the modulation value can then be updated with the next value in the waveform buffer. The LFO can be repeated continuously by making the wavetable a circular buffer. Using a cosine wavetable, the varying delay d(n) will be updated using the following equation:

. $$\mathbf{d(n) = D(0.5 + LFO(2p\, nf_{Delay}))}$$

where D = Delay Line Length
Fdelay = Frequency of the LFO with a period of 2 Pi of the LFO
n = the nth location in the wavetable lookup

The small variations in the time delays and amplitudes can also be simulated by varying them **randomly** at a very low frequency around 3 Hz.

$$\mathbf{d(n) = D(0.5 + v(n))}$$

where,

v(n)= current variable delay value from the random LFO generator

or,

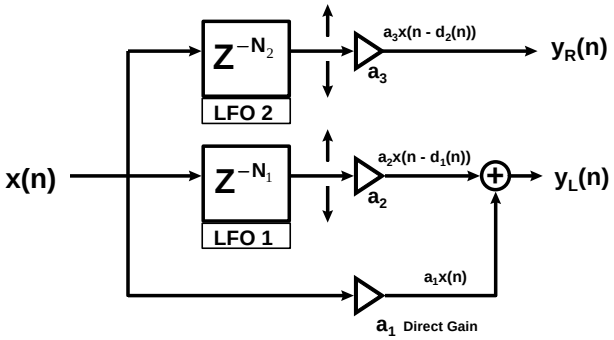
$$\mathbf{d(n) = D(0.5 + random\_LFO\_number(n))}$$

The signal v(n) as described by Orfanidis [2] is a zero-mean low-frequency random signal varying between [-0.5,0.5]. An easy technique to ensure fixed point signals stay within this range would be to take a lookup table with fractional numbers ranging from -1 to 0.9999 and dividing each lookup value by 2. This can easily be done with an arithmetic shift instruction shifting the input to the right by 1 binary place to divide the lookup value by 2.

Stereo Chorus

This effect is achieved by panning the chorus result on back and forth on both stereo channels, creating the impression of movement of the sound in space. The effect can also be created by sending the unaltered input signal on one output stereo channel and the chorused result to the opposite channel.

Example Stereo Chorus Effect



Can use same LFO table with pointers 90 degrees apart.

Figure 72.

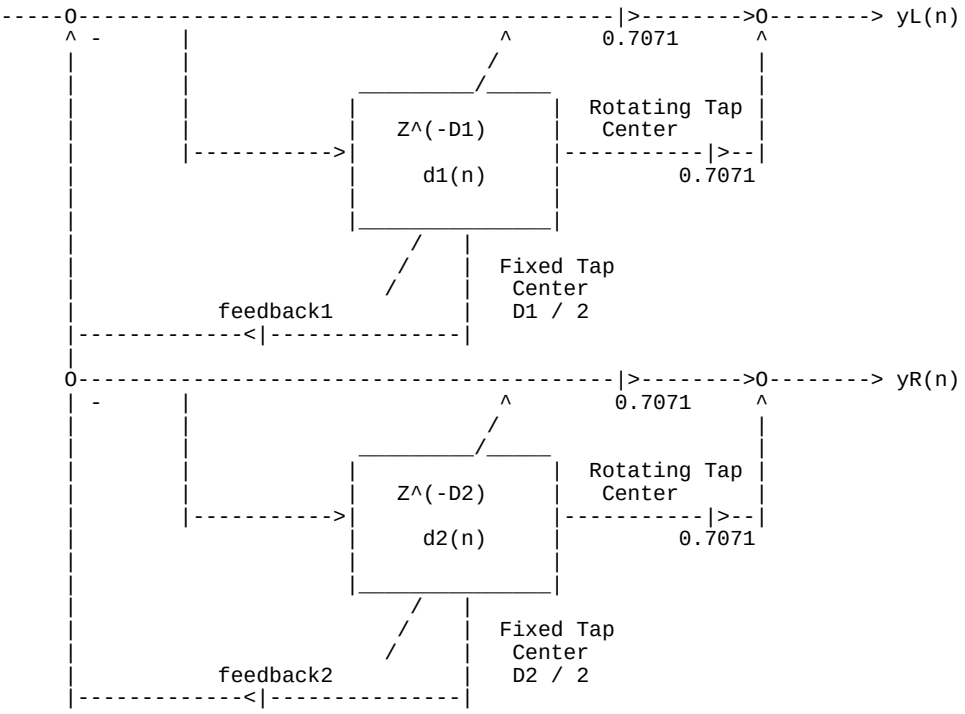
Flanging/Chorusing Similarities and Differences

Both Flanging and Chorusing use variable buffers to change the time delay on the fly. Both effects achieve these variations in delay time by using a low frequency oscillator (LFO). This parameter is available on commercial units as the "sweep rate". The "sweep-depth" parameter is what determines the amount of delay in the sweep period. The greater the depth, the farther the peaks and dips of the phase cancellation.

The key difference between the two effects is the flanger found in many commercial units changes the delay using a low frequency sine-wave generator, where the chorus usually changes the delay using a low-frequency random noise generator. In addition, the flanger modulates the length of the delay from 0 to D, while the chorus modulates the delay from ???? (expand further)

Example Stereo Chorus Implementation of 3 instruments Using the ADSP-21065L

```
/*****************************************************************************************
 STEREO_CHORUS_FEEDBACK.ASM - stereo chorusing effect simulating 3 voices/musical instruments
       (SPORT1 Rx ISR count & update method - delay calculation determined by a counter
       incremented in the serial port's audio processing routine)

       Chorus Effect as Described by:
       1. Jon Dattorro in "Effect Design Part 2 - Delay-Line Modulation and Chorus,"
          J. Audio Eng. Socienty, Vol. 45, No. 10, October 1997

       2. Eq(8.2.20) of Introduction to Signal Processing.
          By Sophocles J. Orfanidis - 1996, Prentice-Hall
          ISBN 0-13-209172-0

       This version uses Linear Interpolation (versus Allpass Interpolation) along with
       integer (not fractional) sample delay.  Since the sample rate is 48K, for most
       lower bandwidth signals, Linear Interpolation is probably adequate for most
       instruments.

        I/O equations:
              yL(n) = 1.0 * x(n) + 0.7071*x(n - d1(n)) - 0.7071*x(n - D1/2)
                yR(n) = 1.0 * x(n) + 0.7071*x(n - d2(n)) - 0.7071*x(n - D2/2)
```




```
       What the Chorusing Effect does?
       Chorusing simulates the effect of multiple instruments/voices playing the same musical
       arrangement at the same time.  In actual concert situations, musicians are usually
       synchronized together, except for small variations in amplitude and timing.
       This effect is achieved by allowing the time delay (and also amplitude) to vary
       randomly or sinusoudally in time by using a random number generation routine or sine
       wavetable.


        For each input sample, the sample processing algorithm does the following:
                     store input sample s0 to 2 delay lines
                     modify wavetable(when necessary)
                     generated variable delay, d = D * (0.5 + randnum(fc*t))
                     s1 = sample1 = tap(D, w1, p1, d)
                     s2 = sample2 = tap(D/2)
                     y = a0 * s0 + a1 * s1 - af * s2
```

```
      Developed for the 21065L EZ-LAB Evaluation Board
   ********************************************************************************************/

/* ADSP-21060 System Register bit definitions */
#include       "def21065l.h"
#include       "new65Ldefs.h"

.GLOBAL        chorus_effect;
.GLOBAL        Init_Chorus_Buffers;
.GLOBAL        change_depth_rate_width;
.GLOBAL        select_feedback_gain;

.EXTERN        Left_Channel;
.EXTERN        Right_Channel;

/* Chorus Control Parameters */
#define a0L            0x7FFFFFFF            /* a0 = 0.99999999, left input gain */
#define a1L            0x5A82799A            /* a1 = 0.707106781, left output gain of tapped delay
                                                line */
#define a0R            0x7FFFFFFF            /* a0 = 0.99999999, right input gain */
#define a1R            0x5A82799A            /* a1 = 0.707106781, left output gain of tapped delay
                                                line */
#define afL            0x5A82799A            /* a0 = 0.707106781, negative feedback gain */
#define afR            0x5A82799A            /* a0 = 0.707106781, negative feedback gain */
                                             /* = 0xA57D8666 for positive feedback, - 0.707106781 */
#define D1             870                   /* Depth, or TD = D1/fs = 870/48000 = 18 msec */
#define D2             1120                  /* Depth, or TD = D2/fs = 1120/48000 = 23 msec */
#define DepthL         D1                    /* DepthL is equivalent to time delay of signal, or
                                                d-line 1 size */
#define DepthR         D2                    /* DepthR is equivalent to time delay of signal, or
                                                d-line 2 size */
#define L_Del_TapCenter        DepthL/2      /* D1/2, used for add & subtracting delay from
                                                nominal tap center */
#define R_Del_TapCenter        DepthR/2      /* D2/2, used for add & subtracting delay from
                                                nominal tap center */
#define WaveSize       4000                  /* semi-random/sinusoidal wavetable size*/

/* chorus control parameters */
#define c              1       /* wavetable signal freq fc1 = c1*(update time)*WaveSize = 4 Hz */
#define chorus_width  -2       /* Enter # from 1 to 31.  Do not enter 0, to keep #'s +/- 0.5 */
#define modulation_rate 80     /* Update wavetable pointer for delay calc every 40 interrupts */
                               /* sine wavetable has a rate of modulation of about 0.15 Hz */
/* playing with the width, modulation rate, depth size  and feedback affects
   the intensity of the chorus'ed sound */


.segment /dm     dmchorus;

.var    IRQ1_counter = 0x00000004;
.var    IRQ2_counter = 0x00000004;

.var    feedback_gainL = afL;
.var    feedback_gainR = afR;
.var    sweep_rate = modulation_rate;
.var    sweep_widthL = chorus_width;          /* controls width of left sweep, ranges from -1 to - 15 */
.var    sweep_widthR = chorus_width;          /* controls width of right sweep, ranges from -1 to - 15 */

.var    w1[DepthL + 1];                       /* delay-line buffer 1, max delay = D1 */
.var    w2[DepthR + 1];                       /* delay-line buffer 2, max delay = D2 */
.var    excursion_valueL;                     /* wavetable offset value for delay calculation */
.var    excursion_valueR;                     /* wavetable offset value for delay calculation */
.var  random[WaveSize]="sinetbl.dat";         /* store one period of the wavetable */
                                              /* minimum frequency of table =
                                                 (freq of delay update)/WaveSize = 8/4 = 2 Hz */
.var    wavetbl_counter = 0x00000000;

.endseg;


/* -------------PROGRAM MEMORY CODE--------------------------------*/
.segment /pm pm_code;

Init_Chorus_Buffers:
       B2 = w1;  L2 = @w1;           /* left delay-line buffer pointer and length */
```

```
       m2 = 1;

       LCNTR = L2;                  /* clear left delay line buffer to zero */
        DO clrDlineL UNTIL LCE;
clrDlineL:         dm(i2, m2) = 0;

       B3 = w2;  L3 = @w2;          /* right delay-line buffer pointer and length */
       m3 = 1;

       LCNTR = L3;                  /* clear right delay line buffer to zero */
        DO clrDlineR UNTIL LCE;
clrDlineR:         dm(i3, m3) = 0;


       B6 = random;                 /* left channel pointer for signal generator */
       L6 = @random;                /* get size of table lookup */
       B7 = random;                 /* right channel pointer for signal generator */
       I7 = random + 1000;          /* offset 90 degrees so modulators in quadradure phase */
       L7 = @random;                /* get size of table lookup */

       RTS;

/* ---------------------------------------------------------------------------*/
/*                                                                            */
/*                    Digital Chorus Routine - process both channels together     */
/*                                                                            */
/* -------------------------------------------------------------------------- */

chorus_effect:
       /* combine both left and right input samples together into 1 signal */
       r0 = dm(Left_Channel);       /* left input sample */
       r1 = dm(Right_Channel);      /* right input sample */
       r0 = ashift r0 by -1;        /* scale signal by 1/2 for equal mix */
       r1 = ashift r1 by -1;        /* scale signal by 1/2 for equal mix */
       r15 = r0 + r1;               /* 1/2xL(n) + 1/2 xR(n) */

test_wav_update:
       /* update sine value from lookup table? Update every 80 SPORT rx interrupts */
       /* sweep frequency = 80 * c * 4000 / fs = 96000 /48k = .15 sec */
       r11 = DM(sweep_rate);        /* count up to 80 interrupts (default) */
       r10 = DM(wavetbl_counter);   /* get last count from memory */
       r10 = r10 + 1;               /* increment preset */
       comp (r10, r11);             /* compare current count to max count */
       if ge r10 = r10 - r10;       /* if count equals max, reset to zero and start over */
       DM(wavetbl_counter) = r10;   /* save updated count */

       r12 = pass r10;              /* test for wave count 0? */
       if eq jump update_wavetbl_ptrs;

       /* if you are here, reuse same random values for now */
       jump do_stereo_chorus;

       /* if necessary, calculate updated pointer to wavetables */
update_wavetbl_ptrs:
       m6 = c;                      /* desired increment c - frequency f = c x fs / D */
       r1 = DepthL;                 /* Total Delay Time */
       r2 = dm(i6, m6);             /* get next value in wavetable */
       r4 = dm(sweep_widthL);
       r2 = ashift r2 by r4;        /* divide by at least 2 to keep 1.31 #s between 0.5 and -0.5 */
       r3 = r1 * r2 (SSFR);         /* multiply Delay 1 by a fractional value from 0 to 0.5 */
       dm(excursion_valueL) = r3;

       m7 = c;                      /* desired increment c - frequency f = c x fs / D */
       r1 = DepthR;                 /* Total Delay Time */
       r2 = dm(i6, m6);             /* get next value in wavetable */
       r4 = dm(sweep_widthR);
       r2 = ashift r2 by r4;        /* divide by at least 2 to keep 1.31 #s between 0.5 and -0.5 */
       r3 = r1 * r2 (SSFR);         /* multiply Delay 1 by a fractional value from 0 to 0.5 */
       dm(excursion_valueR) = r3;

do_stereo_chorus:
       r3 = dm(excursion_valueL);   /* calculate time-varing delay for 2nd voice */
       r1 = L_Del_TapCenter;        /* center tap for delay line */
       r4 = r1 + r3;                /*   r4 = d(n) = D1/2 + D1 * random(fc*t)   */
```

```
process_left_ch:
       /* r4 now will be used to set m2 register to fetch time varying delayed sample */
       m2 = r4;                          /* tap outputs of circular delay line */
       modify(i2, m2);                   /* go to delayed sample */
       r4 = -r4;                         /* negate to get back to where we were */
       m2 = r4;                          /* used to post modify back to current sample */
       r9 = dm(i2, m2);                  /* get time-varying delayed sample 1 */

       /* r8 will be used to get Nominal Tap Center delayed sample for flange feedback */
       m2 = L_Del_TapCenter;             /* tap outputs of circular delay line */
       modify(i2, m2);                   /* go to delayed sample */
       m2 = -L_Del_TapCenter;            /* negate to get back to where we were */
       r7 = dm(i2, m2);                  /* get delayed sample, postmodify back to current sample */

       /* crank out difference equation */
       r8 = a0L;                         /* left input gain */
       mrf = r8 * r15 (SSF);             /*   mrf = a0L * x-input   */
       r8 = dm(feedback_gainL);          /* left gain for feedback of nominal tap center*/
       mrf = mrf - r8 * r7 (SSF);        /*   mrf = a0L * x - afL * sNominalL   */
       r12 = mr1f;                       /* save for input to chorus left delay line */

       r8 = a1L;                         /* left delay line output gain */
       mrf = mrf + r8 * r9 (SSFR);       /*  mrf = a0L * xL + a1L * s1L - afL * sNominalL   */
       mrf = SAT mrf;                    /* saturate result if necessary */
       r10 = mr1f;                       /* chorus result in r10 */

       /* put 'input minus feedback' sample from r12 into tap-0 of delay line */
       /* and backshift circular delay-line buffer pointer */
       dm(i2, -1) = r12;

       /* write chorus'ed output sample to left output channel */
       dm(Left_Channel)  = r10;          /* left output sample */

process_right_ch:
       r3 = dm(excursion_valueR);        /* calculate time-varing delay for 2nd voice */
       r1 = R_Del_TapCenter;             /* center tap for delay line */
       r4 = r1 + r3;                     /*   r4 = d(n) = D2/2 + D2 * random(fc*t)   */

       /* r4 now will be used to set m3 register to fetch time varying delayed sample */
       m3 = r4;                          /* tap outputs of circular delay line */
       modify(i3, m3);                   /* go to delayed sample */
       r4 = -r4;                         /* negate to get back to where we were */
       m3 = r4;                          /* used to post modify back to current sample */
       r9 = dm(i3, m3);                  /* get time-varying delayed sample 1 */

       /* r8 will be used to get Nominal Tap Center delayed sample for chorus feedback */
       m3 = R_Del_TapCenter;             /* tap outputs of circular delay line */
       modify(i3, m3);                   /* go to delayed sample */
       m3 = -R_Del_TapCenter;            /* negate to get back to where we were */
       r7 = dm(i3, m3);                  /* get delayed sample, postmodify back to current sample */

       /* crank out difference equation */
       r8 = a0R;                         /* left input gain */
       mrf = r8 * r15 (SSF);             /*   mrf = a0R * x-input   */
       r8 = dm(feedback_gainR);          /* gain for feedback of nominal tap center*/
       mrf = mrf - r8 * r7 (SSF);        /*   mrf = a0R * xR - afR * sNominalR   */
       r12 = mr1f;                       /* save for input to chorus right delay line */

       r8 = a1R;                         /* right delay line output gain */
       mrf = mrf + r8 * r9 (SSFR);       /*   mrf = a0R * x + a1R * s1R - af * sNominalR   */
       mrf = SAT mrf;                    /* saturate result if necessary */
       r10 = mr1f;                       /* chorus result in r10 */

       /* put 'input minus feedback' sample from r12 into tap-0 of delay line */
       /* and backshift circular delay-line buffer pointer */
       dm(i3, -1) = r12;

       /* write chorus'ed output sample to right output channel */
       dm(Right_Channel) = r10;          /* right output sample */

       rts;
```

3.3.2.3 Vibrato

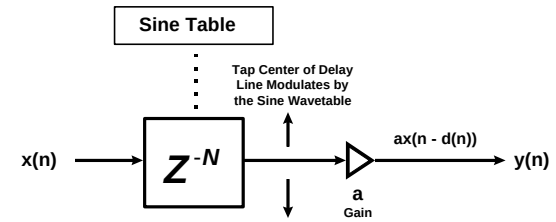
The **vibrato** effect that duplicates 'vibrato' in a singer's voice while sustaining a note, a musician bending a stringed instrument, or a guitarist using the guitars 'whammy' bar. This effect is achieved by evenly modulating the pitch of the signal. The sound that is produced can vary from a slight enhancement to a more extreme variation. It is similar to a guitarist moving the 'whammy' bar, or a violinist creating vibrato with cyclical movement of the playing hand. Some effects units offered vibrato as well as a tremolo. However, the effect is more often seen on chorus effects units.

The slight change in pitch can be achieved (with a modified version of the chorus effect) by varying the depth with enough modulation to produce a pitch oscillation. This is accomplished by changing the modify value of the delay-line pointer on-the-fly, and the value chosen is determined by a lookup table. This results in the interpolation/decimation of the stored samples via rotating the center tap of the delay line. The stored 'history' of samples are thus played back at a slower, or faster rate, causing a slight change in pitch.

To obtain an even variation in the pitch modulation, the delay line is modified using a sine wavetable. Note that this a stripped down of the chorus effect, in that the direct signal is not mixed with the delay-line output.

This effect is often confused with 'tremolo', where the *amplitude* is varied by a LFO waveform. The tremolo and vibrato can both be combined together with a time-varying LPF to produce the effect produced by a rotating speaker (commonly referred to a 'Leslie' Rotating Speaker Emulation). An example rotating speaker emulation effect is also shown in this section.

Figure 73.
Implementation of the Vibrato Effect



N = variable delay d(n)

Can use chorus or flanger algorithm, but no mix of input signal is required

Example Vibrato Implementation Using The ADSP-21065L

```
/* ------------------------------------------------------------------------------------------- */
/*                                                                                             */
/*                    Digital Vibrato Routine - process both channels together                 */
/*                                                                                             */
/* ------------------------------------------------------------------------------------------- */

vibrato_effect:
       /* combine both left and right input samples together into 1 signal */
       r0 = dm(Left_Channel);              /* left input sample */
       r1 = dm(Right_Channel);             /* right input sample */
       r0 = ashift r0 by -1;               /* scale signal by 1/2 for equal mix */
       r1 = ashift r1 by -1;               /* scale signal by 1/2 for equal mix */
       r2 = r0 + r1;                       /* 1/2xL(n) + 1/2 xR(n) */

test_sine_update:
       /* update sine value from lookup table? Update every 12 SPORT rx interrupts */
       /* sweep rate = 12 * sin_inc * 4000 / fs = 48000 /48k = 1 sec */
       r11 = DM(modulation_rate);          /* count up to 24 interrupts */
       r10 = DM(wavetbl_counter);          /* get last count from memory */
       r10 = r10 + 1;                      /* increment preset */
       comp (r10, r11);                    /* compare current count to max count */
       if ge r10 = r10 - r10;              /* if count equals max, reset to zero and start over */
       DM(wavetbl_counter) = r10;          /* save updated count */
```

```
       r12 = pass r10;                      /* test for wave count 0? */
       if eq jump update_sinetbl_ptr;

       jump do_vibrato;                     /* if you are here, reuse same sine value for now */
                                            /* dm(sine_value) remains unchanged */

       /* if necessary, calculate updated pointer to sine wavetable */
update_sinetbl_ptr:
       m6 = dm(sin_inc);                    /* desired increment sin_inc - f = sin_inc x fs / D */
       r6 = D;
       r7 = dm(i6, m6);
       r4 = dm(pitch_bend);
       r7 = ashift r7 by r4;                /* divide by 2 to keep #s between 0.5 and -0.5 */
       r8 = r6 * r7 (SSFR)                  /* delay multiplication factor */
       dm(sine_value) = r8;                 /* save to current sine value to be reused above */

do_vibrato:
       r3 = dm(sine_value);                 /* get previous or newly updated sine value */
       r1 = D2;                             /* get nominal tap center delay */
       r4 = r1 + r3;                        /* r4 = d(n) = D/2 + D * sin(fc*t) */

       /* r4 now will be used to set m2 register to fetch time varying delayed sample */
       m2 = r4;                             /* set tap valud for output of circular delay line */
       modify(i2, m2);                      /* go to delayed sample address */
       r4 = -r4;                            /* negate to get back to where we were */
       m2 = r4;                            /* set up to go back to current mem location after access */
       r10 = dm(i2, m2);                    /* get delayed sample */

       /* put input sample from r2 into tap-0 of delay lines */
       /* and backshift pointer & update circular delay-line buffer*/
       dm(i2, -1) = r2;

       /* write vibrato output sample to AD1819A DAC channels */
       dm(Left_Channel)  = r10;             /* left output sample */
       dm(Right_Channel) = r10;             /* right output sample */

       rts;
```

21065L Rotating Speaker Emulation Implementation (Vibrato & Tremolo Combo)

```
/***   ROTATING_SPEAKER.ASM   ************************************************************
*                                                                                          *
*       AD1819A/ADSP-21065L EZLAB Rotating Speaker Emulation Effect Program                *
*       Developed using ADSP-21065L EZ-LAB Evaluation Platform                             *
*                                                                                          *
*       I/O Equation:  y(n) = [x(n)*sin(2*pi*fc*t)] convolve [ x(n-d(n)) ]                 *
*                                                                                          *
*                                                                                          *
*                                          ^                ^                              *
*              sin(2*PI*fc*t)             /                /                               *
*                    |                 _____/_____       ___/____                         *
*                    |                |           |     |        |                        *
*                    v                |   Z^(-D)  |     |  LPF   |                        *
*       x(n) -------->O--------------->|           |-----|  H(z)  |---> y(n)               *
*                    X                |    d(n)   |     |        |                        *
*                                     |           |     |________|                        *
*                                     |___________|     /  Variable                       *
*                                        / Rotating Tap  /   Cutoff                        *
*                                       /   Center          Freq                           *
*                                      /                                                   *
*                                                                                          *
*                                                                                          *
*                                                                                          *
*       What the rotating speaker effect does?                                             *
*       --------------------------------------                                             *
*       A popular effect used on keyboards and Hammond organs. This effect                 *
*       consists of a combination of the tremolo and vibrato effects, combined             *
*       a low-pass filter with a variable cutoff frequency.  This combination              *
*       simulates what was done by audio engineers who would actually produce              *
*       this effect by rotating a speaker cabinet.  The resulting sound heard              *
*       by the ears is a cyclical increase/decrease in volume, a doppler effect            *
*       from the speaker rotating, and a cyclical drop in high frequencies                 *
```

```
*       whenever the speaker is facing away from the listener.                          *
*                                                                                       *
****************************************************************************************/

/* ADSP-21065L System Register bit definitions */
#include       "def21065l.h"
#include       "new65Ldefs.h"

.EXTERN        Left_Channel;
.EXTERN        Right_Channel;

.GLOBAL        Init_Tremolo_Vibrato_Buffers;
.GLOBAL        Rotating_Speaker_Effect;
.GLOBAL        change_speaker_rotate_rate;
.GLOBAL        select_tremolo_effect;

.segment /dm    rotspeak;

#define WaveTableSize  4000          /* sinusoidal wavetable, 1 period in table of 4000 sine wave
elements  */
#define D              2000          /* Depth, or TD = D/fs = 1000/48000 = 20.83 msec */
                                     /* increasing depth size D increases pitch variation */
#define Depth          D          /* Depth is equivalent to time delay of a signal, or delay-line size */
#define D2             Depth/2       /* D/2, used for addig & subtracting delay from tap center */


.var    IRQ1_counter = 0x00000003;
.var    IRQ2_counter = 0x00000000;
.var    wavetbl_counter = 0x00000000;
.var    Effect_Ctrl = 0x00000001;     /* memory flag that determines which tremolo routine executes */
.var    pitch_bend = -6;              /* Enter # from -1 to -15.  Do not enter 0.*/
                                      /* -1 to -5  - large pitch bend*/
                                      /* -6 to -10 - medium pitch bend */
                                      /* -11 to -15 - small pitch bend */
.var    sin_inc = 2;                  /* wavetable signal frequency ftable = sin_inc * fs = ? Hz */
.var    modulation_rate = 3;          /* controls how fast the wavetable is updated in the SPORT1 rx ISR
*/
.var    sine_value;                   /* used for tremolo control */
.var    excursion_value;              /* used for vibrato delay offset calculation */
.var     sine[WaveTableSize] = "sinetbl.dat";
.var    w[D + 1];                     /* delay-line buffer, max delay = D */

.endseg;

/*-------------------------------------------------------------------------*/

.segment /pm pm_code;

/*-----------------------------------------------------------------------------------*/
Init_Tremolo_Vibrato_Buffers:
       B2 = w;  L2 = @w;              /* delay-line buffer pointer and length */
       m2 = 1;

       LCNTR = L2;                    /* clear delay line buffer to zero */
        DO clrDline UNTIL LCE;
clrDline:         dm(i2, m2) = 0;

       B6 = sine;                    /* pointer for signal generator */
       L6 = @sine;                   /* get size of sine table lookup */

       RTS;

/*****************************************************************************************

                     ROTATING SPEAKER (Vibrato & Tremolo Combo) AUDIO EFFECT

*****************************************************************************************/

Rotating_Speaker_Effect:
       r1 = DM(Left_Channel);
       r1 = ashift r1 by -1;
       r2 = DM(Right_Channel);
       r2 = ashift r2 by -1;
       r3 = r2 + r1;
```

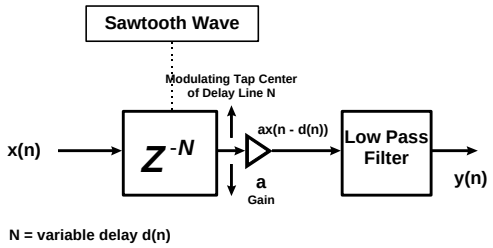
```
       /* generate sine value from wavetable generator if necessary, where r4 = sin(2*pi*fc*t)   */
test_wavtbl_update:
       /* update sine value from lookup table? Update every 80 SPORT rx interrupts */
       /* sweep frequency = 80 * c * 4000 / fs = 96000 /48k = .15 sec */
       r11 = DM(modulation_rate);           /* count up to 80 interrupts */
       r10 = DM(wavetbl_counter);           /* get last count from memory */
       r10 = r10 + 1;                       /* increment preset */
       comp (r10, r11);                     /* compare current count to max count */
       if ge r10 = r10 - r10;               /* if count equals max, reset to zero and start over */
       DM(wavetbl_counter) = r10;           /* save updated count */

       r12 = pass r10;                              /* test for wave count 0? */
       if eq jump update_wavetbl_ptr;

       /* if you are here, reuse same sine value for now.. dm(sine_value) remains unchanged */
       jump do_vibrato;

       /* if necessary, calculate updated pointer to wavetable */
update_wavetbl_ptr:
       m6 = dm(sin_inc);                    /* desired increment sin_inc - frequency f = sin_inc x fs /
D */
       r7 = dm(i6, m6);                     /* get next value from sine lookup table */
       dm(sine_value) = r7;                 /* use for tremolo_effect amplitude scaling factor */
       r4 = dm(pitch_bend);                 /* controls scaling of center tap delay offset */
       r7 = ashift r7 by r4;                /* divide by at least 2 to keep #s between 0.5 and -0.5 */
       r6 = D;                              /* Total Delay Time D = amplitude of sine wave lookup */
       r8 = r6 * r7 (SSFR);                 /* delay multiplication factor */
       dm(excursion_value) = r8;            /* save to current sine value to be reused above */

do_vibrato:
       r2 = dm(excursion_value);            /* get previous or newly updated scaled sine value */
       r1 = D2;                             /* get nominal tap center delay */
       r4 = r1 + r2;                        /* r4 = d(n) = D/2 + D * sin(fc*t) */

       /* r4 now will be used to set m2 register to fetch time varying delayed sample */
       m2 = r4;                             /* set tap valud for output of circular delay line */
       modify(i2, m2);                              /* go to delayed sample address */
       r4 = -r4;                            /* negate to get back to where we were */
       m2 = r4;                             /* set up to go back to current location after memory access
*/
       r10 = dm(i2, m2);                    /* get delayed sample */

       /* write vibrato output sample to r0 for tremolo routine */
       r0 = r10;

       /* put input sample from r2 into tap-0 of delay lines */
       /* and backshift pointer & update circular delay-line buffer*/
       dm(i2, -1) = r3;

which_tremolo_routine:
       r4 = DM(Effect_Ctrl);
       r4 = pass r4;
       if eq jump mono_tremolo_effect;              /* if == 1, execute mono tremolo routine */
                                            /* otherwise, execute stereo tremolo routine */
stereo_tremolo_effect:
       /* get generated sine value from wavetable generator, where r4 = sin(2*pi*fc*t)   */

       r4 = dm(sine_value);
       r5 = r0 * r4 (SSFR);
       /* test current sine value to pan left or right, if + then pan left, if - then pan right */
       r4 = pass r4;
       IF LE JUMP (pc, pan_right_channel);

pan_left_channel:
       /* write tremolo result sample to left/right output channels */
       DM(Left_Channel) = r5;
       r6 = 0x00000000;
       DM(Right_Channel) = r6;
       JUMP (pc, tremolo_done);

pan_right_channel:
       /* write tremolo result sample to left/right output channels */
       r6 = 0x00000000;
       DM(Left_Channel) = r6;
       DM(Right_Channel) = r5;
```

```
tremolo_done:
        rts;

/* ------------------------------------------------------------------------------- */

mono_tremolo_effect:
        /* get generated sine value from wavetable generator, where r4 = sin(2*pi*fc*t)  */
        r4 = dm(sine_value);
        r5 = r0 * r4 (SSFR);

        /* write tremolo result sample to left/right output channels */
        dm(Left_Channel)  = r5;                     /* left output sample */
        dm(Right_Channel) = r5;                     /* right output sample */

        RTS;

/* ------------------------------------------------------------------------------- */
```

3.3.2.4 Pitch Shifter

An interesting and commonly used effect is changing the pitch of an instrument or voice. The algorithm that can be used to implement a pitch shifter is the chorus or vibrato effect. The chorus routine is used if the user wishes to include the original and pitch shifted signals together. The vibrato routine can be used if the desired result is only to have pitch shifted samples, which is often used by TV interviewers to make an anonymous persons voice unrecognizable. The only difference from these other effects is the waveform used for delay line modulation. The pitch shifter requires using a sawtooth/ramp wavetable to achieve a 'linear' process of dropping and adding samples in playback from an input buffer. The slope of the sawtooth wave as well as the delay line size determines the amount of pitch shifting that is performed on the input signal.

Figure 74.
Implementation of a Generic Pitch Shifter



The audible side effect of using the 2 instrument chorus algorithm (with one delay line) is the ‘clicks’ that are produced whenever the delay pointer passes the input signal pointer when samples are added or dropped. This is because output pointer is moving through the buffer at a faster/slower rate than the input pointer, thus eventually causing an overlap. To reduce or eliminate this undesired artifact cross-fading techniques can be used between two alternating delay line buffers with a windowing function, so when one of delay line output pointers are close to the input, a zero crossing will occur at the overlay to avoid the 'pop' that is produced. For higher pitch shifted values, there is a noticable 'warble' audio modulation produced as a result of the outputs of the delay lines being out of phase, which causes periodic cancellation of frequencies to occur. Methods to control the delay on-the-fly to prevent the phase cancellations have been proposed, but are not implemented in our reference examples. A basic 21065L assembly example of the pitch shifter used as a Detune Effect is shown in the next section.

Again, the DSP timer can update the delay-line retrieval address value of a previous sample which results in a linear adding and dropping of samples. Using a positive or negative slope determines if the pitch of an audio signal will be shifted up or down (see diagram below). The same look-up table can be used to pitch shift up or down. The address generation unit only needs to use an increment or decrement modify register that will move forward or backwards in the table. Multiple harmonies can be created by having multiple pointers with positive and negative address modify values circling through the table.

Figure 75.
Example Two-Voice Pitch Shifter Implementation

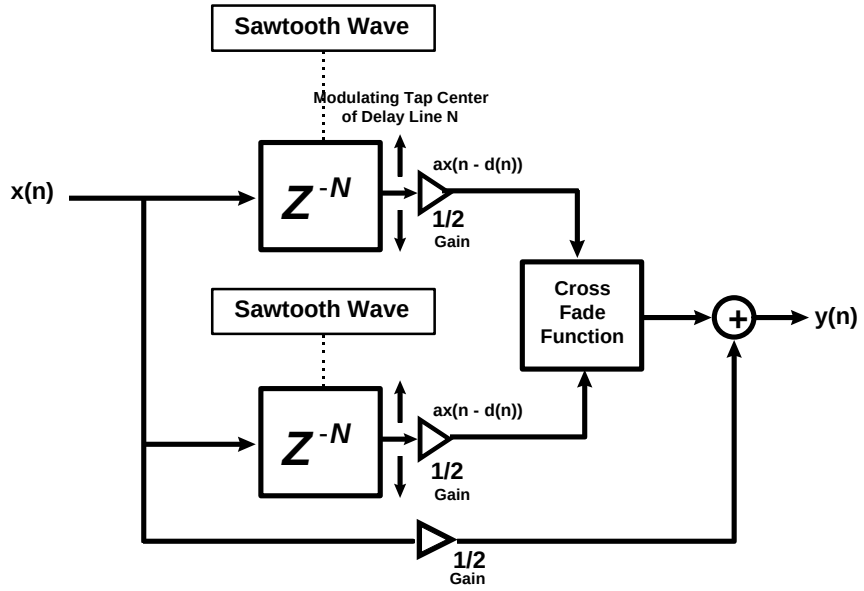
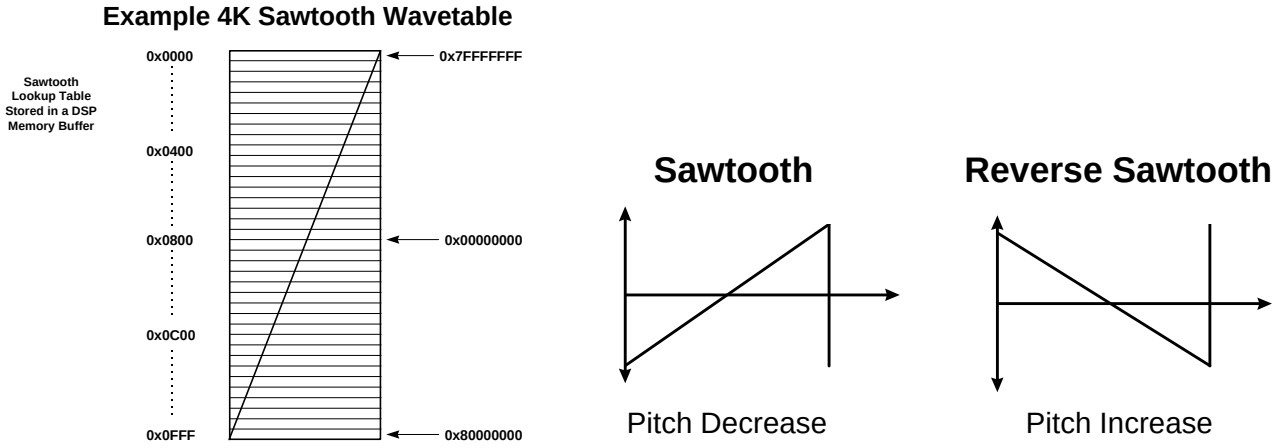


Figure 76.



3.3.2.5 Detune Effect

The Detune Effect is actually a version of the Pitch Shifter. The pitch shifted result is set to vary from the input by about +/- 1% of the input frequency. This is done by setting the Pitch Shift factor to 0.99 or 1.01 The effect's result is to increase or decrease the output and combine the pitch shift with the input to vary a few Hz, resulting in an 'out of tune effect'. (The algorithm actually uses a version of the chorus effect with a sawtooth to modulate the delay-line). Small pitch scaling values produce a 'chorus like' effect and imitates two instruments slightly out of tune. This effect is useful on vocal tracks to give impression of 2 musicians singing the same part using 1 person's voice. The pitch shifting result is to small for the formant frequencies of the vocal track to be affected, so the shifted voice still sounds realistic.

For a strong Detune Effect, vary the pitch by 5-10 Hz
For a weak Detune Effect ('Sawtooth Chorus' sound), vary the pitch by 2-3 Hz

21065L Example Detune Effect - Slight Pitch Shift Variation of an Input Signal

```
/* ********************************************************************************************
       DETUNE_EFFECT.ASM - DETUNE.ASM - Pitch Shift Effect Simulating 2 Voices Slightly Out Of Pitch
       (Timer0 Update Method - delay calculation determined by the on-chip programmable timer)

   x(n) ----------------------------------------|>------------------------------>O----> y(n)
                |                              ^       1/2                             + ^
                |                             /                                        |
                |                  _________/______                                   |
                |                 |                |                                  |
                |                 |     Z^(-D)     |                                  |
                |-----------------|>               |-----------|>--------             |
                |                 |     d1(n)      |           1/2        |           |
                |                 |                |                      |           |
                |                 |________________|                      |           |
                |                     /                              +  v             |
                |                    /                            |-------------|     |
                |                   / Linear Incrementing Tap     |  Cross Fade |     |
                |                     via Sawtooth Wavetable      |  Windowing  |-------|
                |                                                 |  Functions  |
                |                                                 |-------------|
                |                                                     +  ^
                |                                ^                    |
                |                               /                     |
                |                  _________/____                     |
                |                 |              |                    |
                |                 |     Z^(-D)   |                    |
                |-----------------|>             |-----------|>-------|
                                  |     d2(n)    |           1/2
                                  |              |
                                  |______________|
                                      /
                                     /
                                    / Linear Incrementing Tap
                                      via Sawtooth Wavetable


********************************************************************************************** */

/* ADSP-21060 System Register bit definitions */
#include       "def21065l.h"
#include       "new65Ldefs.h"

.GLOBAL        pitch_shifter;
.GLOBAL        Init_Pitch_Buffers;
.GLOBAL        Timer0_Initialization;
.GLOBAL        wavetable_gen;
.EXTERN        Left_Channel;
.EXTERN        Right_Channel;

/* pitch shift filter coefficients */
#define a0             0x40000000            /* One Half, or 0.5 in 1.31 format */
#define a1             0x40000000            /* One Half, or 0.5 in 1.31 format */
#define a2             0x40000000            /* One Half, or 0.5 in 1.31 format */
/* delay buffer and wavetable definitions */
/* increasing depth size D increases pitch variations */
#define D              2000           /* Depth, or TD = D/fs = 1500/44100 = 35 msec */
#define D2             D/2
#define WaveSize       4000           /* triangular wavetable size*/
#define WinSize        4000           /* window function for cross-fading delay-lines */

/* pitch shift control parameters */
#define c              2              /* signal frequency fc1 = c1 * freqtimer = 4 Hz */
#define modulation_rate 50000         /* # of DSP cycles between Timer Expire ISR update */
#define pitch_depth    1              /* Enter # from 1 to 10.  DO NOT ENTER '0'!, to keep #'s
                                         between +/- 0.5 in fractional format */
                                      /* The lower the number, the greater the pitch bend */

/* playing with the pitch depth, modulation rate, and pitch-depth size affects
   the intensity of the pitch shifted sound,  the pitch-depth setting is strongest
   at lower numbers 1 to 3, it is less intense for 4 to 7.*/

{sawtooth chorus - modrate=50000, pitchdepth=1, c=1,  D=1000)
{detune effect  - modrate=50000, pitchdepth=1, c=2,  D=1000}
```

```
.segment /dm   delaylin;

.var         w1[D + 1];              /* delay-line buffer, max delay = D */
.var         w2[D + 1];              /* delay-line buffer, max delay = D */

.endseg;

.segment /dm    dmpitch;

.var         saw_value1;             /* wavetable update via timer for delay calculation */
.var         saw_value2;
.var         volume_dline1;
.var         volume_dline2;
.var         window_fnc[WinSize] = "triang_window.dat";  /* load window function for crossfade control */
.var        sawtooth_wav[WaveSize] = "sawtooth.dat";       /* load one period of the wavetable */
                                        /* min frequency f1 = fs/Ds = 8/4 = 2 Hz */
.endseg;


/* -------------PROGRAM MEMORY CODE--------------------------------*/
.segment /pm pm_code;

Init_Pitch_Buffers:
       B2 = w1;  L2 = @w1;              /* delay-line 1 buffer pointer and length */
       m2 = 1;

       LCNTR = L2;                      /* clear delay line buffer to zero */
        DO clrDline1 UNTIL LCE;
clrDline1:         dm(i2, m2) = 0;

       B3 = w2;  L3 = @w2;              /* delay-line 2 buffer pointer and length */
       m3 = 1;

       LCNTR = L3;                      /* clear delay line buffer to zero */
        DO clrDline2 UNTIL LCE;
clrDline2:         dm(i3, m3) = 0;

       B6 = sawtooth_wav;               /* pointer 1 for sawtooth signal generator */
       L6 = @sawtooth_wav;              /* get size from sawtooth number table lookup */
       B7 = sawtooth_wav;               /* pointer 2 for sawtooth signal generator */
       I7 = sawtooth_wav + 2000;        /* start in middle of table */
       L7 = @sawtooth_wav;              /* get size from sawtooth number table lookup */

       B4 = window_fnc;                 /* pointer for crossfade window for delay-line 1 */
       L4 = @window_fnc;                /* get length of window buffer */
       B5 = window_fnc;                 /* pointer for crossfade window for delay-line 2 */
       I5 = window_fnc + 2000;                  /* start in middle of table */
       L5 = @window_fnc;                /* get length of window buffer */

       RTS;

/*---------------------------------------------------------------------*/

/* Set up timer for the Chorus Effects wavetable generator */

Timer0_Initialization:
       bit clr mode2 TIMEN0;            /* timer off initially */
       bit set mode2 PWMOUT0 | PERIOD_CNT0 | INT_HI0;      /* latch timer0 to high priority timer int */

       r0 = modulation_rate;
       DM(TPERIOD0) = r0;
        DM(TCOUNT0) = r0;               /* assuming 16.7 nSec cycle @ 60 MIPs */
       r0 = 10;
       DM(TPWIDTH0) =r0;

       bit set imask TMZHI;             /* timer high priority  */
       bit set mode2 TIMEN0;            /* timer on */

       rts;


/* ------------------------------------------------------------------------------------------  */
/*                                                                                             */
/*                  Wavetable Generator used for Pitch Shift Delay Line Modulation             */
```

```
/*                                                                                                  */
/* ------------------------------------------------------------------------------------------------ */
/* High Priority Timer Interrupt Service Routine for Delay Line Modulation of Chorus Buffers */
/* This routine is a wavetable generator, where r3 = D2 * sin(2*pi*fc*t)   */
/* and it modulates the chorus delay line around rotating tap center */

wavetable_gen:
       bit set mode1 SRRFL;
       nop;                                /* 1 cycle latency writing to Mode1 register */

       /* c = desired wavetable increment (DAG modifier), where frequency f = c x fs / D */

sawtooth1:
       r1 = D;                             /* Total Delay Time */
       r2 = dm(i6, c);                     /* get next value in wavetable */
       r2 = ashift r2 by -pitch_depth;     /* divide by at least 2 to keep 1.31 #s between 0.5/-0.5 */
       r3= r1 * r2 (SSFR);                 /* multiply Delay by a fractional value from 0 to 0.5 */
       dm(saw_value1) = r3;                /* store to memory for chorus routine */

sawtooth2:
       r1 = D;                             /* Total Delay Time */
       r2 = dm(i7, c);                     /* get next value in wavetable */
       r2 = ashift r2 by -pitch_depth;     /* divide by at least 2 to keep 1.31 #s between 0.5/-0.5 */
       r3 = r1 * r2 (SSFR);                /* multiply Delay by a fractional value from 0 to 0.5 */
       dm(saw_value2) = r3;                /* store to memory for pitch shift routine */

/* determine cross-fade gain control values for both delay-line buffers used in pitch shift routine */
/* scaling factor will be 0x00000000 whenever the center tap crosses input of the delay line buffer */

triangl_window_value1:
       r1 = dm(i4, c);                           /* corresponds with sawtooth 1 */
       dm(volume_dline1) = r1;

triangl_window_value2:
       r1 = dm(i5, c);                           /* corresponds with sawtooth 2 */
       dm(volume_dline2) = r1;

       bit clr mode1 SRRFL;
       rti;

/* ------------------------------------------------------------------------------------------------ */
/*                                                                                                  */
/*                    Digital Pitch Shifter Routine - process right channel only                    */
/*                                                                                                  */
/* ------------------------------------------------------------------------------------------------ */

pitch_shifter:
       r15 = DM(Right_Channel);            /* get x-input, right channel */

       r3 = dm(saw_value1);                /* calculate time-varing delay for 2nd voice */
       r1 = D2;                            /* center tap for delay line 1*/
       r4 = r1 + r3;                       /*   r4 = d(n) = D/2 + D * random(fc*t)  */

       r5 = dm(saw_value2);                /* calculate time-varing delay for 3nd voice */
       r2 = D2;                            /* center tap for delay line 2 */
       r6 = r2 + r5;                       /*   r6 = d(n) = D/2 + D * random(fc*t)  */

       r8 = a0;                            /* input gain */
       mrf = r8 * r15 (SSF);               /*   mrf = a0 * x-input  */

       m2 = r4;                            /* tap outputs of circular delay line */
       modify(i2, m2);                     /* go to delayed sample */
       r4 = -r4;                           /* negate to get back to where we were */
       m2 = r4;                            /* used to post modify back to current sample */
       r9 = dm(i2, m2);                    /* get time-varying delayed sample 1 */
       r11 = dm(volume_dline1);            /* get scaling factor */
       r9 = r9 * r11 (SSF);                /* multiply by delay line output */

       r8 = a1;                            /* delay-line 1 gain */
       mrf = mrf + r8 * r9 (SSF);          /*   mrf = a0 * x + a1 * s1  */

       m3 = r6;                            /* tap outputs of circular delay line */
       modify(i3, m3);                     /* go to delayed sample */
       r6 = -r6;                           /* negate to get back to where we were */
       m3 = r6;                            /* used to post modify back to current sample */
```

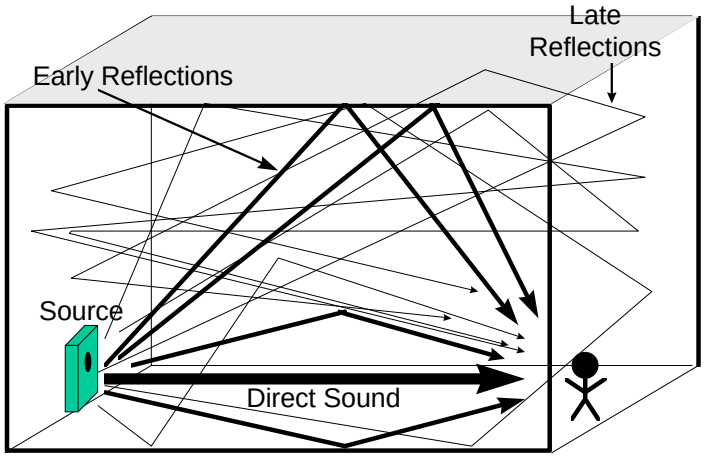
```
        r10 = dm(i3, m3);                       /* get time-varying delayed sample 2 */
        r11 = dm(volume_dline2);
        r10 = r10 * r11 (SSF);

        r9 = a2;                                /* delay-line 2 gain */
        mrf = mrf + r9 * r10 (SSFR);            /*   mrf = a0 * x + a1 * s1 + a2 + s2  */
        mrf = SAT mrf;                          /* saturate result if necessary */
        r10 = mr1f;                             /* pitch shifted result in r10 */


        /* put input sample from r15 into tap-0 of delay lines */
        dm(i2, 0) = r15;
         dm(i3, 0) = r15;

        /* backshift pointers & update circular delay-line buffers */
        modify(i2, -1);
        modify(i3, -1);

        /* write pitch shifted output sample to left/right output channels */
        DM(Left_Channel)=r10;
        DM(Right_Channel)=r10;

        rts;

.endseg;
```

3.3.3 Digital Reverberation Algorithms for Simulation of Large Acoustic Spaces

Reverberation is another time-based effect. More complex processing than echoing, chorusing or flanging, reverberation is often mistaken with delay or echo effects. Most multi-effects processing units provide a variation of both effects.

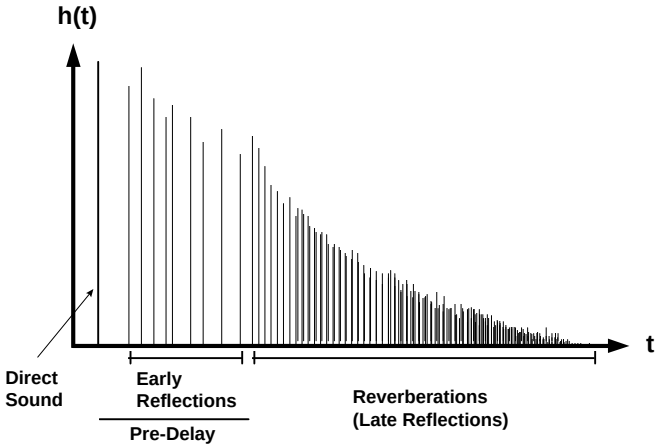
The first simulation reverb units in the 60's and 70's consisted of using a mechanical spring or plate attached to a transducer and passing the electrical signal through. Another transducer at the other end converted the mechanical reflections back to the output transducer. However, this did not produce realistic reverberation. M.A Schroeder and James A. Moorer developed algorithms for producing realistic reverb using a DSP.

Figure 78.
Reverberation of Large Acoustic Spaces



The reverb effect simulates the effect of sound reflections in a large concert hall or room (Figure 78). Instead of a few discrete repetitions of a sound like a multi-tap delay effect, the reverb effect implements many delayed repetitions so close together in time that the ear cannot distinguish the differences between the delays. The repetitions are blended together to sound continuous. The sound source goes out in every direction from the source, bounces off the walls and ceilings and returns from many angles with different delays. Reverberation is almost always present in indoor environments, and the reflections are greater for hard surfaces. As Figure 12 below shows, Reverberated Sound is classified as three components: Direct Sound, Early reflections and the Closely Blended Echos (Reverberations) [11,12,14].

Figure 79.
Impulse Response For Large Auditorium Reverberations



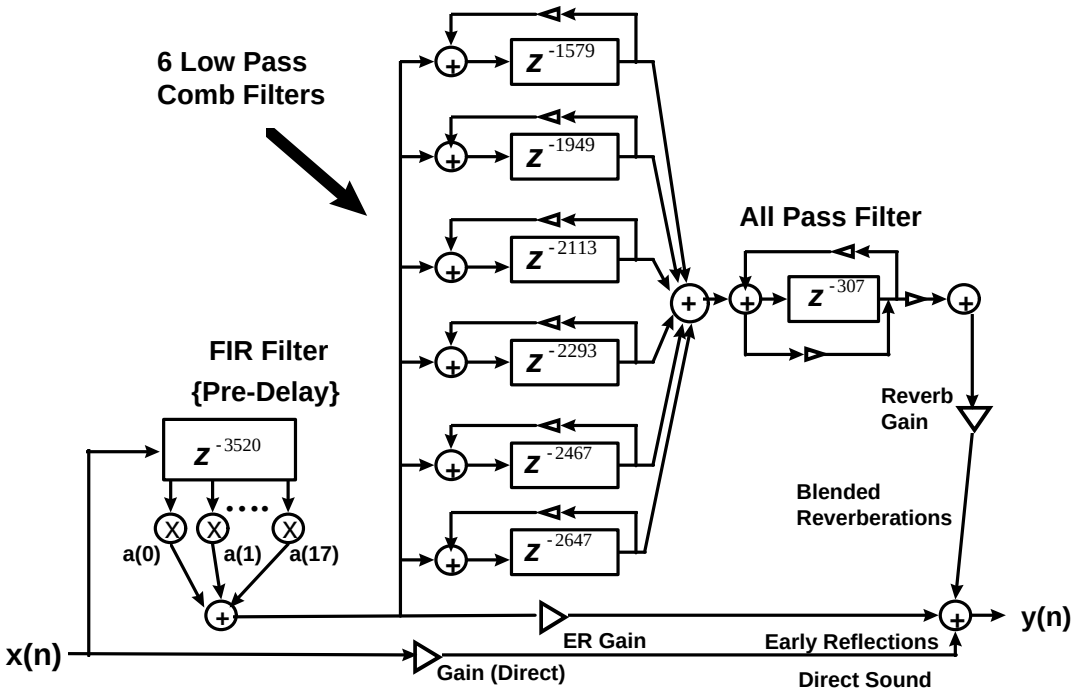
- **Direct Sound** - directly reaches the listener from the sound source.
- **Early reflections -** early echos which arrive within 10 ms to 100 ms by the early reflections of surfaces after the direct sound.
- **Closely Blended Echos** - is produced after 100 ms early reflections.

Figure 79 shows an impulse response of a large acoustic space, such as an auditorium or gymnasium. In a typical large auditorium, the first distinct delay responses that the user will hear are termed 'early reflections'. These early reflections are a few relatively close echos that actually occur in as reverberation in large spaces. The early reflections are the result of the first bounce back of the source by surfaces that are nearby. Next come echos which follow one another at such small intervals that the later reflections are no longer distinguishable to the human ear. A Digital Reverb typically will process the input through multiple delayed filters and add the result together with early reflection computations. Various parameters to consider in the algorithm would be the decay time (time it takes for reverb to decay) , presense (dry signal output vs. reverberations), and tone control (bass or treble) of the output reverberations.

M.A. Schroeder suggested 2 ways for producing a more realistic sounding reverb. The first approach was to implement 5 allpass filters cascaded together. The second way was to use 4 comb filters in parallel, summing their outputs, then passing the result through 2 allpass filters in cascade.

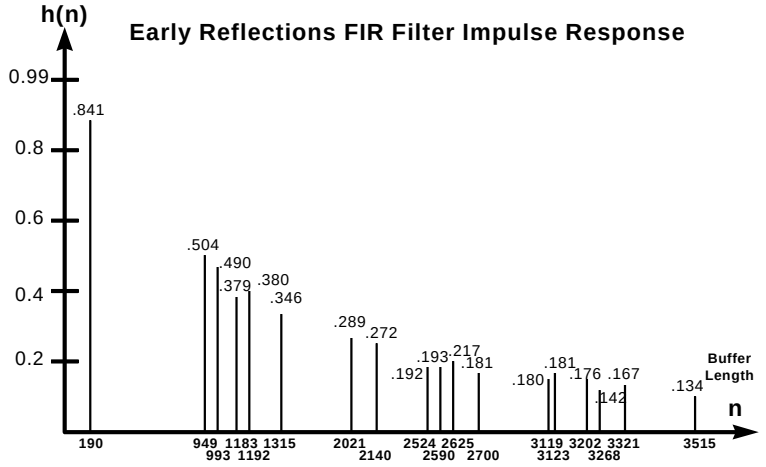
James A. Moorer expanded on Schroeder's research. One drawback to the Schroeder Reverb is that the high frequencies tend to reverberate longer than the lower frequencies. Moorer proposed using a *low pass comb filter* for each reverb stage to enlarge the density of the response. He demonstrated a technique involving 6 parallel comb filters with a low pass output, summing their outputs and then sending the summed result to an allpass filter before producing the final result. Moorer also recommended including the simulation of the early reflections common in concert halls using a tapped-delay FIR filter structure, along with the reverb filters for a more realistic response. Some initial delays can be added to the input signal by using an FIR filter ranging from 0 to 80 milliseconds. Moorer chose appropriate filter coefficients to produce 19 early reflections. Moorer's reverberator produced a more realistic reverb sound than Schroeder's, but still produces a rough sound for impulse signals such as drums.

Figure 80.
James A. Moorer's Digital Reverberation Structure



The figure above shows the structure for J.A Moorer's Digital Reverb [11] algorithm for a large auditorium response assuming a 44.1 kHz sampling rate. Moorer demonstrated a technique involving 6 parallel comb filters with a low pass output, summing their outputs and then sending the summed result to an all-pass filter before producing the final result. For realistic sounding reverberation, the DSP requires the use of large delay lines for both the comb filter and early reflection buffers. Each comb filter incorporates a different length delay-line. The reverberation delay time depends on the length the delay-line buffer sizes and the sampling rate. Fine tuning of input and feedback for each comb filter gain and delay-line values vary the reverberation effect to provide a different room size characteristic. Since all are programmable, the decay response can be modified on the fly to change the amount of the reverb effect.

Figure 82.



Example Reverb Specifications for a Large Auditorium response at 44.1 kHz Sampling Rate

Delay Line Buffer Length		*Time Delay*
Comb 1	*1759*	*40 ms*
Comb 2	*1949*	*44 ms*
Comb 3	*2113*	*48 ms*
Comb 4	*2293*	*52 ms*
Comb 5	*2467*	*56 ms*
Comb 6	*2647*	*60 ms*
Early Reflections	*3520*	*80 ms*
All-Pass Filter	*307*	*7 ms*

Table 6.

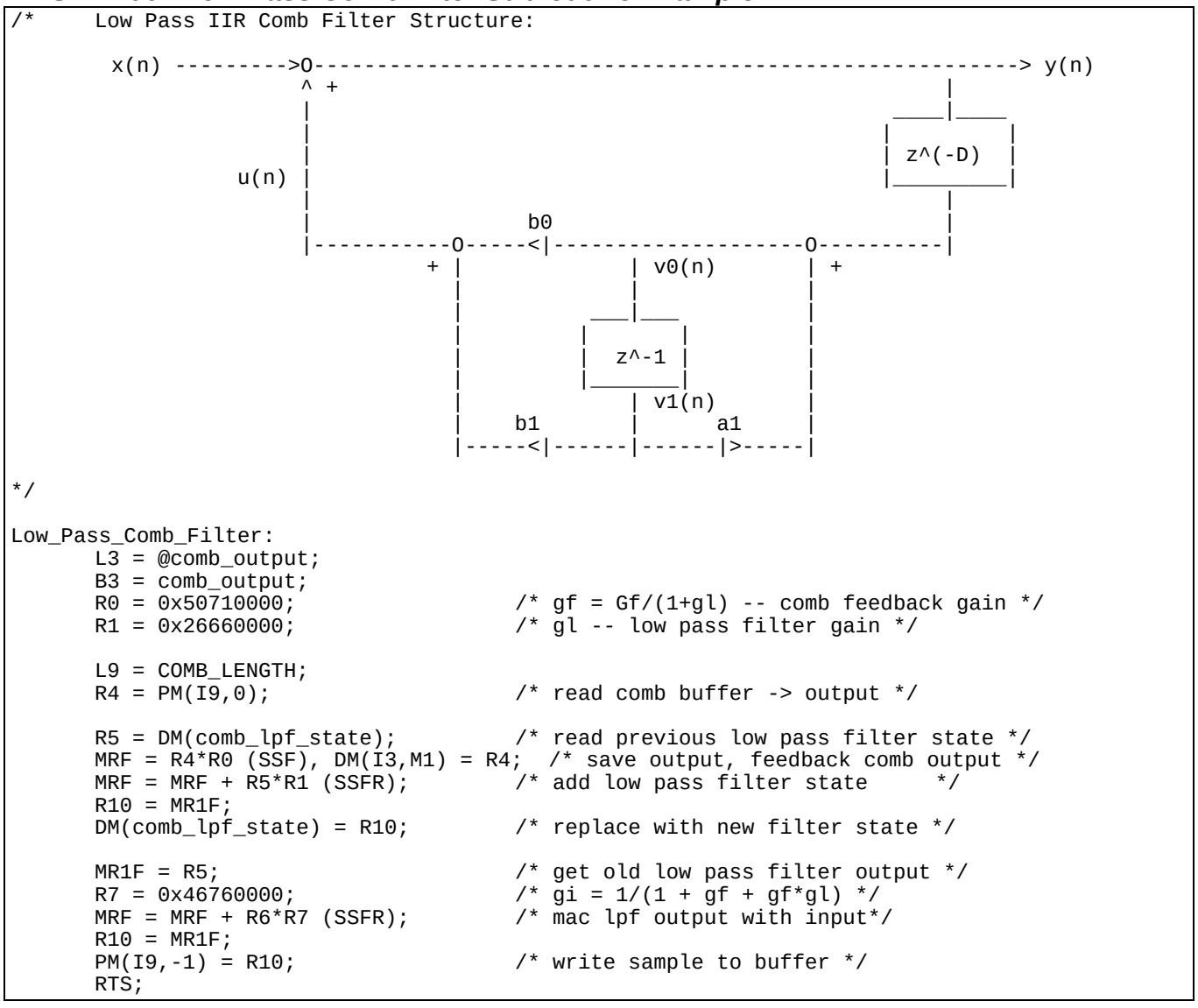
Early Reflection Delay Tap Lengths	Early Reflection Tap Gain Parameters Fractional 1.15 Representation
-190	0.841 = 0x6BA6
-759	0.504 = 0x4083
-44	0.490 = 0x3ED9
-190	0.379 = 0x3083
-9	0.380 = 0x30A4
-123	0.346 = 0x2C4A
-706	0.289 = 0x24FE
-119	0.272 = 0x22D1
-384	0.192 = 0x1893
-66	0.193 = 0x18B4
-35	0.217 = 0x1BC7
-75	0.181 = 0x172B
-419	0.180 = 0X170A
-4	0.181 = 0x172B
-79	0.176 = 0x1687
-66	0.142 = 0x122D
-53	0.167 = 0x1560
-194	0.134 = 0x1127

Reverb Building Blocks Low Pass Comb Filter and All Pass Filter Structures

For realistic sounding reverberation, the DSP requires the use of large delay lines for both the comb filter and early reflection buffers. The comb filter is used to increase echo density and give the impression of a sizable acoustic space. Each comb filter incorporates a different length delay line. Each delay line can be tuned to a different value to provide a different room size characteristic. Fine tuning of input and feedback gains for each comb filter gain and comb filter delay-line sizes will vary the reverberation response. Since these parameters are programmable, the decay response can be modified on the fly to change the amount of the reverb effect for simulation of a large hall or small room. The total reverberation delay time depends on the size the comb filter/early reflections buffers and the sampling rate.

Low pass filtering in each comb filter stage reduces the metallic sound and shortens the reverberation time of the high frequencies, just as a real auditorium response does. The allpass filter is used along with the comb filters to add some color to the 'colorless/flat' sound by varying the phase, thus helping to emulate the sound characteristics of a real auditorium.

ADSP-21061 Low Pass Comb Filter Subroutine Example



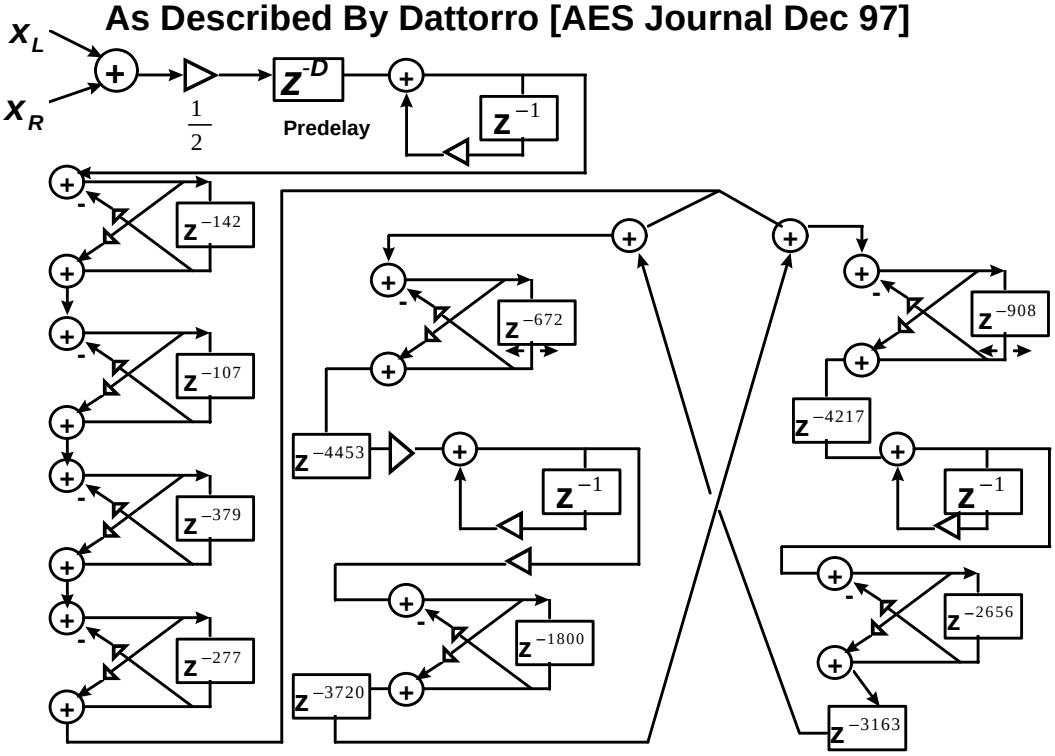

```
/*     Low Pass IIR Comb Filter Structure:

*/

Low_Pass_Comb_Filter:
       L3 = @comb_output;
       B3 = comb_output;
       R0 = 0x50710000;                /* gf = Gf/(1+gl) -- comb feedback gain */
       R1 = 0x26660000;                /* gl -- low pass filter gain */

       L9 = COMB_LENGTH;
       R4 = PM(I9,0);                  /* read comb buffer -> output */

       R5 = DM(comb_lpf_state);        /* read previous low pass filter state */
       MRF = R4*R0 (SSF), DM(I3,M1) = R4;  /* save output, feedback comb output */
       MRF = MRF + R5*R1 (SSFR);       /* add low pass filter state     */
       R10 = MR1F;
       DM(comb_lpf_state) = R10;       /* replace with new filter state */

       MR1F = R5;                      /* get old low pass filter output */
       R7 = 0x46760000;                /* gi = 1/(1 + gf + gf*gl) */
       MRF = MRF + R6*R7 (SSFR);       /* mac lpf output with input*/
       R10 = MR1F;
       PM(I9,-1) = R10;                /* write sample to buffer */
       RTS;
```

ADSP-21065L Example All-Pass Filter Implementation

```
/*     1st Order Allpass Transfer function and I/O difference equations:
              -a + z(-D)
        H(z) = -----------          y(n) = ay(n - D) - ax(n) + x(n - D)
              1 - az^(-D)



       Allpass Filter Structure:
                              a
                         |-------------|>------------|
                         |                           |
                         |      _________            |
                         |     |         |         + v
        x(n) ---------->O---------| z^(-D)  |-----------O--------> y(n)
                        ^ +       |_________|       |
                        |                           |
                        |           -a              |
                        |---------<|----------------|



       Using the Allpass Filter as a reverb building block
       ---------------------------------------------------
       IIR comb filters tend to magnify input signal frequencies near comb filter peak
       frequencies.  Allpass filters can be used to prevent this 'coloration' of the input
       since it has a relatively flat magnitude response for all frequencies.
*/

all_pass_filter:
       L0 = @all_pass;
       R1 = 0x599A0000;                /* feedback gain    */

       R10 = DM(I0,0);                 /* load output of buffer */
       MR1F = R10;
       MRF = MRF + R1*R0 (SSFR);       /* add to (feedback gain)*(input) */
       MRF = SAT MRF;
       R3 = MR1F;                      /* output of all pass in R3 */

       MR1F = R0;                      /* put input of all pass in MR1F */
       MRF = MRF - R1*R3 (SSFR);       /* input - (feedback gain)*(output) */
       MRF = SAT MRF;
       R10 = MR1F;
       DM(I0,M7) = R10;                /* save to input of buffer */
       RTS;
```

Figure 83 is an example implementation of a Plate Reverb topology that is described by Dattorro[30], and will not be discussed in too much detail here. The reader should refer to Dattorro's description on this class of reverb algorithms, although it is not explored in much theoretical detail, as he suggests that it is best explored through tinkering with the gains delay line values and output tap points. This suggested implementation yields very high quality reverberation very efficiently. Notice that is has similar building blocks using comb filters. In addition, it uses allpass filter diffusers based on lattice structure topologies.

Figure 83.
Griesinger's Plate Class Reverberation Structure
As Described By Dattorro [AES Journal Dec 97]




```
/* ********************************************************************************************
       Digital Plate-Class Reverberation Audio Effect - Griesinger's Model
       Described by Jon Dattorro in "Effect Design Part 1: Reverberator and
       Other Filters, " Journal of the Audio Engineering Society," Vol. 45,
       No. 9, pp. 660-684, September 1997.

       Created for the 21065L EZ-LAB Evaluation Platform

       Includes on-the-fly selection of reverb comb & allpass filter gains/lengths,
       predelay gain/length presets, and left/right panning via IRQ1 and IRQ2 pushbutton control

       ** Works well for single instruments such as a guitar, keyboard, violin....

   ******************************************************************************************** */

/* ADSP-21060 System Register bit definitions */
#include       "def21065l.h"
#include       "new65Ldefs.h"

.GLOBAL        Digital_Reverberator;
.GLOBAL        Init_Reverb_Buffers;
.GLOBAL        change_reverb_settings;
.GLOBAL        modify_reverb_mix;
.EXTERN        Left_Channel;
.EXTERN        Right_Channel;

/* Default Reverberation Parameters - Reverb DM Variables and pointers */
.segment /dm    rev_vars;

#define Fs1    29761          /* Sample Rate is 29761 Hz, default recommended by Dattorro */
                              /* reducing Fs will reduce delay line memory requirements */
```

```
                              /* Reverberation can be convincing at sample rates as low as 20-24 kHz */
#define Fs2    48000          /* Sample Rate is 48000 Hz, delay lines increased for 48 kHz */

/* reverb left channel output taps at 48 kHz fs */
#define D1_L                  429            /* = 266 @ 29761 Hz Fs */
#define D2_L                  4797           /* = 2974 @ 29761 Hz Fs*/
#define D3_L                  3085           /* = 1913 @ 29761 Hz Fs */
#define D4_L                  3219           /* = 1996 @ 29761 Hz Fs*/
#define D5_L                  3210           /* = 1990 @ 29761 Hz Fs */
#define D6_L                  302            /* = 187 @ 29761 Hz Fs */
#define D7_L                  1719           /* = 1066 @ 29761 Hz Fs */

/* reverb right channel output taps at 48 kHz fs*/
#define D1_R                  569            /* = 353 @ 29761 Hz Fs */
#define D2_R                  3627           /* = 3627 @ 29761 Hz Fs */
#define D3_R                  5850           /* = 1228 @ 29761 Hz Fs */
#define D4_R                  2673           /* = 2673 @ 29761 Hz Fs */
#define D5_R                  4311           /* = 2111 @ 29761 Hz Fs */
#define D6_R                  540            /* = 335 @ 29761 Hz Fs */
#define D7_R                  195            /* = 121 @ 29761 Hz Fs */

/* pointers for input and decay diffusers */
.VAR    all_pass_ptr1;
.VAR    all_pass_ptr2;
.VAR    all_pass_ptr3;
.VAR    all_pass_ptr4;
.VAR    all_pass_ptr5;
.VAR    all_pass_ptr6;
.VAR    all_pass_ptr7;
.VAR    all_pass_ptr8;

/* Single Length Comb Filter State Variables */
.VAR    comb1_feedback_state;
.VAR    comb2_feedback_state;
.VAR    comb3_feedback_state;

.VAR    recirculate1_feedback;
.VAR    recirculate2_feedback;
.VAR    diffusion_result;

.VAR    predelay_output;
.VAR    reverb_left;
.VAR    reverb_right;
.VAR    Lrev_output_taps[6];
.VAR    Rrev_output_taps[6];

.endseg;


/* -------------DATA MEMORY DELAY LINE & ALLPASS FILTER BUFFERS  ------------------------*/
.segment /dm   dm_revrb;

/* Allpass and Delay Line Filter Length Definitions */
#define allpass_Dline1         2168                  /* = 1344 @ 29761 Hz Fs */
#define allpass_Dline2         2930                  /* = 1816 @ 29761 Hz Fs */
#define ALLPASS1_LENGTH        229                   /* = 142 @ 29761 Hz Fs */
#define ALLPASS2_LENGTH        172                   /* = 107 @ 29761 Hz Fs */
#define ALLPASS3_LENGTH        611                   /* = 379 @ 29761 Hz Fs */
#define ALLPASS4_LENGTH        447                   /* = 277 @ 29761 Hz Fs */
#define ALLPASS5_LENGTH        allpass_Dline1/2      /* 2168/2 variable delay rotating around tap center
*/
#define ALLPASS6_LENGTH        allpass_Dline2/2      /* 2930/2 variable delay rotating around tap center
*/
#define ALLPASS7_LENGTH        2903                  /* = 1800 @ 29761 Hz Fs */
#define ALLPASS8_LENGTH        4284                  /* = 2656 @ 29761 Hz Fs */
#define D_Line1                7182                  /* = 4453 @ 29761 Hz Fs */
#define D_Line2                6000                  /* = 3720 @ 29761 Hz Fs */
#define D_Line3                6801                  /* = 4217 @ 29761 Hz Fs */
#define D_Line4                5101                  /* = 3163 @ 29761 Hz Fs */

/* Audio delay lines */
.VAR    predelay[6321];
.VAR    w1[D_Line1];
.VAR    w2[D_Line2];
.VAR    w3[D_Line3];
```

```
.VAR    w4[D_Line4];

/* input diffusers using allpass filter structures */
.VAR    diffuser_1[ALLPASS1_LENGTH];
.VAR    diffuser_2[ALLPASS2_LENGTH];
.VAR    diffuser_3[ALLPASS3_LENGTH];
.VAR    diffuser_4[ALLPASS4_LENGTH];
.VAR    decay_diffuser_A1[ALLPASS5_LENGTH];   /* allpass diffuser with variable delay */
.VAR    decay_diffuser_B1[ALLPASS6_LENGTH];   /* allpass diffuser with variable delay */
.VAR    decay_diffuser_A2[ALLPASS7_LENGTH];
.VAR    decay_diffuser_B2[ALLPASS8_LENGTH];

.endseg;


/* ----------- INTERRUPT/FLAG REVERB FX DEMO CONTROL PARAMETERS ---------*/
.segment /dm   IRQ_ctl;

/* Reverb Control Parameters, these control 'knobs' are used to change the response on-the-fly */
.VAR    decay = 0x40000000;           /* Rate of decay - 0.05 */
.VAR    bandwidth = 0x7f5c28f6;       /*  = 0.9995, High-frequency attenuation on input */
                                      /* full bandwidth = 0x9999999 */
.VAR    damping = 0x0010624d;         /*  = 0.0005, High-frequency damping; no damping = 0.0 */
.VAR    predelay_time = 200;          /* controls length (L6 register)of predelay buffer */
                                      /* length value always << max buffer length! */

.VAR    decay_diffusion_1 = 0x5999999a;       /*  = 0.70, Controls density of tail */
.VAR    decay_diffusion_2 = 0x40000000;       /*  = 0.50, Decorrelates tank signals */
                                      /* decay diffusion 2 = decay +0.15, floor = 0.25, ceiling - 0.50 */
.VAR    input_diffusion_1 = 0x60000000;       /*  = 0.75, Decorrelates incoming signal */
.VAR    input_diffusion_2 = 0x50000000;       /*  = 0.625, Decorrelates incoming signal */

.VAR    DRY_GAIN_LEFT = 0x7FFFFFFF;   /* Gain Control for left channel */
                                      /* scale between 0x00000000 and 0x7FFFFFFF */
.VAR    DRY_GAIN_RIGHT = 0x7FFFFFFF;  /* Gain Control for right channel */
                                      /* scale between 0x00000000 and 0x7FFFFFFF */
.VAR    PREDEL_GAIN_LEFT = 0x00000000;/* Gain Control for predelay output */
                                      /* scale between 0x00000000 and 0x7FFFFFFF */
.VAR    PREDEL_GAIN_RIGHT = 0x00000000; /* Gain Control for predelay output */
                                      /* scale between 0x00000000 and 0x7FFFFFFF */
.VAR    WET_GAIN_LEFT = 0x7FFFFFFF;   /* Gain for reverb result */
                                      /* scale between 0x00000000 and 0x7FFFFFFF */
.VAR    WET_GAIN_RIGHT = 0x7FFFFFFF;  /* Gain for reverb result */
                                      /* scale between 0x00000000 and 0x7FFFFFFF */

.VAR    IRQ1_counter = 0x00000004;    /* selects preset 1 on first IRQ1 assertion */
.VAR    IRQ2_counter = 0x00000004;    /* selects preset 1 on first IRQ2 assertion */

.endseg;

/* -------------PROGRAM MEMORY CODE--------------------------------*/
.segment /pm pm_code;

Init_Reverb_Buffers:
       /* initialize all-pass filter pointers to top of respective buffers */
       B7 = diffuser_1;
       DM(all_pass_ptr1) = B7;
       B7 = diffuser_2;
       DM(all_pass_ptr2) = B7;
       B7 = diffuser_3;
       DM(all_pass_ptr3) = B7;
       B7 = diffuser_4;
       DM(all_pass_ptr4) = B7;
       B7 = decay_diffuser_A1;
       DM(all_pass_ptr5) = B7;
       B7 = decay_diffuser_B1;
       DM(all_pass_ptr6) = B7;
       B7 = decay_diffuser_A2;
       DM(all_pass_ptr7) = B7;
       B7 = decay_diffuser_B2;
       DM(all_pass_ptr8) = B7;

       /* Initialize Audio Delay Lines */
       B2 = w1;  L2 = @w1;           /* delay-line buffer 1 pointer and length */
       B3 = w2;  L3 = @w2;           /* delay-line buffer 1 pointer and length */
```

```
       B4 = w3;  L4 = @w3;           /* delay-line buffer 1 pointer and length */
       B5 = w4;  L5 = @w4;           /* delay-line buffer 1 pointer and length */
       B6 = predelay; L6 = @predelay;

       /* clear all audio delay line buffers to zero */
       m2 = 1; m3 = 1; m4 = 1; m5 = 1; m6 = 1;

       LCNTR = L2;
        DO clrDline_1 UNTIL LCE;
clrDline_1:    dm(i2, m2) = 0;

       LCNTR = L3;
        DO clrDline_2 UNTIL LCE;
clrDline_2:    dm(i3, m3) = 0;

       LCNTR = L4;
        DO clrDline_3 UNTIL LCE;
clrDline_3:    dm(i4, m4) = 0;

       LCNTR = L5;
        DO clrDline_4 UNTIL LCE;
clrDline_4:    dm(i5, m5) = 0;

       LCNTR = L6;
        DO clrDline_5 UNTIL LCE;
clrDline_5:    dm(i6, m6) = 0;

       RTS;

/* ------------------------------------------------------------------------------------------ */
/*                                                                                            */
/*                            Digital Reverb Filter Routines                                  */
/*                                                                                            */
/* ------------------------------------------------------------------------------------------ */

Digital_Reverberator:
       /* combine both left and right input samples together into 1 signal */
       r0 = dm(Left_Channel);              /* left input sample */
       r1 = dm(Right_Channel);             /* right input sample */
       r0 = ashift r0 by -1;               /* scale signal by 1/2 for equal mix */
       r1 = ashift r1 by -1;               /* scale signal by 1/2 for equal mix */
       r2 = r0 + r1;                       /* 1/2xLeft(n) + 1/2 xRight(n) = sum of input samples */

compute_predelay:
       L6 = dm(predelay_time);             /* get predelay time setting, default is L6=@predelay */
       r3 = dm(i6, 0);                     /* get oldest sample, time delay = D_Line1 x fs*/
       dm(i6, -1) = r2;                    /* write input sample to buffer */
       dm(predelay_output) = r3;           /* used for final mix */
       r2 = r3;                            /* r2 is input to reverb routines */

       call (PC, hi_freq_input_atten);       /* attenuate high-freqencies on input */

       r0 = r2;
       call (PC, input_diffusers);         /* call all-pass filters */

       call (PC, reverberation_tank);      /* controls the rate of reverberation decay */

       call reverb_mixer;

/*
bypass_mixer:
       r0 = dm(reverb_left);
       dm(Left_Channel) = r0;
       r0 = dm(reverb_right);
       dm(Right_Channel) = r0;
*/
       rts;

/* ------------------------------------------------------------------------------------------ */
/*                                                                                            */
/*                            High Frequency Input Attenuator                                 */
/*                                                                                            */
/*     This routine is a simple comb filter, used to attenuate high frequencies on the input.  */
/*     Higher frequencies tend to diminish faster than lower frequecies in reverberant        */
/*     acoustic spaces                                                                        */
```

```
/*      DM(bandwidth) = full bandwidth gain when set to 0.9999999 (0x7FFFFFFF in 1.31 format)     */
/* ------------------------------------------------------------------------------------------- */

hi_freq_input_atten:
       /* comb filter 1 - attenuate high frequencies on the input */
       /* full bandwidth gain is 0.9999999 */
       r4 = r2;                              /* r4 = comb filter input */
       r6 = dm(bandwidth);                   /* high frequency bandwidth gain */
       r7 = 0x7FFFFFFF;                      /* 0.99999 or approximately = 1 */
       r8 = r7 - r6;                         /* r8 = 1 - bandwidth */
       r5 = DM(comb1_feedback_state);        /* read previous low pass filter output state */
       mrf = r4*r6 (SSF);                    /* scale comb filter input */
       mrf = mrf + r5*r8 (SSFR);             /* add previous filter state */
       r2 = mr1f;                            /* r10 = comb filter output */
       dm(comb1_feedback_state) = r10;       /* replace with new filter state */
       rts;                                  /* return from subroutine */

/* ------------------------------------------------------------------------------------------   */

/* --------------- All Pass Filter Routines ------------------------------ */
/*                                                                         */
/*      Each all-pass filter diffusers are implemented in the topology     */
/*      of a two-multiplier lattice structure.                             */
/*                                                                         */
/*             input   ->      R0                                          */
/*             output  ->     R3                                           */
/*                                                    `                    */
/*      Also, it is not necessary to save and restore all comb filter      */
/*      FIR or all-pass filter index and length registers, if only         */
/*      doing this reverb demo.  These extra instructions are included     */
/*      so that this example can easily be combined with other audio       */
/*      effects that require the use of multiple buffers                   */
/*                                                                         */
/*      1 index register I7 is used for all 4 allpass filters              */
/* ----------------------------------------------------------------------- */

input_diffusers:

       B7 = diffuser_1;                      /* set base address to buffer */
       I7 = DM(all_pass_ptr1);               /* get previous allpass 1 pointer address */
       L7 = @diffuser_1;                     /* set length of circular buffer */
       R1 = DM(input_diffusion_1);           /* feedback gain for allpass */

allpass_filt1:
       R10 = DM(I7,0);                       /* load output of buffer */
       MR1F = R10;                           /* put in forground MAC register */
       MRF = MRF + R1*R0 (SSFR);             /* add to (feedback gain)*(input) */
       MRF = SAT MRF;                        /* saturate if necessary */
       R3 = MR1F;                            /* output of all pass in R3 */

       MR1F = R0;                            /* put input of all pass in MR1F */
       MRF = MRF - R1*R3 (SSFR);             /* input - (feedback gain)*(output) */
       MRF = SAT MRF;                        /* saturate if necessary */
       R10 = MR1F;                           /* put MAC result in register file */
       DM(I7,1) = R10;                       /* save to input of buffer, update pointer */
       DM(all_pass_ptr1) = I7;               /* save current allpass 1 pointer address for next time */


allpass_filt2:
       B7 = diffuser_2;                      /* set base address to buffer */
       I7 = DM(all_pass_ptr2);                    /* get previous allpass 2 pointer address */
       L7 = @diffuser_2;                     /* set length of circular buffer */
       R1 = DM(input_diffusion_1);           /* feedback gain for allpass 1*/
       R0 = R3;                              /* output of allpass 1 = input of allpass 2 */

       R10 = DM(I7,0);                       /* load output of buffer */
       MR1F = R10;                           /* put in forground MAC register */
       MRF = MRF + R1*R0 (SSFR);             /* add to (feedback gain)*(input) */
       MRF = SAT MRF;                        /* saturate if necessary */
       R3 = MR1F;                            /* output of all pass in R3 */

       MR1F = R0;                            /* put input of all pass in MR1F */
       MRF = MRF - R1*R3 (SSFR);             /* input - (feedback gain)*(output) */
       MRF = SAT MRF;                        /* saturate if necessary */
       R10 = MR1F;                           /* put MAC result in register file */
```

```
        DM(I7,1) = R10;                         /* save to input of buffer, update pointer */
        DM(all_pass_ptr2) = I7;                 /* save current allpass 2 pointer address for next time */

allpass_filt3:
        B7 = diffuser_3;                        /* set base address to buffer */
        I7 = DM(all_pass_ptr3);                 /* get previous allpass 3 pointer address */
        L7 = @diffuser_3;                       /* set length of circular buffer */
        R1 = DM(input_diffusion_2);             /* feedback gain for allpass 2*/
        R0 = R3;                                /* output of allpass 2 = input of allpass 3 */

        R10 = DM(I7,0);                         /* load output of buffer */
        MR1F = R10;                             /* put in forground MAC register */
        MRF = MRF + R1*R0 (SSFR);               /* add to (feedback gain)*(input) */
        MRF = SAT MRF;                          /* saturate if necessary */
        R3 = MR1F;                              /* output of all pass in R3 */

        MR1F = R0;                              /* put input of all pass in MR1F */
        MRF = MRF - R1*R3 (SSFR);               /* input - (feedback gain)*(output) */
        MRF = SAT MRF;                          /* saturate if necessary */
        R10 = MR1F;                             /* put MAC result in register file */
        DM(I7,1) = R10;                         /* save to input of buffer, update pointer */
        DM(all_pass_ptr3) = I7;                 /* save current allpass 3 pointer address for next time */

allpass_filt4:
        B7 = diffuser_4;                        /* set base address to buffer */
        I7 = DM(all_pass_ptr4);                 /* get previous allpass 4 pointer address */
        L7 = @diffuser_4;                       /* set length of circular buffer */
        R1 = DM(input_diffusion_2);             /* feedback gain for allpass 3 */
        R0 = R3;                                /* output of allpass 3 = input of allpass 4 */

        R10 = DM(I7,0);                         /* load output of buffer */
        MR1F = R10;                             /* put in forground MAC register */
        MRF = MRF + R1*R0 (SSFR);               /* add to (feedback gain)*(input) */
        MRF = SAT MRF;                          /* saturate if necessary */
        R3 = MR1F;                              /* output of all pass in R3 */
        DM(diffusion_result) = R3;              /* save for holding tank routines */

        MR1F = R0;                              /* put input of all pass in MR1F */
        MRF = MRF - R1*R3 (SSFR);               /* input - (feedback gain)*(output) */
        MRF = SAT MRF;                          /* saturate if necessary */
        R10 = MR1F;                             /* put MAC result in register file */
        DM(I7,1) = R10;                         /* save to input of buffer, update pointer */
        DM(all_pass_ptr4) = I7;                 /* save current allpass 4 pointer address for next time */

        RTS;                                    /* return from subroutine */


/* ----------------- Reverb Holding Tank Routines ------------------------------  */
/*                                                                                */
/*                     R3      <-      input                                      */
/*                                                                                */
/*                     output  ->      DM(reverb_left)                            */
/*                                     DM(reverb_right)                           */
/*                                                                                */
/*      The reverberation tank recirculates 4 diffusers.  It's purpose            */
/*      is to 'trap' the input and make it recirculate in a 'figure 8'            */
/*      structure, thus altering the tail of the decaying reverb response.        */
/*      The decay coefficients determine the rate of decay.                       */
/*      It is recommended to set the coefficents by ear.                          */
/*                                                                                */
/*      Also note:  Diffuser Lattices A1 and B1 have negative coefficients.       */
/*      The allpass structures have the MAC adds and subtract instructions        */
/*      reversed.  The impulse response changes, but it is still an allpass       */
/*      filter. This is recommended by Dattorro to further enhance the delay      */
/*      diffusion between both sides of the holding tank                          */
/*                                                                                */
/* -------------------------------------------------------------------------------  */

reverberation_tank:
        /* initialize left & right output tap buffer pointers */
        B0 = Lrev_output_taps;
        L0 = @Lrev_output_taps;
        B1 = Rrev_output_taps;
        L1 = @Rrev_output_taps;
```

```
        r1 = DM(recirculate1_feedback);       /* get previously trapped incoming audio signal */
        r2 = DM(decay);                       /* Rate of decay on previous trapped signal */
        r4 = r1*r2 (SSF);                     /* scale prior signal state */
        r3 = DM(diffusion_result);            /* get output from input diffusion section */
        r14 = r3 + r4;                        /* add to current input diffuser result */

        r1 = DM(recirculate2_feedback);       /* get previously trapped incoming audio signal */
        r2 = DM(decay);                       /* Rate of decay on previous trapped signal */
        r4 = r1*r2 (SSF);                     /* scale prior signal state */
        r3 = DM(diffusion_result);            /* get output from input diffusion section */
        r15 = r3 + r4;                        /* add to current input diffuser result */
        /* r14 and r15 are inputs to holding tank */

diffusor_A1:                                  /* allpass filter with variable delay */
        B7 = decay_diffuser_A1;               /* set base address to buffer */
        I7 = DM(all_pass_ptr5);               /* get previous allpass 2 pointer address */
        L7 = @decay_diffuser_A1;              /* set length of circular buffer */
        r1 = DM(decay_diffusion_1);           /* feedback gain for allpass 1*/
        r0 = r15;                             /* tank input 1 */

        r10 = DM(I7,0);                       /* load output of buffer */
        mr1f = r10;                           /* put in forground MAC register */
        mrf = mrf - r1*r0 (SSFR);             /* add to -(feedback gain)*(input) */
        mrf = SAT mrf;                        /* saturate if necessary */
        r3 = mr1f;                            /* output of all pass in R3 */

        mr1f = r0;                            /* put input of all pass in MR1F */
        mrf = mrf + r1*r3 (SSFR);             /* input + (feedback gain)*(output) */
        mrf = SAT mrf;                        /* saturate if necessary */
        r10 = mr1f;                           /* put MAC result in register file */
        DM(I7,-1) = r10;                      /* save to input of buffer, update pointer */
        DM(all_pass_ptr5) = I7;               /* save current allpass 2 pointer address for next time */

audio_delay1:
        r5 = dm(i2, 0);                       /* get oldest sample, time delay = D_Line1 x fs*/

        /* tap inside of circular delay line 1, r0 = sampleD5_L = D5_L-th tap */
        m4 =  D5_L;  modify(i2, m2);          /* point to d-th tap */
        m4 = -D5_L;  r0 = dm(i2, m2);         /* put d-th tap in data register */
        DM(Lrev_output_taps + 4)= r0;         /* write to 4th location in left reverb output buffer */

        /* tap inside of circular delay line 1, r0 = sampleD1_R = D1_R-th tap */
        m4 =  D1_R;  modify(i2, m2);          /* point to d-th tap */
        m4 = -D1_R;  r0 = dm(i2, m2);         /* put d-th tap in data register */
        DM(Rrev_output_taps + 0)= r0;         /* write to 0'th location in left reverb output buffer */

        /* tap inside of circular delay line 1, r0 = sampleD2_R = D2_R-th tap */
        m4 =  D2_R;  modify(i2, m2);          /* point to d-th tap */
        m4 = -D2_R;  r0 = dm(i2, m2);         /* put d-th tap in data register */
        DM(Rrev_output_taps + 1)= r0;         /* write to 1st location in left reverb output buffer */

        dm(i2, -1) = r3;                      /* write input sample to buffer */


comb_filter_2:
        /* comb filter 2 */
        r4 = r5;                              /* r4 = comb filter input */
        r6 = dm(damping);                     /* high frequency dampling gain */
        r7 = 0x7FFFFFFF;                      /* 0.99999 or approximately = 1 */
        r8 = r7 - r6;                         /* r8 = 1 - damping */
        r5 = DM(comb2_feedback_state);        /* read previous low pass filter output state */
        mrf = r4*r8 (SSF);                    /* scale comb filter input */
        mrf = mrf + r5*r6 (SSFR);             /* add previous filter state  */
        r10 = mr1f;                           /* r10 = comb filter output */
        dm(comb2_feedback_state) = r10;       /* replace with new filter state */

        r11 = DM(decay);
        r0 = r10*r11 (SSFR);

diffusor_A2:                                  /* allpass filter with variable delay */
        B7 = decay_diffuser_A2;               /* set base address to buffer */
        I7 = DM(all_pass_ptr6);               /* get previous allpass 2 pointer address */
        L7 = @decay_diffuser_A2;              /* set length of circular buffer */
        R1 = DM(decay_diffusion_2);           /* feedback gain for allpass 1*/
        R0 = R3;                              /* output of allpass 1 = input of allpass 2 */
```

```
        r10 = DM(I7,0);                     /* load output of buffer */
        mr1f = r10;                         /* put in forground MAC register */
        mrf = mrf + r1*r0 (SSFR);           /* add to (feedback gain)*(input) */
        mrf = SAT mrf;                      /* saturate if necessary */
        r3 = mr1f;                          /* output of all pass in R3 */

        mr1f = r0;                          /* put input of all pass in MR1F */
        mrf = mrf - r1*r3 (SSFR);           /* input - (feedback gain)*(output) */
        mrf = SAT mrf;                      /* saturate if necessary */
        r10 = mr1f;                         /* put MAC result in register file */

        /* tap inside of decay diffuser A2 filter delay line, r0 = sampleD6_L = D6_L-th tap */
        m7 =  D6_L;  modify(i7, m7);        /* point to d-th tap */
        m7 = -D6_L;  r0 = dm(i7, m7);       /* put d-th tap in data register */
        DM(Lrev_output_taps + 5)= r0;       /* write to 5th location in left reverb output buffer */

        /* tap inside of decay diffuser A2 filter delay line, r0 = sampleD3_R = D3_R-th tap */
        m7 =  D3_R;  modify(i7, m7);        /* point to d-th tap */
        m7 = -D3_R;  r0 = dm(i7, m7);       /* put d-th tap in data register */
        DM(Rrev_output_taps + 2)= r0;       /* write to 2nd location in left reverb output buffer */

        DM(I7,-1) = r10;                    /* save to input of buffer, update pointer */
        DM(all_pass_ptr6) = I7;             /* save current allpass 2 pointer address for next time */

audio_delay2:
        r5 = dm(i3, 0);                     /* get oldest sample, time delay = D_Line2 x fs*/

        /* tap inside of circular delay line 2, r0 = sampleD7_L = D7_L-th tap */
        m4 =  D7_L;  modify(i3, m3);        /* point to d-th tap */
        m4 = -D7_L;  r0 = dm(i3, m3);       /* put d-th tap in data register */
        DM(Lrev_output_taps + 6)= r0;       /* write to 6th location in left reverb output buffer */

        /* tap inside of circular delay line 2, r0 = sampleD4_R = D4_R-th tap */
        m4 =  D4_R;  modify(i3, m3);        /* point to d-th tap */
        m4 = -D4_R;  r0 = dm(i3, m3);       /* put d-th tap in data register */
        DM(Rrev_output_taps + 3)= r0;       /* write to 3rd location in left reverb output buffer */

        dm(i3, -1) = r3;                    /* write input sample to buffer */

        /* feed output back to top of tank */
        r5 = DM(recirculate2_feedback);     /* feed to other side of tank 8 */

diffusor_B1:                                /* allpass filter with variable delay */
        B7 = decay_diffuser_B1;             /* set base address to buffer */
        I7 = DM(all_pass_ptr7);             /* get previous allpass 2 pointer address */
        L7 = @decay_diffuser_B1;            /* set length of circular buffer */
        r1 = DM(decay_diffusion_1);         /* feedback gain for allpass 1*/
        r0 = r15;                           /* tank input 1 */

        r10 = DM(I7,0);                     /* load output of buffer */
        mr1f = r10;                         /* put in forground MAC register */
        mrf = mrf - r1*r0 (SSFR);           /* add to -(feedback gain)*(input) */
        mrf = SAT mrf;                      /* saturate if necessary */
        r3 = mr1f;                          /* output of all pass in R3 */

        mr1f = r0;                          /* put input of all pass in MR1F */
        mrf = mrf + r1*r3 (SSFR);           /* input + (feedback gain)*(output) */
        mrf = SAT mrf;                      /* saturate if necessary */
        r10 = mr1f;                         /* put MAC result in register file */
        DM(I7,-1) = r10;                    /* save to input of buffer, update pointer */
        DM(all_pass_ptr7) = I7;             /* save current allpass 2 pointer address for next time */

audio_delay3:
        r5 = dm(i4, 0);                     /* get oldest sample, time delay = D_Line3 x fs */

        /* tap inside of circular delay line 3, r0 = sampleD1_L = D1_L-th tap */
        m4 =  D1_L;  modify(i4, m4);        /* point to d-th tap */
        m4 = -D1_L;  r0 = dm(i4, m4);       /* put d-th tap in data register */
        DM(Lrev_output_taps)= r0;           /* write to 0'th location in left reverb output buffer */

        /* tap inside of circular delay line 3, r0 = sampleD2_L = D2_L-th tap */
        m4 =  D2_L;  modify(i4, m4);        /* point to d-th tap */
        m4 = -D2_L;  r0 = dm(i4, m4);       /* put d-th tap in data register */
        DM(Lrev_output_taps + 1)= r0;       /* write to 1st location in left reverb output buffer */
```

```
       /* tap inside of circular delay line 3,  r0 = sampleD5_R = D5_R-th tap */
       m4 =  D5_R;  modify(i4, m4);          /* point to d-th tap */
       m4 = -D5_R;  r0 = dm(i4, m4);         /* put d-th tap in data register */
       DM(Rrev_output_taps + 4)= r0;         /* write to 4th location in left reverb output buffer */

       dm(i4, -1) = r3;                      /* store current input into delay line */

comb_filter_3:
       /* comb filter 3 */
       r4 = r5;                              /* r4 = comb filter input */
       r6 = dm(damping);                     /* high frequency dampling gain */
       r7 = 0x7FFFFFFF;                      /* 0.99999 or approximately = 1 */
       r8 = r7 - r6;                         /* r8 = 1 - damping */
       r5 = DM(comb3_feedback_state);        /* read previous low pass filter output state */
       mrf = r4*r8 (SSF);                    /* scale comb filter input */
       mrf = mrf + r5*r6 (SSFR);             /* add previous filter state  */
       r10 = mr1f;                           /* r10 = comb filter output */
       dm(comb3_feedback_state) = r10;       /* replace with new filter state */

       r11 = DM(decay);
       r0 = r10*r11 (SSFR);

diffusor_B2:                                 /* allpass filter with variable delay */
       B7 = decay_diffuser_B2;               /* set base address to buffer */
       I7 = DM(all_pass_ptr8);               /* get previous allpass 2 pointer address */
       L7 = @decay_diffuser_B2;              /* set length of circular buffer */
       R1 = DM(decay_diffusion_2);           /* feedback gain for allpass 1*/
       R0 = R3;                              /* output of allpass 1 = input of allpass 2 */

       r10 = DM(I7,0);                       /* load output of buffer */
       mr1f = r10;                           /* put in forground MAC register */
       mrf = mrf + r1*r0 (SSFR);             /* add to (feedback gain)*(input) */
       mrf = SAT mrf;                        /* saturate if necessary */
       r3 = mr1f;                            /* output of all pass in R3 */

       mr1f = r0;                            /* put input of all pass in MR1F */
       mrf = mrf - r1*r3 (SSFR);             /* input - (feedback gain)*(output) */
       mrf = SAT mrf;                        /* saturate if necessary */
       r10 = mr1f;                           /* put MAC result in register file */

       /* tap inside of decay diffuser filter delay line, r0 = sampleD3_L = D3_L-th tap */
       m7 =  D3_L;  modify(i7, m7);          /* point to d-th tap */
       m7 = -D3_L;  r0 = dm(i7, m7);         /* put d-th tap in data register */
       DM(Lrev_output_taps + 2)= r0;         /* write to 2nd location in left reverb output buffer */

       /* tap inside of decay diffuser A2 filter delay line, r0 = sampleD6_R = D6_R-th tap */
       m7 =  D6_R;  modify(i7, m7);          /* point to d-th tap */
       m7 = -D6_R;  r0 = dm(i7, m7);         /* put d-th tap in data register */
       DM(Rrev_output_taps + 5)= r0;         /* write to 5th location in left reverb output buffer */

       DM(I7,-1) = r10;                      /* save to input of buffer, update pointer */
       DM(all_pass_ptr8) = I7;               /* save current allpass 2 pointer address for next time */

audio_delay4:
       r5 = dm(i5, 0);                       /* get oldest sample, time delay = D_Line4 x fs */

       /* tap inside of circular delay line 4, r0 = sampleD4_L = D4_L-th tap */
       m4 =  D4_L;  modify(i5, m5);          /* point to d-th tap */
       m4 = -D4_L;  r0 = dm(i5, m5);         /* put d-th tap in data register */
       DM(Lrev_output_taps + 3)= r0;         /* write to 3rd location in left reverb output buffer */

       /* tap inside of circular delay line 4, r0 = sampleD7_R = D7_R-th tap */
       m4 =  D7_R;  modify(i5, m5);          /* point to d-th tap */
       m4 = -D7_R;  r0 = dm(i5, m5);         /* put d-th tap in data register */
       DM(Rrev_output_taps + 6)= r0;         /* write to 6th location in left reverb output buffer */

       dm(i5, -1) = r3;                      /* save delay line input to buffer */

       /* feed output back to top of tank */
       r5 = DM(recirculate1_feedback);       /* feed to other side of tank 8 */

reverb_output_taps:
/* combine all of the left & right reverb output taps taken at different point in the holding tank */
```

```
       /* left output, all wet */
       B0 = Lrev_output_taps;               /* sum of left reverb output taps */
       R1 = 0x4CCCCCCD;                     /* scale tap outputs by 0.60  */
       R0 = DM(I0,1);                       /* load first output tap      */
       MRF = R0*R1 (SSF), R0 = DM(I0,1);         /* compute product, load next output*/
       MRF = MRF + R0*R1 (SSF), R0 = DM(I0,1);   /* compute sum of products    */
       MRF = MRF - R0*R1 (SSF), R0 = DM(I0,1);
       MRF = MRF + R0*R1 (SSF), R0 = DM(I0,1);   /* and so on ...              */
       MRF = MRF - R0*R1 (SSF), R0 = DM(I0,1);
       MRF = MRF - R0*R1 (SSF), R0 = DM(I0,1);
       MRF = MRF - R0*R1 (SSFR);
       R2 = MR1F;
       DM(reverb_left) = R2;                     /* save left reverb result */

       /* right output, all wet */
       B1 = Rrev_output_taps;                    /* sum of right reverb output taps */
       R1 = 0x4CCCCCCD;                          /* scale tap outputs by 0.60  */
       R0 = DM(I1,1);                            /* load first output tap      */
       MRF = R0*R1 (SSF), R0 = DM(I1,1);         /* compute product, load next output*/
       MRF = MRF + R0*R1 (SSF), R0 = DM(I1,1);   /* compute sum of products    */
       MRF = MRF - R0*R1 (SSF), R0 = DM(I1,1);   /* subtract product           */
       MRF = MRF + R0*R1 (SSF), R0 = DM(I1,1);   /* and so on ...              */
       MRF = MRF - R0*R1 (SSF), R0 = DM(I1,1);
       MRF = MRF - R0*R1 (SSF), R0 = DM(I1,1);
       MRF = MRF - R0*R1 (SSFR);
       R2 = MR1F;
       DM(reverb_right) = R2;                    /* save right reverb result */
       RTS;
/* ------------------------------------------------------------------------------------- */

reverb_mixer:
       r2 = 0x2AAA0000;                  /* set up scaling factor for result */
                                         /* mix input with eref & reverb result by 1/3 */
       /* mix left channel */
       r10 = DM(Left_Channel);           /* get current left input sample    */
       r11 = DM(DRY_GAIN_LEFT);          /* scale between 0x0 and 0x7FFFFFFF */
       r10 = r10 * r11(ssf);             /* x(n) *(G_left)                   */
       mrf = r2 * r10(ssf);              /* 0.33 * (G_left) * x(n)           */

       r1 = dm(reverb_left);             /* get reverb result                */
       r11 = DM(WET_GAIN_LEFT);          /* scale reverb between 0x0 and 0x7FFFFFFF */
       r1 = r1 * r11 (ssf);              /* x_reverb(n) * RG_left            */
       mrf = mrf + r1*r2 (ssf);          /* add reverb to input sample       */
       r10 = mr1f;                       /* 0.33*x(n) +0.33*x(rev_result)    */

       r1 = dm(predelay_output);
       r4 = DM(PREDEL_GAIN_LEFT);        /* scale between 0x0 and 0x7FFFFFFF */
       r3 = r1 * r4 (ssf);               /* x_er(n) * (ER_G_left)            */
       mrf = mrf + r3*r2 (ssfr);         /* yL(n)=0.33*x(n) +0.33*x(rev_result) + 0.33*x(ear_ref) */
       r10 = mr1f;
       dm(Left_Channel) = r10;           /* output left result */

       /* mix right channel */
       r10 = DM(Right_Channel);          /* get current right input sample    */
       r11 = DM(DRY_GAIN_RIGHT);         /* scale between 0x0 and 0x7FFFFFFF */
       r10 = r10 * r11(ssf);             /* x(n) *(G_right)                  */
       mrf = r2 * r10(ssf);              /* 0.33 * (G_right) * x(n)          */

       r1 = dm(reverb_right);            /* get reverb result                */
       r11 = DM(WET_GAIN_RIGHT);         /* scale reverb between 0x0 and 0x7FFFFFFF */
       r1 = r1 * r11 (ssf);              /* x_reverb(n) * RG_right           */
       mrf = mrf + r1*r2 (ssf);          /* add reverb to input sample       */
       r10 = mr1f;                       /* 0.33*x(n) +0.33*x(rev_result)    */

       r1 = dm(predelay_output);
       r4 = DM(PREDEL_GAIN_RIGHT);       /* scale between 0x0 and 0x7FFFFFFF */
       r3 = r1 * r4 (ssf);               /* x_er(n) * (ER_G_right)           */
       mrf = mrf + r3*r2 (ssfr);         /* yR(n)=0.33*x(n) +0.33*x(rev_result) + 0.33*x(ear_ref) */
       r10 = mr1f;
       dm(Right_Channel) = r10;          /* output right result  */
       rts;
```

3.4 Amplitude-Based Audio Effects

Amplitude-Based audio effects simply involve the manipulation of the amplitude level of the audio signal, from simply attenuating or increasing the volume to more sophisticated effects such as dynamic range compression/expansion. Below is a list of effects that can fall under this category:

Volume Control
Amplitude Panning (Trigonometric / Vector-Based)
Tremolo (Auto Tremolo)
"Ping-Pong" Panning (Stereo Tremolo)
Dynamic Range Control
- ***Compression***
- ***Expansion***
- ***Limiting***

Noise Gating

3.4.1 Tremolo - Digital Stereo Panning Effect

Tremolo consists of *panning* the output result between the left and right output stereo channels at a slow periodic rate. This is achieved by allowing the output panning to vary in time periodically with a low frequency sinusoid. This example pans the output to the left speaker for positive sine values and pans the output to the right speaker for negative sine values (Figure 85). The analog version of this effect was used frequently on guitar and keyboard amplifiers manufactured in the '70s. A mono version of this effect (Figure 84) can be done easily by modifying the code to place the tremolo result to both speakers instead of periodically panning the result. The I/O difference equation is as follows:

$$y(n) = x(n) * \sin(2\pi f_{cycle} t)$$, Mono Tremolo

Figure 84.
Mono Implementation of the Tremolo Effect

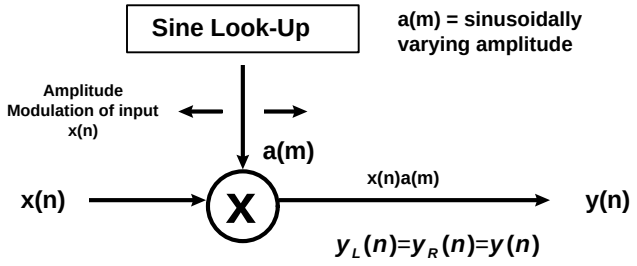
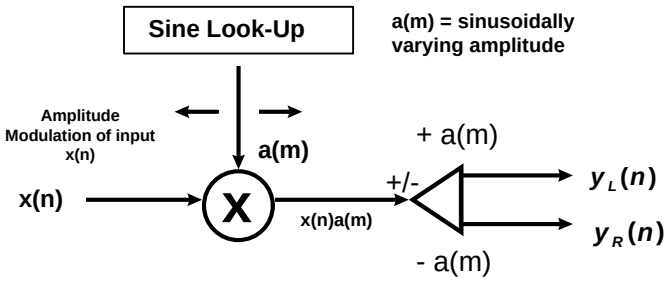


Figure 85.
Stereo Implementation of the Tremolo Effect



For Stereo Tremolo:
when a(m) > 0
x(n)a(m) panned to left channel
right channel is zero
when a(m) < 0
x(n)a(m) panned to right channel
left channel is zero

Example Stereo Tremolo Implementation on the ADSP-21065L

```
tremolo_effect:
      r1 = DM(left_input);
      r1 = ashift r1 by -1;
      r2 = DM(right_input);
      r2 = ashift r2 by -1;
      r3 = r2 + r1;

/* generated sine value from wavetable generator, where r4 = sin(2*pi*fc*t)   */
      r4 = dm(sine_value);
      mrf = r3 * r4 (SSFR);
      r5 = mr1f;
      r4 = pass r4;           /* read current sine value to pan left or right */
                              /* if + then pan left, if - then pan right */
      IF LE JUMP pan_right_channel;

pan_left_channel:
      DM(left_output) = r5;
      r6 = 0x00000000;
      DM(right_output) = r6;
      JUMP done;

pan_right_channel:
      r6 = 0x00000000;
      DM(left_output) = r6;
      DM(right_output) = r5;

done: rts;
```

3.4.2 **Signal Level Measurement**

There are many ways to measure the amplitude of a signal. The technique described below uses a simple signal averaging algorithm to determine the signal level. It rectifies the incoming signal and averages it with the 63 previous rectified samples. Notice, however, that it only requires 6 instructions to average 64 values. This is because we are not recalculating the summation of 64 values and dividing this sum by 64 but rather updating a running average. This is how it works:

$$x_{averageold} = \frac{x[n-64]+x[n-63]+x[n-62]+\ldots+x[n-1]}{64}$$

$$x_{averageold} = \frac{x[n-64]}{64}+\frac{x[n-63]}{64}+\frac{x[n-62]}{64}+\ldots+\frac{x[n-1]}{64}$$

$$x_{averagenew} = \frac{x[n-63]+x[n-62]+x[n-61]+\ldots+x[n]}{64}$$

$$x_{averagenew} = \frac{x[n-63]}{64}+\frac{x[n-63]}{64}+\frac{x[n-61]}{64}+\ldots+\frac{x[n]}{64}$$

$$x_{averagenew} - x_{averageold} = \frac{x[n]}{64}-\frac{x[n-64]}{64}$$

$$x_{averagenew} = x_{averageold} + \frac{x[n]}{64}-\frac{x[n-64]}{64}$$

This algorithm needs to be run 64 times before $x_{averageold}$ contains a valid signal average value.

```
/* set up variables */

.segment /dm dm_variables;
.var average_line[64];
.endseg;


/*=====================================================================
      Amplitude Measurement

      f15 = 1/64
      f0 = current sample
      f14 = current amplitude
      i7 = pointer to average_line
=====================================================================*/

Amplitude:
      f0 = abs f0;      /* take absolute value of incoming sample */
      f0 = f0 * f15;    /* divide incoming sample by length of average line */
      f1 = dm(i7,0);    /* fetch last value in average line */
      f14 = f14 + f0;   /* add it to the running average value */
      rts(db);          /* delayed return from subroutine */
      f14 = f14 - f1;   /* subtract new sample from running average */
      dm(i7,1) = f0;    /* store new sample over old sample in average line */
```

3.4.3 **Dynamics Processing**

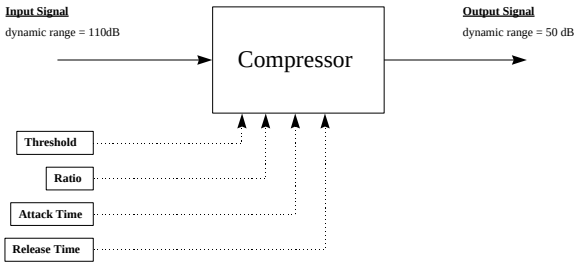
Dynamic processing algorithms are used to change the dynamic range of a signal. This means altering the distance in volume between the softest sound and the loudest sound in a signal. There are two types of dynamic processing algorithms : compressors/limiters and expanders.

3.4.3.1 Compressors and Limiters

The function of a compressor and limiter is to keep the level of a signal within a specific dynamic range. This is done using a technique called *gain reduction*. A gain reduction circuit reduces the amount of additional gain above a threshold setting by a certain ratio (Stark, 1996). The ultimate objective is to keep the signal from going past a specific level.

Compressors and limiters have many applications. They are used to limit the dynamic range of a signal so it can be transmitted through a medium with a limited dynamic range. An expander (covered later in this section) can then be used on the other side to expand the dynamic range back to its original levels. Compressors are also widely used in the recording industry to prevent signals from distorting as a result of overdriven mixer circuitry.

Compressors



Compressors are used to 'compress' the dynamic range of a signal

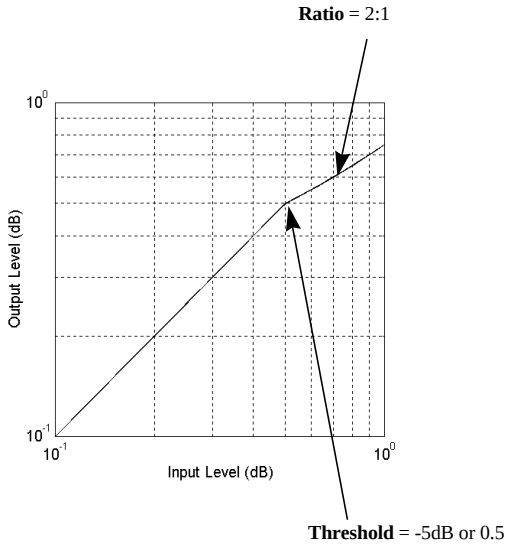
Parameters

Threshold : the level at which the dynamics processor begins adjusting the volume of the signal

Compression Ratio : level comparison of the input and output signals of the dynamics processor past the threshold

Attack Time : The amount of time it takes once the input signal has passed the threshold for the dynamics processsor to begin attenuating the signal

Release Time : The amount of time it takes once the input signal has passed below the threshold for the dynamics processor to stop attenuating the signal



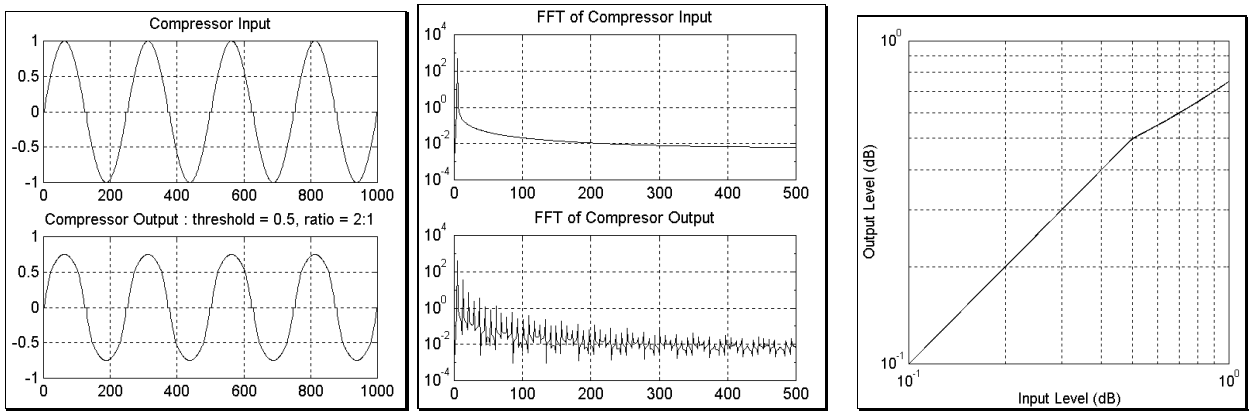
There are two primary parameters for a compressor : threshold and ratio. The threshold is the signal level at which the gain reduction begins and the ratio is the amount of gain reduction that takes place past the threshold. A ratio of 2:1, for example, would reduce the signal by a factor of two when it passed the threshold level as seen in the first compressor example below.

Two other parameters commonly found in compressors are attack time and release time. The attack is the amount of time it takes the compressor to begin compressing a signal once it has crossed the threshold. This helps preserve the natural dynamics of a signal. The release time, on the other hand, is the amount of time it takes the compressor to stop attenuating the signal once its level has passed below the threshold.

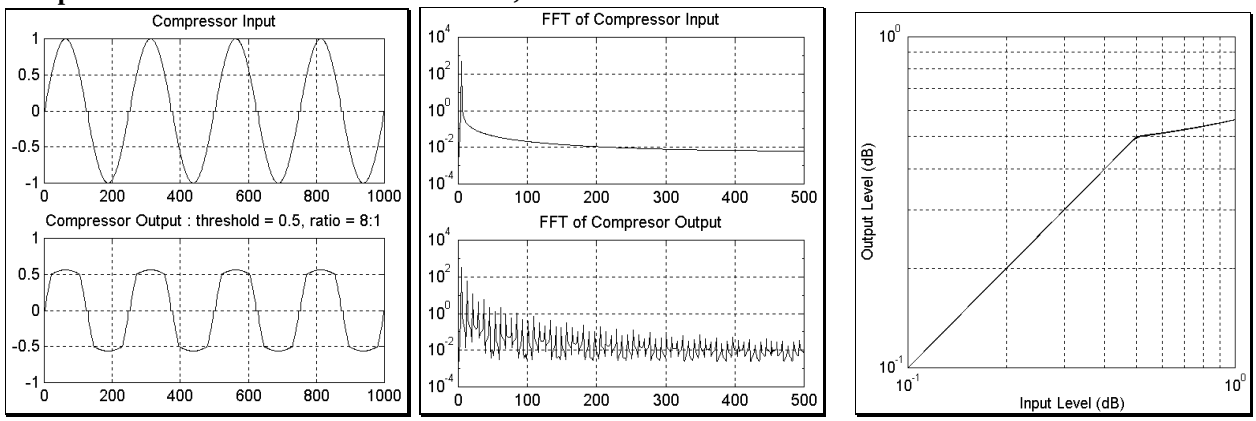
A compressor with a ratio greater than about 10:1 is considered a limiter (Stark, 1996). The effect of a limiter is more like a clipping effect than a dampening effect of a low-ratio compressor. This clipping effect can add many gross harmonics to a signal as seen in the examples below. These harmonics increase in number and amplitude as the threshold level is lowered.

The figures on the following page show the example input and output waveforms, FFTs and gain ratios of some different compressor configurations.

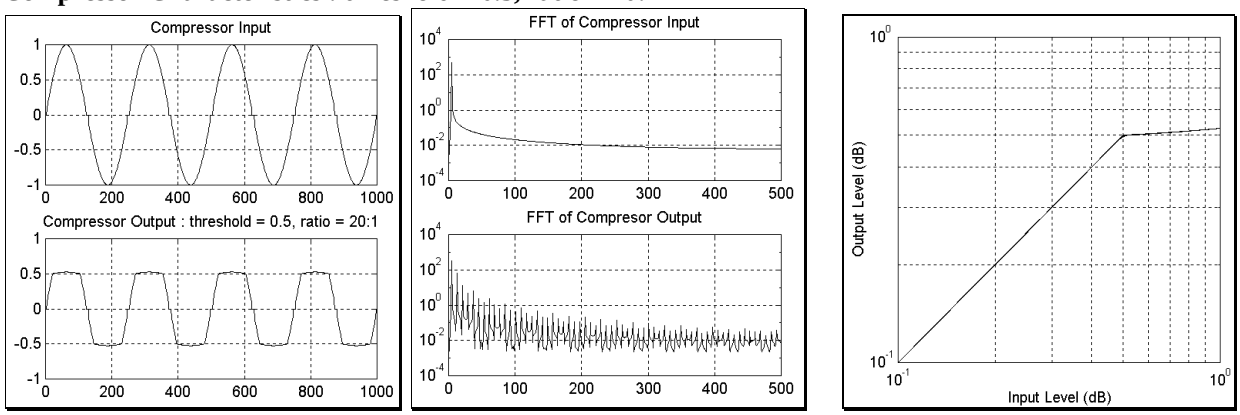
Compressor Characteristics : threshold = 0.5, ratio = 2:1



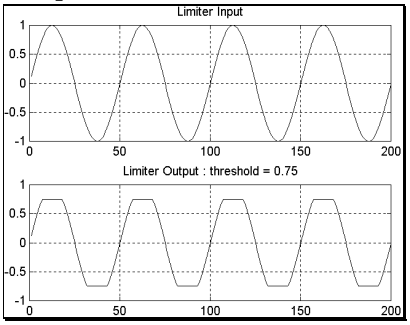
Compressor Characteristics : threshold = 0.5, ratio = 8:1

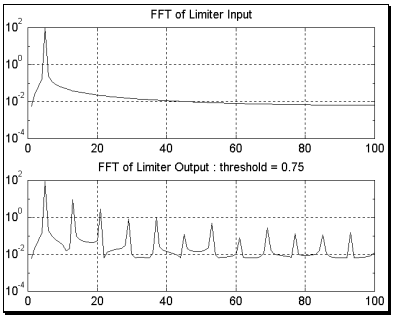


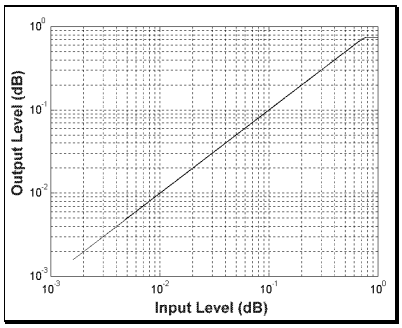
Compressor Characteristics : threshold = 0.5, ratio = 20:1



Compressor Characteristics : threshold = 0.75, ratio = ∞:1







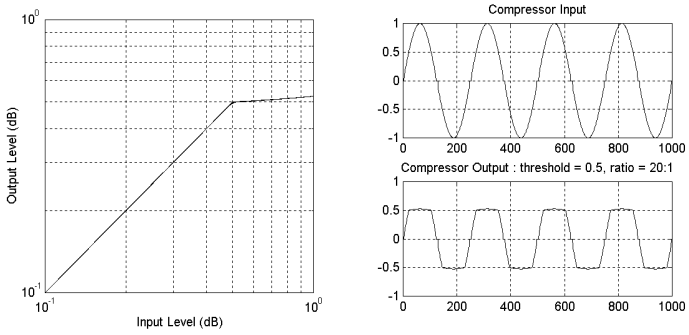
The code example below is a simple stereo compressor. This implementation does not use the attack and release parameters.

Stereo Compressor Implementation on an Analog Devices' ADSP21065L

```
/*     Stereo Compressor

       inputs:
       f2 = left channel data
       f3 = right channel data

       outputs:
       f2 = compressed left channel data
       f3 = compressed right channel data
*/

Compressor:
       f0 = 0.05;              /* f0 = ratio = 1/20 */
       f1 = 0.5;               /* f1 = threshold = 0.5 */
       f4 = abs f2;
       comp(f4,f1);            /* Is left channel past threshold? */
       if LT jump CheckRight;  /* If not, check the 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 contains compressed left channel*/
CheckRight:
       f4 = abs f3;
       comp(f4,f1);            /* Is right channel past threshold? */
       if LT rts;              /* if not, return from subroutine */
       f4 = f4 - f1;           /* signal = signal - threshold */
       f4 = f4 * f0;           /* signal = signal * ratio */
       rts (db);               /* delayed return from subroutine */
       f4 = f4 + f1;           /* signal = signal + threshold *
       f3 = f4 copysign f3;    /* f2 now contains compressed right channel*/
```

Limiters

A limiter is a compressor with a compression ratio greater than about 10:1



The following code example is a stereo limiter with a ratio of 1:∞ - in other words, it clips the signal at a certain threshold. As seen below, this is extremely simple to do using the **clip** function.

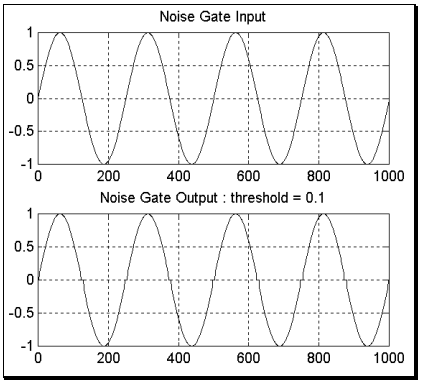
Stereo Limiter Implementation on an Analog Devices' ADSP21065L

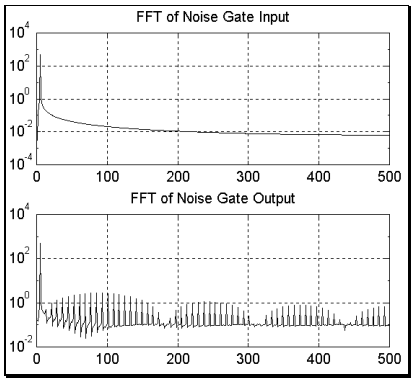
```
/*    Stereo Limiter

      Inputs:
      f2 = left channel data
      f3 = right channel data

      Outputs:
      f2 = limited left channel data
      f3 = limited right channel data
 */

Limit:
      f1 = 0.75;                    /* Threshold = .75 */
      rts (db);                     /* delayed return from subroutine */
      f2 = clip f2 by f1;           /* Limit left channel */
      f3 = clip f3 by f1;           /* Limit right channel */
```

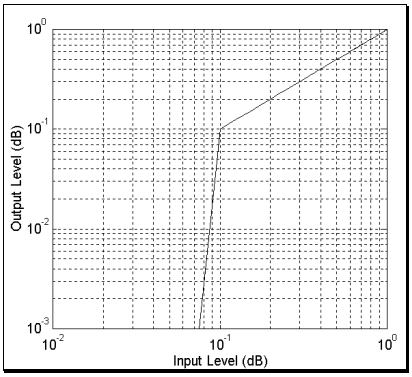
3.4.3.2 Noise Gate/Downward Expander

A noise gate or downward expander is used to reduce the gain of a signal below a certain threshold. This is useful for reducing and eliminating noise on a line when no signal is present. The difference between a noise gate and a downward expander is similar to the difference between a limiter and a compressor. A noise gate cuts signals that fall below a certain threshold while a downward expander has a ratio at which it dampens signals below a threshold.

Noise Gate Characteristics : Threshold = 0.1

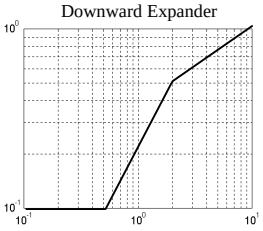


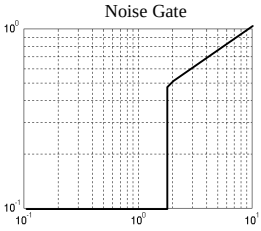




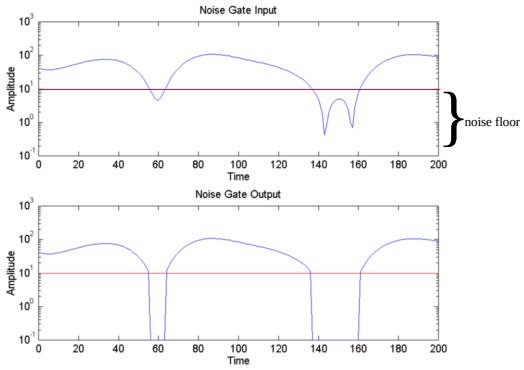
Noise Gates

Compressor : Limiter :: Downward Expander : Noise Gate





Noise Gate Example



The following code example is a stereo noise gate. This algorithm turns off the signal if its amplitude is below a certain level using RMS level detection of a signal to determine if the signal itself should be turned on or not.

Stereo Noise Gate Implementation on an Analog Devices' ADSP21065L

```
/***    NOISE_GATE.ASM    ************************************************************************
*                                                                                                 *
*       ADSP-21065L EZLAB  Noise Gate Effect Program                                              *
*       Developed using ADSP-21065L EZ-LAB Evaluation Platform                                    *
*                                                                                                 *
*                                                                                                 *
*       What the noise gate does?                                                                 *
*       -------------------------                                                                 *
*       Reduces the amount of gain below a certain threshold to reduce or eliminate               *
*       noise produced when no audio signal is present, while still allowing the                  *
*       signal to pass thru.  This is useful after processing multiple audio effects              *
*       that can introduce noise above the noise floor of the AD1819a DACs.                       *
*                                                                                                 *
*       Parameters:                                                                               *
*       ----------                                                                                *
*       Threshold:  The level at which the noise gate processor begins decreasing the             *
*       volume of the signal.                                                                     *
*       NOTE: Threshold values are in RMS.  This routine calculates the RMS of the                *
*       audio signal in determining if low level noise should be removed.  A running             *
*       average is not sufficent otherwise the audio signal will be severely distorted            *
*                                                                                                 *
*       Future Parameters that will be added in Rev 2.0:                                          *
*       ------------------------------------------------                                          *
*       Attack Time:  The amount of time it takes once the input signal has passed the            *
*       threshold for the dynamics processor to begin attenuating the signal.                     *
*       Release Time:  The amount of time it takes once the input signal has passed               *
```

```
*       below the threshold for the dynamics processor to stop attenuating the signal      *
*                                                                                           *
*       The audio data is sent out to the AD1819A Line Outputs                              *
*                                                                                           *
*       ++ TABs set to 8 characters in VDSP IDE under TOOLS->>OPTIONS Menu                  *
*                                                                                           *
*                                           Dan Ledger & John Tomarakos                     *
*                                           ADI DSP Applications Group                      *
*                                           Revision 1.0                                    *
*                                           11/19/98                                        *
*                                                                                           *
*********************************************************************************************/

/* ADSP-21065L System Register bit definitions */
#include       "def21065l.h"
#include       "new65Ldefs.h"

.EXTERN        Left_Channel;
.EXTERN        Right_Channel;

.GLOBAL        Noise_Gate;
.GLOBAL        select_threshold;
.GLOBAL        init_averaging_buffers;


.segment /dm    noisegt;

.var    IRQ1_counter = 0x00000003;
.var    threshold = 0.04;
.var    Left_RMS_Result;
.var    Right_RMS_Result;
.var    left_float;
.var    right_float;
.var    left_RMS_squared = 0.0;
.var    right_RMS_squared = 0.0;
.var    left_RMS_line[500];           /* used to detect the RMS value of the left channel audio signal */
.var    right_RMS_line[500];          /* used to detect the RMS value of the right channel audio signal
*/

.endseg;


.segment /pm pm_code;

init_averaging_buffers:
       B6 = left_RMS_line;
       L6 = @left_RMS_line;          /* delay-line buffer pointer and length */
       m6 = 1;

       LCNTR = L6;                   /* clear delay line buffer to zero */
        DO clrDlineL UNTIL LCE;
clrDlineL:         dm(i6, m6) = 0;

       B7 = right_RMS_line;
       L7 = @right_RMS_line;         /* delay-line buffer pointer and length */
       m7 = 1;

       LCNTR = L7;                   /* clear delay line buffer to zero */
        DO clrDlineR UNTIL LCE;
clrDlineR:         dm(i7, m7) = 0;

       RTS;

/*************************************************************************
 *                                                                       *
 *                            STEREO NOISE GATE ROUTINE                  *
 *                                                                       *
 *             inputs:                                                   *
 *             f2 = left channel data                                    *
 *             f3 = right channel data                                   *
 *                                                                       *
 *             outputs:                                                  *
 *             f2 = compressed left channel data                         *
 *             f3 = compressed right channel data                        *
 *                                                                       *
```

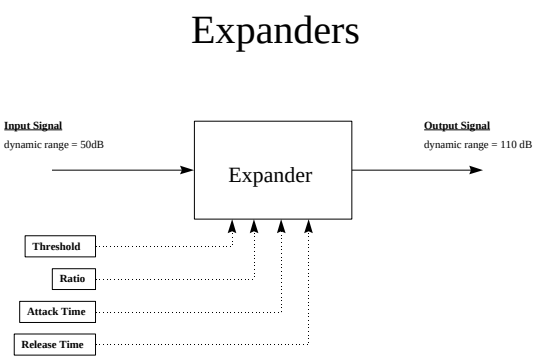
```
 ************************************************************************************/

Noise_Gate:
       r2 = DM(Left_Channel);        /* left input sample */
       r3 = DM(Right_Channel);       /* right input sample */

       r1 = -31;                     /* scale the sample to the range of +/-1.0 */

       f2 = float r2 by r1;          /* convert left fixed point sample to floating point */

       f3 = float r3 by r1;          /* convert right fixed point sample to floating point */

       DM(left_float) = f2;          /* save floating point samples temporarily */
       DM(right_float) = f3;

       f15 = 0.002;                  /* 1/500 = 0.002*/
       f5 = DM(threshold);           /* f1 = Threshold = 0.1 */

RMS_left_value:
       f0 = abs f2;                  /* take absolute value of incoming left sample */
       f1 = f0;                      /* get ready to square the input */
       f0 = f0 * f1;                 /* f0 = square(abs(x)) */
       f0 = f0 * f15;                /* divide incoming squared sample by length of RMS line */
       f1 = dm(i6,0);                /* fetch oldest value in RMS line */
       f10 = DM(left_RMS_squared);   /* get previous running average of the squares of the input */
       f10 = f10 + f0;               /* add scaled squared input to the running average value */
       f10 = f10 - f1;               /* subtract oldest squared sample from running average */
       DM(left_RMS_squared) = f10;   /* save new running average of the square of the inputs samples */
       dm(i6,1) = f0;                /* store new scaled squared sample over old sample in RMS line */

       /* calculate square root of new average in f10 based on the Newton-Raphson iteration algorithm */
       f8 = 3.0;
       f2 = 0.5;
       f4 = RSQRTS f10;              /* Fetch seed */
       f1 = f4;
       f12 = f4 * f1;                /* F12=X0^2 */
       f12 = f12 * f0;               /* F12=C*X0^2 */
       f4 = f2 * f4, f12 = f8 - f12; /* F4=.5*X0, F10=3-C*X0^2 */
       f4 = f4 * f12;                /* F4=X1=.5*X0(3-C*X0^2) */
       f1 = f4;
       f12 = f4 * f1;                /* F12=X1^2 */
       f12 = f12 * f0;               /* F12=C*X1^2 */
       f4 = f2 * f4, f12 = f8 - f12; /* F4=.5*X1, F10=3-C*X1^2 */
       f4 = f4 * f12;                /* F4=X2=.5*X1(3-C*X1^2) */
       f1 = f4;
       f12 = f4 * f1;                /* F12=X2^2 */
       f12 = f12 * f0;               /* F12=C*X2^2 */
       f4 = f2 * f4, f12 = f8 - f12; /* F4=.5*X2, F10=3-C*X2^2 */
       f4 = f4 * f12;                /* F4=X3=.5*X2(3-C*X2^2) */
       f10 = f4 * f10;               /* X=sqrt(Y)=Y/sqrt(Y) */
       DM(Left_RMS_Result) = f10;

gate_left:
       f2 = DM(left_float);
       f10 = abs f10;                /* get absolute value of running average */
       comp(f10,f5);                 /* compare to desired threshold */
       if LT f2 = f2 - f2;           /* if left channel < threshold, left channel = 0.0 */

       /* send gated results to left DAC channel */
       r1 = 31;                      /* scale the result back up to MSBs */
       r2 = fix f2 by r1;            /* convert back to fixed point number */
       DM(Left_Channel) = r2;

RMS_right_value:
       f0 = abs f3;                  /* take absolute value of incoming right sample */
       f1 = f0;                      /* get ready to square the input */
       f0 = f0 * f1;                 /* f0 = square(abs(x)) */
       f0 = f0 * f15;                /* divide incoming squared sample by length of RMS line */
       f1 = dm(i7,0);                /* fetch oldest value in RMS line */
       f10 = DM(right_RMS_squared);  /* get previous running average of the squares of the input */
       f10 = f10 + f0;               /* add scaled squared input to the running average value */
       f10 = f10 - f1;               /* subtract oldest squared sample from running average */
       DM(right_RMS_squared) = f10;  /* save new running average of the square of the inputs samples */
       dm(i7,1) = f0;                /* store new scaled squared sample over old sample in RMS line */
```

```
        /* caclulate square root of new average in f10 based on the Newton-Raphson iteration algorithm */
        f8 = 3.0;
        f2 = 0.5;
        f4 = RSQRTS f10;              /* Fetch seed */
        f1 = f4;
        f12 = f4 * f1;                /* F12=X0^2 */
        f12 = f12 * f0;               /* F12=C*X0^2 */
        f4 = f2 * f4, f12 = f8 - f12; /* F4=.5*X0, F10=3-C*X0^2 */
        f4 = f4 * f12;                /* F4=X1=.5*X0(3-C*X0^2) */
        f1 = f4;
        f12 = f4 * f1;                /* F12=X1^2 */
        f12 = f12 * f0;               /* F12=C*X1^2 */
        f4 = f2 * f4, f12 = f8 - f12; /* F4=.5*X1, F10=3-C*X1^2 */
        f4 = f4 * f12;                /* F4=X2=.5*X1(3-C*X1^2) */
        f1 = f4;
        f12 = f4 * f1;                /* F12=X2^2 */
        f12 = f12 * f0;               /* F12=C*X2^2 */
        f4 = f2 * f4, f12 = f8 - f12; /* F4=.5*X2, F10=3-C*X2^2 */
        f4 = f4 * f12;                /* F4=X3=.5*X2(3-C*X2^2) */
        f10 = f4 * f10;               /* X=sqrt(Y)=Y/sqrt(Y) */
        DM(Right_RMS_Result) = f10;

gate_right:
        f3 = DM(right_float);
        f10 = abs f10;
        comp(f10,f5);
        if LT f3 = f3 - f3;           /* if right channel < threshold, right channel = 0 */

        /* send gated results to right DAC channel */
        r1 = 31;                      /* scale the result back up to MSBs */
        r3 = fix f3 by r1;            /* convert back to fixed point number */
        DM(Right_Channel) = r3;
        rts;
```

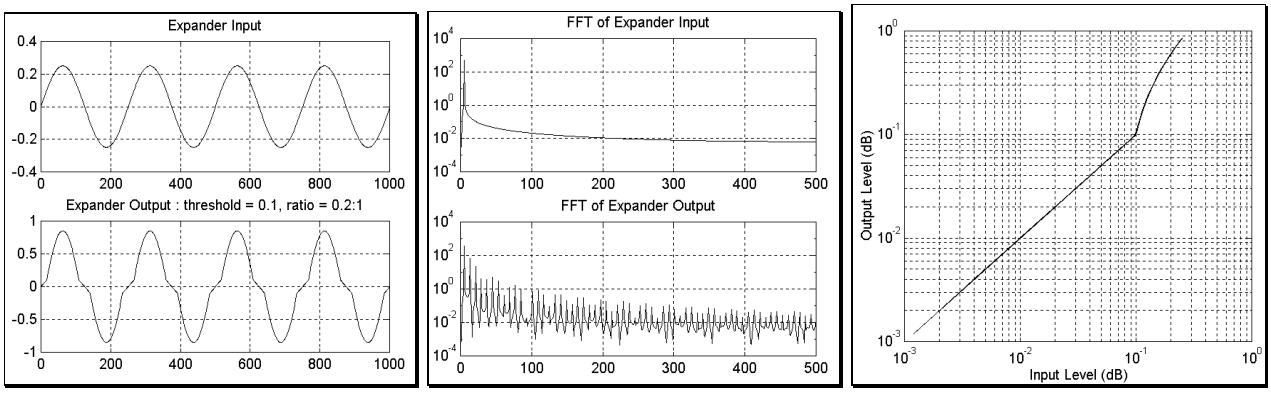
3.4.3.3 Expanders

An expander is a device used to increase the dynamic range of a signal and complement compressors. For example, a signal with a dynamic range of 70 dB might pass through an expander and exit with a new dynamic range of 100 dB. These can be used to restore a signal that was altered by a compressor.

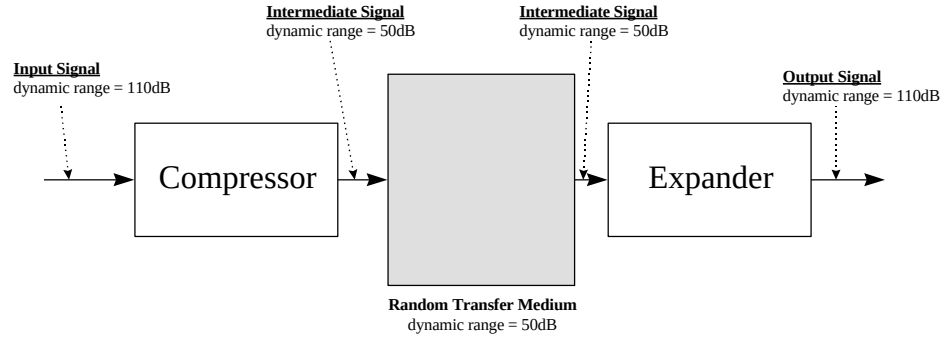
Below are the properties of an expander.



Expanders are used to ‘expand’ the dynamic range of a signal



Compressor-Expander Application Example




```
/***    COMPANDER.ASM    ************************************************************************
*                                                                                               *
*       ADSP-21065L EZ-LAB Companding Effect Program                                            *
*       Developed using ADSP-21065L EZ-LAB Evaluation Platform                                  *
*                                                                                               *
*                                                                                               *
*       What the compander does?                                                                *
*       ------------------------                                                                *
*       Combination of a compressor and expander.  The peaks signal levels above the            *
*       upper threshold are compressed, while lower level signals above the lower               *
*       threshold are expanded.                                                                 *
*                                                                                               *
*       Parameters:                                                                             *
*       ----------                                                                              *
*       Threshold:  The level at which the dynamics processor begins adjusting the              *
*       volume of the signal.                                                                   *
*       Compression Ratio:  Level comparison of the input and output signals of the             *
*        dynamics processor past the threshold.                                                 *
*                                                                                               *
*       Future Parameters that will be added in Rev 2.0:                                        *
*       ------------------------------------------------                                        *
*       Attack Time:  The amount of time it takes once the input signal has passed the          *
*       threshold for the dynamics processor to begin attenuating the signal.                   *
*       Release Time:  The amount of time it takes once the input signal has passed             *
```

```
*       below the threshold for the dynamics processor to stop attenuating the signal      *
*****************************************************************************************/

/* ADSP-21065L System Register bit definitions */
#include       "def21065l.h"
#include       "new65Ldefs.h"

.EXTERN        Left_Channel;
.EXTERN        Right_Channel;

.GLOBAL        Stereo_Compander;
.GLOBAL        select_compander_ratios;
.GLOBAL        select_compander_thresholds;


.segment /dm    compand;

.var    IRQ1_counter = 0x00000003;
.var    IRQ2_counter = 0x00000003;

.var    comp_ratio = 0.05;
.var    comp_threshold = 0.5;
.var    expan_ratio = 1.5;
.var    expan_threshold = 0.2;

.endseg;


.segment /pm pm_code;

/*************************************************************************
 *                                                                       *
 *                        STEREO COMPANDER ROUTINE                       *
 *                                                                       *
 *************************************************************************/

Stereo_Compander:
       r2 = DM(Left_Channel);        /* left input sample */
       r3 = DM(Right_Channel);       /* right input sample */

       r1 = -31;                     /* scale the sample to the range of +/-1.0 */

       f2 = float r2 by r1;          /* convert fixed point sample to floating point */

       f3 = float r3 by r1;          /* convert fixed point sample to floating point */

       f0 = DM(comp_ratio);          /* f0 = ratio = 1/20 */
       f1 = DM(comp_threshold);      /* f1 = threshold = 0.5 */

compress_left:
       f4 = abs f2;
       comp(f4,f1);                  /* Is left channel past threshold? */
       if LT jump compress_right;    /* If not, check the 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 contains compressed left channel*/

compress_right:
       f4 = abs f3;
       comp(f4,f1);                  /* Is right channel past threshold? */
       if LT jump expansion;         /* if not, return from subroutine */
       f4 = f4 - f1;                 /* signal = signal - threshold */
       f4 = f4 * f0;                 /* signal = signal * ratio */
       f4 = f4 + f1;                 /* signal = signal + threshold */
       f3 = f4 copysign f3;          /* f3 now contains compressed right channel*/

expansion:
       f0 = DM(expan_ratio);         /* f0 = ratio = 1.4 */
       f1 = DM(expan_threshold);     /* f1 = threshold = 0.2 */

expand_left:
       f4 = abs f2;
       comp(f4,f1);                  /* Is left channel past threshold? */
       if LT jump expand_right;      /* If not, check the 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 contains compressed left channel*/

expand_right:
        f4 = abs f3;
        comp(f4,f1);                  /* Is right channel past threshold? */
        if LT jump finish_processing; /* if not, return from subroutine */
        f4 = f4 - f1;                 /* signal = signal - threshold */
        f4 = f4 * f0;                 /* signal = signal * ratio */
        f4 = f4 + f1;                 /* signal = signal + threshold */
        f3 = f4 copysign f3;          /* f3 now contains compressed right channel*/

finish_processing:
        r1 = 31;                      /* scale the result back up to MSBs */
        r2 = fix f2 by r1;            /* convert back to fixed point number */
        r3 = fix f3 by r1;            /* convert back to fixed point number */

        /* send companded results to left and right DAC channels */
        DM(Left_Channel) = r2;
        DM(Right_Channel) = r3;

        rts;
```

3.5 Sound Synthesis Techniques

Sound synthesis is a technique used to create specific waveforms. It is widely used in the audio market in products like sound cards and synthesizers to digitally recreate musical instruments and other sound effects. The most simple forms of sound synthesis such as FM and Additive synthesis use basic harmonic recreation of a sound using the addition and multiplication of sinusoids of varying frequency, amplitude and phase. Sample playback and wavetable synthesis use digital recordings of a waveform played back at varying frequencies to achieve life-like reproductions of the original sound. Subtractive Synthesis and Physical Modeling attempt to simulate the physical model of an acoustic system.

3.5.1 **Additive Synthesis**

Fourier theory dictates that any periodic sound can be constructed of sinusoids of various frequency, amplitude and phase [25]. Additive synthesis is the processes of summing such sinusoids to produce a wide variety of envelopes. By varying the three fundamental properties : frequency, amplitude and phase over time, additive synthesis can accurately reproduce a variety instruments.

In comparison to all other synthesis techniques, additive synthesis can require a significant amount of processing power based on the number of sinusoidal oscillators used. There is a direct relationship between the number of harmonics generated and the number of processor cycles required. Below is the basic formula.

$$y(n) = A_1 \sin(2\Pi f_1 n + f_1) + A_2 \sin(2\Pi f_2 n + f_2) + A_3 \sin(2\Pi f_3 n + f_3)...$$

3.5.2 **FM Synthesis**

FM Synthesis is similar to additive synthesis in that it uses simple sinusoids to create a wide range of sounds. FM synthesis, however, uses one finite formula to create an infinite number harmonics. The FM synthesis equation shown below uses a fundamental sinusoid which is modulated by another sinusoid.

$$y(n) = A(n)\sin\left(2\Pi f_c n + I(n)\sin(2\Pi f_m n)\right)$$

When this equation is expanded, we can see that an infinite number of harmonics are created.

$$y(n) = J_1(n)\sin(2\Pi f_c n)$$

$$+J_1(n)[\sin(2\Pi(f_c+f_m)n)-\sin(2\Pi(f_c-f_m)n)]$$
$$+J_2(n)[\sin(2\Pi(f_c+2f_m)n)-\sin(2\Pi(f_c-2f_m)n)]$$
$$+J_3(n)[\sin(2\Pi(f_c+3f_m)n)-\sin(2\Pi(f_c-3f_m)n)]... \quad [2]$$

Because this method is very computationally efficient, it is widely used in the sound card and synthesizer industry.

3.5.3 Wavetable Synthesis

Wavetable synthesis is a popular and efficient technique for synthesizing sounds, especially in sound cards and synthesizers. Using a lookup table of pre-recorded waveforms, the wavetable synthesis engine repeatedly plays the desired waveform or combinations of multiple waveforms to simulate the timbre of an instrument. The looped playback of the sample can also be modulated by an amplitude function which controls its attack, decay, sustain and release to create an even more realistic reconstruction of the original instrument.

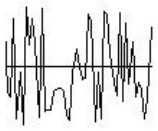


Figure 14:
One waveform period stored in wavetable.

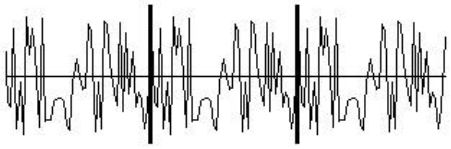


Figure 15:
Waveforms added together and repeated over time

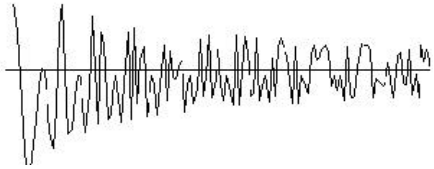


Figure 16 :
Repeated waveform modulated by an amplitude envelop.

This method of synthesis is simple to implement and is computationally efficient. In a DSP, the desired waveforms can be loaded into a circular buffers to allow for zero-overhead looping. The only real computational operations will be adding multiple waveforms, calculating the amplitude envelope and modulating the looping sample with it. The downside of wave-table synthesis is that it is difficult to approximate rapidly changing spectra.[1]

3.5.4 Sample Playback

Sample Playback is another computationally efficient synthesis technique that yields extremely high sound quality. An entire sample is stored in memory for each instrument which is played back at a selected pitch. Often times, these sample will have loop points within them which can be used to alter the duration of the sustain thus giving an even more life-like reproduction of the sound.

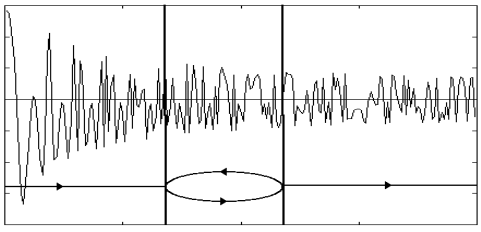


Figure 17

Although this method is capable of producing extremely accurate reproductions of almost any instrument, it requires large amounts of memory to hold the sampled instrument data. For example, to duplicate the sound of a grand piano, the sample stored in memory would have to be about 5 seconds long. If this sample were stereo and sampled at 44.1kHz, this single instrument would require 441,000 Words of memory! To recreate many octaves of a piano, the system would require multiple piano samples because slowing down a sample of a C5 on a piano to the pitch of a C2 will sound nothing like an actual C2. This technique is widely used in high-end keyboards and sound cards. Just like wavetable synthesis, sample playback requires very little computational power. It can be easily implemented in a DSP using circular buffers with simple loop-point detection.

3.5.5 Subtractive Synthesis

Subtractive synthesis begins with a signal containing all of the required harmonics of a signal and selectively attenuating (or boosting) certain frequencies to simulate the desired sound[2]. The amplitude of the signal can be varied using an envelope function as in the other simple synthesis techniques. This technique is effective at recreating instruments that use impulse-like stimulus like a plucked string or a drum.

4. CONCLUSION

We have explored many of the basics in selecting the ADSP-21065L for use in digital audio applications. There are many different DSPs on the market today and chances are there is one that fits your design needs perfectly. Because of this, it is important to fully understand the type of algorithms and the amount of processing power that an application will require before selecting a DSP. This paper has presented a subset of the expanding number of audio applications for DSPs and provided some insight into their functionality and implementation. As DSPs become faster and more powerful, we will undoubtedly witness new creative and ingenious DSP audio applications.

Appendix - References

[1] Steven W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, San Diego, CA, (1998)

[2] S. J. Orfanidis, *Introduction to Signal Processing*, Chapter 8, Sec 8.2, pp. 355-383, Prentice Hall, Englewood Cliffs, NJ, (1996).

[3] J. G. Proakis & D. G. Manolakis, Introduction To Digital Signal Processing, Macmillan Publishing Company, New York, NY, (1988)

[4] A. V. Oppenheim and R. W. Schafer*, Discrete-Time Signal Processing*, Prentice Hall, Englewood Cliffs, NJ, (1989)

[5] P. Lapsley, J. Bier, A. Shoham and E. A. Lee*, DSP Processor Fundamentals: Architectures and Features*, Berkley Design Technology, Inc., Fremont, CA, (1996)

[6] Jon Dattorro, "Effect Design, Part 2: Delay-Line Modulation and Chorus," *J. Audio Engineering Society*, 10, pp. 764-788, October 1997

[7] Scott Lehman, *Harmony Central Effects Explained*, Internet: Http://www.harmony-central.com/Effects/Articcles, (1996)

[8] Jack W. Crenshaw, *Programmer's Toolbox: Putting It Together*, Embedded Systems Programming, pp. 9-24, August 1995

[9] R. Wilson, "Filter Topologies", *J. Audio Engineering Society*, Vol 41, No. 9, September 1993

[10] Udo Zolzer, "Roundoff Error Analysis of Digital Filters", *J. Audio Engineering Society*, Vol42, No. 4, April 1994

[11] J.A Moorer, "About This Reverberation Business," *Computer Music Journal*, **3**, 13 (1979).

[12] M.R. Schroeder, "Natural Sounding Artificial Reverberation," *J. Audio Eng*, Soc., **10**, p. 219, (1962).

[13] M.R. Schroeder, "Digital Simulation of Sound Transmission in Reverberant Spaces", *J. Acoust. Soc. Am.,* **47**, p. 424, (1970).

[14] D. Griesinger, "Practical Processors and Programs for Digital Reverberation," *Audio in Digital Times, Proc. Audio Eng. Soc. 7th Inter. Conf.,* Toronto, Ont., Canada, May 14th-17th 1989, pp. 187-195.

[15] K. L. Kloker, B. L. Lindsley, C.D. Thompson, "VLSI Architectures for Digital Audio Signal Processing," *Audio in Digital Times, Proc. Audio En g. Soc. 7th Inter. Conf.,* Toronto, Ont., Canada, May 14th-17th 1989, pp. 313-325

[16] K. Bogdanowicz & R. Belcher, "Using Multiple Processor for Real-Time Audio Effects", *Audio in Digital Times, Proc. Audio En g. Soc. 7th Inter. Conf.,* Toronto, Ont., Canada, May 14th-17th 1989, pp. 337-342

[17] Gary Davis & Ralph Jones, *Sound Reinforcement Handbook*, 2nd Edition", **Ch. 14**, pp. 259-278, Yamaha Corporation of America, (1989, 1990)

[18] Analog Devices, Inc, ADSP-2106x SHARC User's Manual, Second Edition, Analog Devices, 3 Technology Way, Norwood, MA (1996)

[19] Analog Devices Whitepaper, *ADSP-21065L: Low-Cost 32-bit Processing for High Fidelity Digital Audio*, Analog Devices, 3 Technology Way, Norwood, MA, November 1997

[20] J. Dattorro, "The Implementation of Digital Filters for High Fidelity Audio", *Audio in Digital Times, Proc. Audio En g. Soc. 7th Inter. Conf.,* Toronto, Ont., Canada, May 14th-17th 1989, pp. 165-180.

[21] W. Chen, "Performance of Cascade and Parallel IIR Filters," *Audio in Digital Times, Proc. Audio En g. Soc. 7th Inter. Conf.,* Toronto, Ont., Canada, May 14th-17th 1989, pp. 148-158

[22] V. Pulkki, "Virtual Sound Source Positioning Using Vector Base Amplitude Panning", *J. Audio Engineering Soc.,* Vol. 45, No. 6, pp. 456-466, June 1997

[23] S. J. Orfanidis, "Digital Parametric Equalizer Design with Prescribed Nyquist-Frequency Gain," *J. Audio Engineering Soc.*, Vol. 45, No. 6, pp. 444 - 455, June 1997

[24] D. C. Massie, “An Engineering Study of the Four-Multiply Normalized Ladder Filter,” *J. Audio Engineering Soc*., Vol.41, No. 7/8, pp. 564-582, July/August 1993

[25] Gregory Sandell, “Perceptual Evaluation of Principal-Component-Based Synthesis of Musical Timbres”, *J. Audio Engineering Soc*., Vol.43, No. 12, pp.1013-1027, December 1995

[26] C. Anderton, *Home Recording for Musicians*, Amsco Publications, New York, NY, (1996)

[27] B. Gibson, The AudioPro Home Recording Course, MixBooks, Emeryville, CA, (1996)

[28] D. P. Weiss, “Experiences with the AT&T DSP32 Digital Signal Processor in Digital Audio Applications,” *Audio in Digital Times, Proc. Audio En g. Soc. 7th Inter. Conf.,* Toronto, Ont., Canada, May 14th-17th 1989, pp. 343-351

[29] J. Bier, P. Lapsley, and G. Blalock, “Choosing a DSP Processor,” *Embedded Systems Programming*, pp. 85-97, (October 1996)

[30] Jon Dattorro, “Effect Design, Part 1: Reverberator and Other Filters,” *J. Audio Engineering Society*, 10, pp. 660-684, September 1997

[31] *ME-5 Guitar Multiple Effects Processor*, User Manual, Roland/Boss Corp., 1990.

[32] Dominic Milano, *Multi-Track Recording, A Technical And Creative Guide For The Musician And Home Recorder*, Reprinted from *The Keyboard Magazine*, **Ch. 2**, pp. 37 - 50. Hal Leonard Books, 8112 W. Bluemound Road, Milwaukee, WI (1988)

[33] R. Bristow-Johnson, “A Detailed Analysis of a Time-Domain Formant-Corrected Pitch-Shifting Algorithm”, *J. Audio Engineering Soc.*, Vol. 43, No. 5, May 1995

[34] R. Adams & T. Kwan, “Theory and VLSI Architectures for Asynchronous Sample Rate Converters*”, J. Audio Engineering Soc*., Vol. 41, No. 7/8, pp. 539-555, July/August 1993

[35] R. Bristow-Johnson, “The Equivalence of Various Methods of Computing Biquad Coefficients for Audio Parametric Equalizers”, presented at AES 97th Convention*, J. Audio Engineering Soc. (Abstracts Preprint 3096)*, Vol 42, pp. 1062-1063, (December 1994)

[36] D. J. Shpak, “Analytical Design of Biquadratic Filter Sections for Parametric Filters”, J. Audio Engineering Soc., Vol 40, No 11, pp. 876-885, (November 1992)

[37] NVision*, THE BOOK: An Engineer’s Guide To The Digital Transition - CH2: Designing A Digital Audio System*, Internet: http://www.nvision1.com/thebook/chapter2.htm, NVision, Inc. Nevada City CA

[38] L. D. Fielder, "Human Auditory Capabilities and Their Consequences in Digital-Audio Converter Design", *Audio in Digital Times, Proc. Audio En g. Soc. 7th Inter. Conf*., Toronto, Ont., Canada, May 14th-17th 1989, pp. 45-62.

[39] A. Chrysafis, "Digital Sine-Wave Synthesis Using the DSP56001/2", Application Note, Semiconductor Products Sector, Motorola, Inc., Austin, TX, (1988)