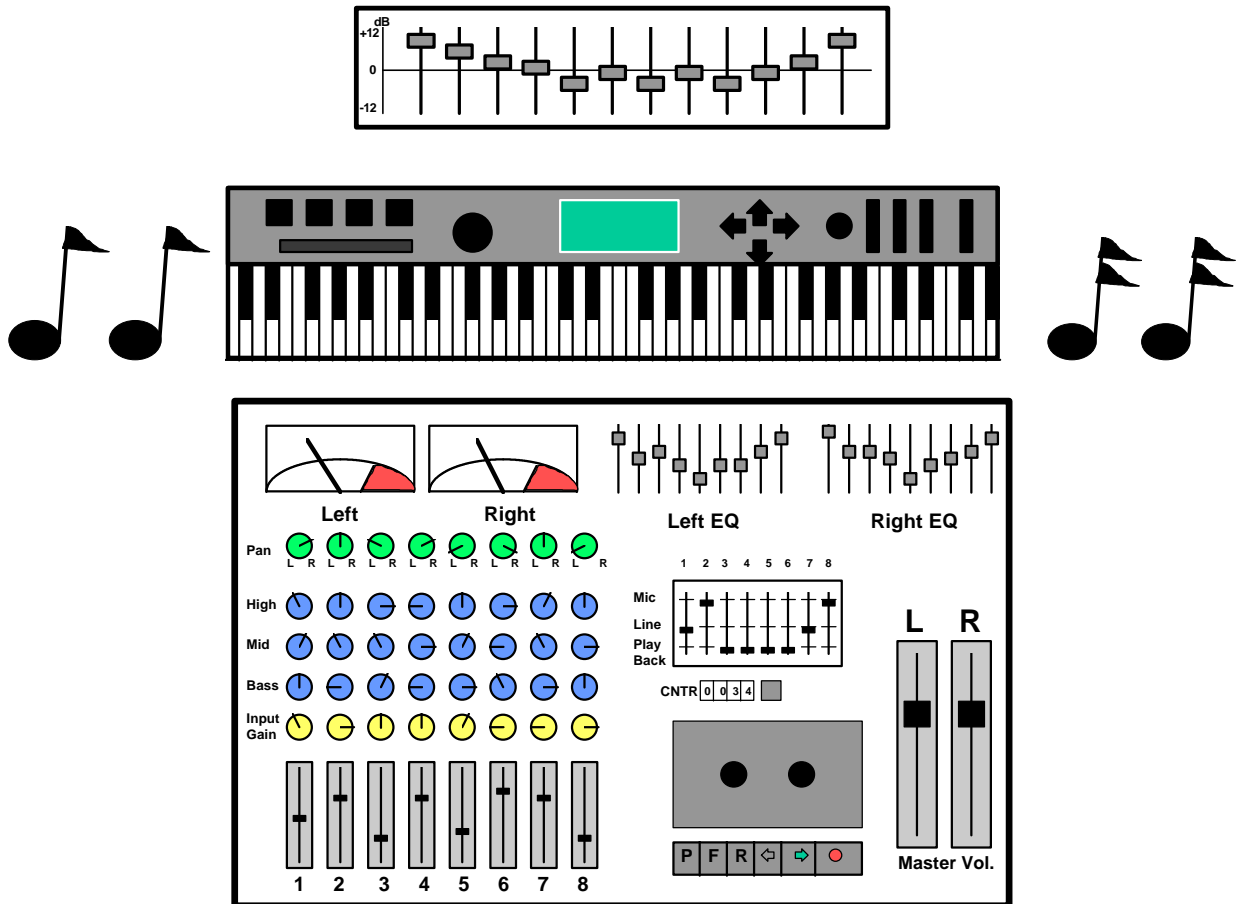


a

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. Comparisons 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	4
1. SELECTING AN AUDIO SIGNAL PROCESSOR.....	5
1.1 GENERAL PURPOSE DIGITAL SIGNAL PROCESSORS AND DECODERS FOR AUDIO	5
1.2 PRICE	ERROR! BOOKMARK NOT DEFINED.
1.3 PROCESSOR SPEED	5
1.4 ON-CHIP MEMORY	7
1.5 I/O CAPABILITIES AND INTERFACES FOR PROCESSING OF AUDIO SAMPLES	7
1.5.1 DMA (Direct Memory Access) Controllers.....	7
1.5.2 Serial Interface to Audio Converters and other Digital Audio Devices	7
1.6 DSP NUMERIC DATA FORMATS : FIXED/FLOATING POINT ARITHMETIC.....	9
1.6.1 16/24/32-Bit Fixed-Point Arithmetic	10
1.6.2 Floating-Point Arithmetic	10
1.7 DOUBLE-PRECISION FIXED POINT ARITHMETIC VERSUS SINGLE-PRECISION ARITHMETIC.....	11
1.8 THE IMPORTANCE OF DYNAMIC RANGE IN DSP-AUDIO PROCESSING	11

2. USEFUL DSP HARDWARE/SOFTWARE BUILDING BLOCKS FOR AUDIO	18
2.1 BASIC ARITHMETIC OPERATIONS	18
2.2 IMPLEMENTING CONVOLUTION WITH ZERO-OVERHEAD LOOPING, MULTIPLY/ACCUMULATE INSTRUCTIONS (MAC), AND DUAL MEMORY FETCHES	18
2.3 HARDWARE CIRCULAR BUFFERING FOR EFFICIENT STORAGE/RETRIEVAL OF AUDIO SAMPLES	19
2.4 ZERO-OVERHEAD LOOPING.....	20
2.5 BLOCK PROCESSING VS. SAMPLE PROCESSING	20
2.6 DELAY-LINES.....	20
2.7 SIGNAL GENERATION WITH LOOK-UP TABLES	21
3. IMPLEMENTING DSP AUDIO ALGORITHMS.....	23
3.1 BASIC AUDIO SIGNAL MANIPULATION.....	23
3.1.1 <i>Volume Control</i>	<i>24</i>
3.1.2 <i>Mixing Multiple Audio Signal Channels</i>	<i>24</i>
3.1.3 <i>Amplitude Panning of Signals to a Left or Right Stereo Field.....</i>	<i>25</i>
3.2 FILTERING TECHNIQUES AND APPLICATIONS.....	29
3.2.1 <i>The FIR Filter</i>	<i>29</i>
3.2.2 <i>The IIR Filter</i>	<i>30</i>
3.2.3 <i>Parametric Filters.....</i>	<i>31</i>
3.2.4 <i>Graphic Equalizers</i>	<i>33</i>
3.2.5 <i>Comb Filters</i>	<i>35</i>
3.2.6 <i>Scaling to Prevent Overflow.....</i>	<i>36</i>
3.3 TIME-DELAY DIGITAL AUDIO EFFECTS.....	37
3.3.1 <i>Digital Delay - (Echo, Single Delay, Multi-tap Delays and ADT).....</i>	<i>37</i>
3.3.2 <i>Delay Modulation Effects.....</i>	<i>44</i>
3.3.2.1 <i>Flanger Effect</i>	<i>45</i>
3.3.2.2 <i>Chorus Effect</i>	<i>48</i>
3.3.2.3 <i>Vibrato.....</i>	<i>54</i>
3.3.2.4 <i>Pitch Shifter</i>	<i>54</i>
3.3.2.5 <i>Detune Effect.....</i>	<i>55</i>
3.3.3 <i>Digital Reverberation Algorithms for Simulation of Large Acoustic Spaces.....</i>	<i>55</i>
3.4 AMPLITUDE-BASED AUDIO EFFECTS.....	61
3.4.1 <i>Tremolo - Digital Stereo Panning Effect.....</i>	<i>61</i>
3.4.2 <i>Signal Level Measurement.....</i>	<i>62</i>
3.4.3 <i>Dynamics Processing.....</i>	<i>63</i>
3.4.3.1 <i>Compressors and Limiters</i>	<i>63</i>
3.4.3.2 <i>Noise Gate/Downward Expander</i>	<i>66</i>
3.4.3.3 <i>Expanders.....</i>	<i>67</i>
3.5 SOUND SYNTHESIS TECHNIQUES.....	67
3.5.1 <i>Additive Synthesis</i>	<i>67</i>
3.5.2 <i>FM Synthesis</i>	<i>68</i>
3.5.3 <i>Wavetable Synthesis</i>	<i>68</i>
3.5.4 <i>Sample Playback</i>	<i>68</i>
3.5.5 <i>Subtractive Synthesis</i>	<i>69</i>
4. CONCLUSION.....	69

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 <ul style="list-style-type: none"> • Electronic Music Keyboards • Digital Audio Effects Processors (<i>Reverb, Chorus, Flanging, Vibrato</i> <i>Pitch Shifting, Dyn Ran. Compression....</i>) • Vocal "Harmonizers" / Formant-Corrected Pitch Shifters • Graphic and Parametric Equalizers • Digital Mixing Consoles • Digital Recording Studios (DAT) / Multichannel Digital Audio Recorders • Speaker Equalization • Room Equalization 	Algorithms Used <p>Wavetable/FM synthesis, Sample Playback, Physical Modeling Delay-Line Modulation/Interpolation, Digital Filtering (Comb, FIR....)</p> <p>STFFT(Phase Vocoder), additive synthesis, frequency-domain interpolation(Lent's Alg), windowing Digital FIR/IIR filters Filtering, Digital Amplitude Panning, Level Detection, Volume Control Compression techniques: MPEG, ADPCM, AC-3 Filtering Filtering</p>
Consumer Audio Products <p>Karaoke Digital Graphic Equalizers Digital Amplifiers/Speakers Home Theater Systems {Surround-Sound Receivers/Tuners} Digital Versatile Disk (DVD) Players Digital Audio Broadcasting Equip. CD Players and Recorders CD-I Satellite (DBS) Broadcasting Satellite Receiver Systems Digital Camcorders Digital Car Audio Systems (<i>Digital Speakers, Amps, Equalizers</i> <i>Surround-Sound Systems</i>)</p> <p>-----</p> <p>Computer Audio Multimedia Systems</p>	Algorithms Used <p>MPEG, audio effects algorithms Digital Filtering Digital Filtering AC-3, Dolby Prologic, THX DTS, MPEG, Hall/Auditorium Effects AC-3, MPEG... AC-3, MPEG... PCM ADPCM, AC-3, MPEG AC-3, MPEG AC-3,</p> <p>Ex. Circle Surround (RSP Tech.) Digital Filtering...</p> <p>3D Positioning (HRTFs), ADPCM, MPEG, AC-3</p>

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 its 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 others 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 requirements 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 I²S, 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 I²S 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 I²S. 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 (I²S) bus, a serial interface developed for digital audio to enable easy connectivity and ensure successful designs. In short, I²S 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. I²S 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

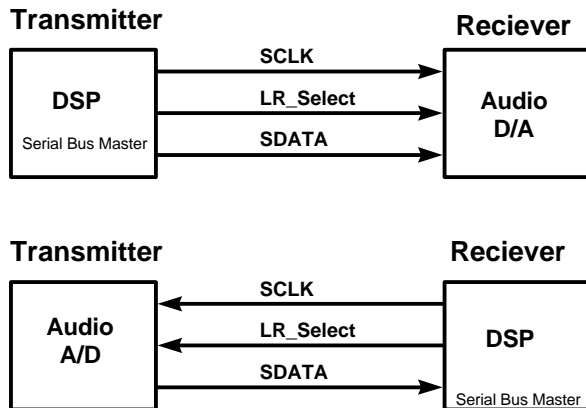
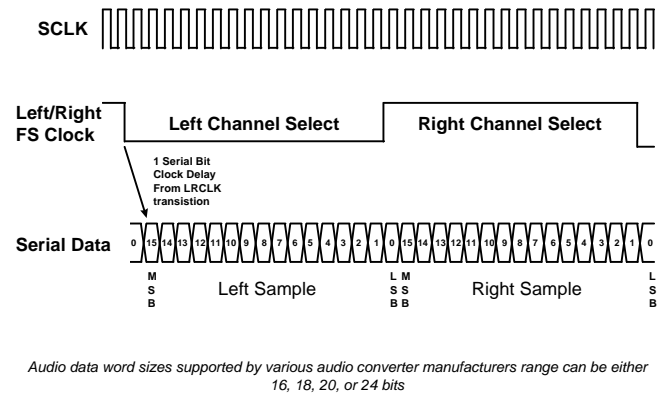


Figure 2. Example I²S Timing Diagram for 16-bit Stereo PCM Audio Data



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

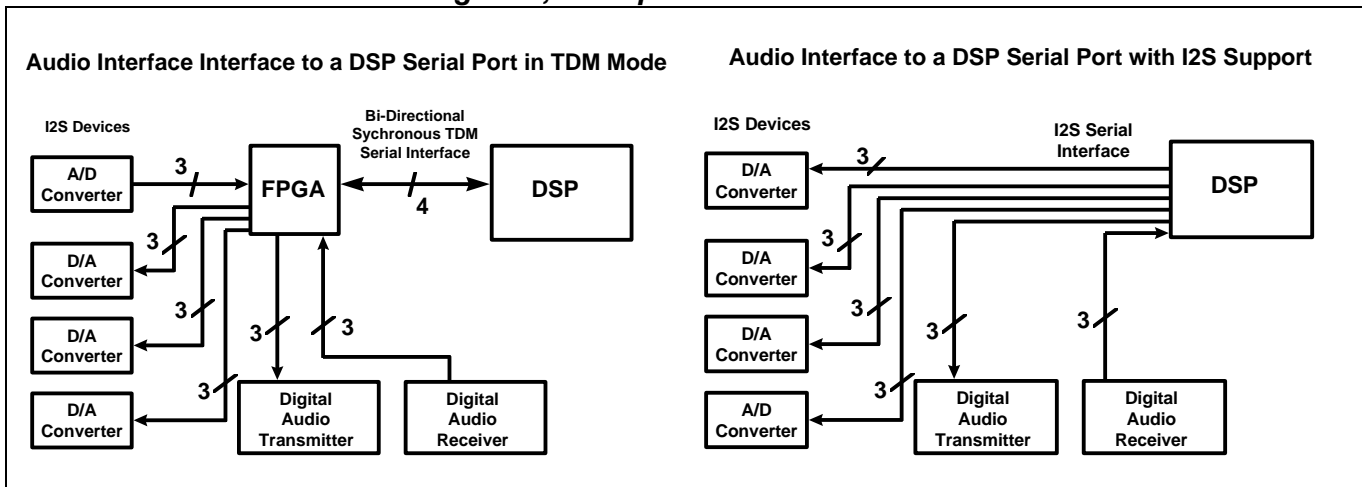
- Audio A/D and D/A converters
- PC Multimedia Audio Controllers
- Digital Audio Transmitters and Receivers that support serial digital audio transmission standards such as AES/EBU, SP/DIF, IEC958, CP-340 and CP-1201.
- Digital Audio Signal Processors
- Dedicated Digital Filter Chips
- Sample Rate Converters

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

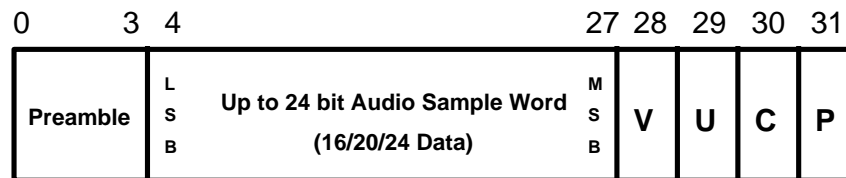


SPD/IF & AES/EBU Digital Audio Transmitters and Receivers

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 (two's 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

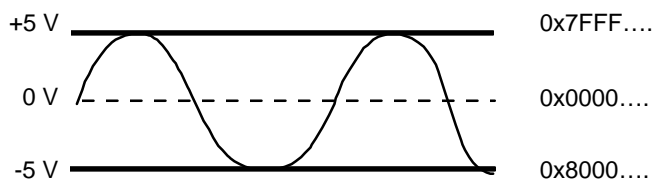
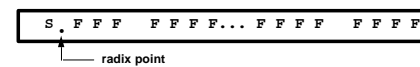


Figure 5

Fractional And Integer Formats

- Fractional format is 1.15 / 1.23 / 1.31 notation



- Integer format is 16.0 / 24.0 / 32.0 notation



In the fractional format, the binary point is assumed to be 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

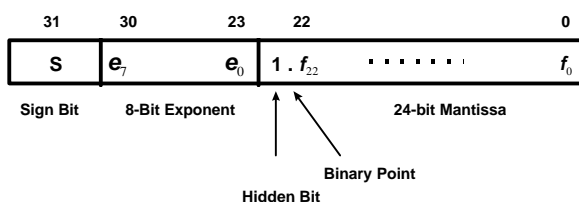
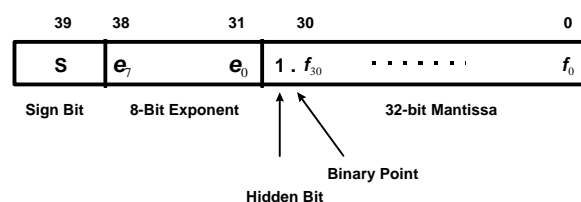


Figure 7.

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:

$$n = (-1)^S \times 2^{e-128} (1.b_1b_2...b_{23})$$

It is important to know that the IEEE standard always refers to the mantissa in signed-magnitude format, and not in two's-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:

$$n=(-1)^s \times 2^{e-128} (1.b_0b_1b_2---b_{30})$$

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.

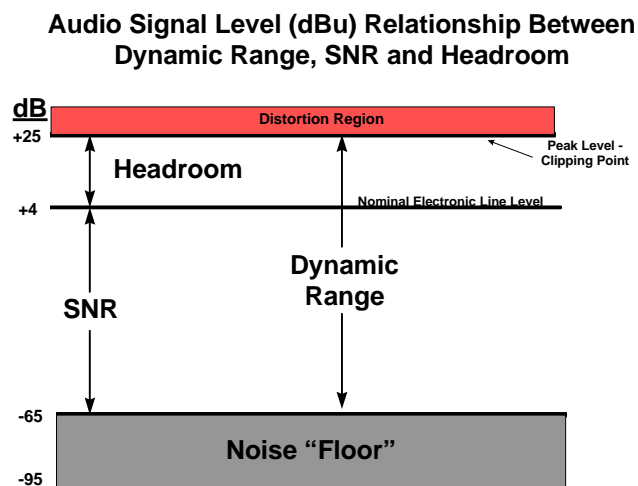
1.6 The Importance Of Dynamic Range In DSP-Audio Processing

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.

Figure 9.

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



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 ratio 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 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

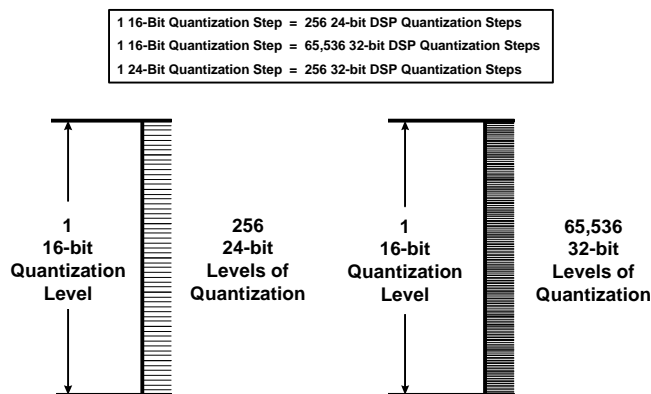
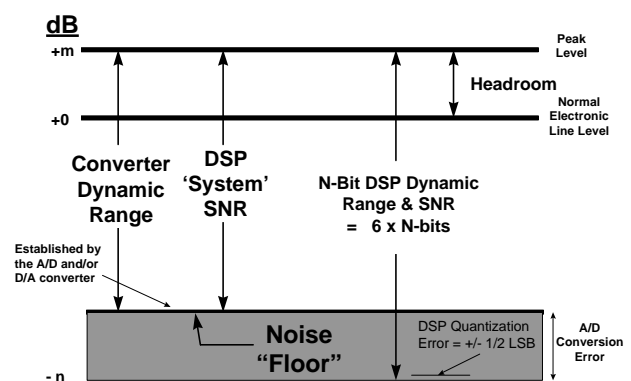


Figure 11.

DSP/Converter SNR and Dynamic Range



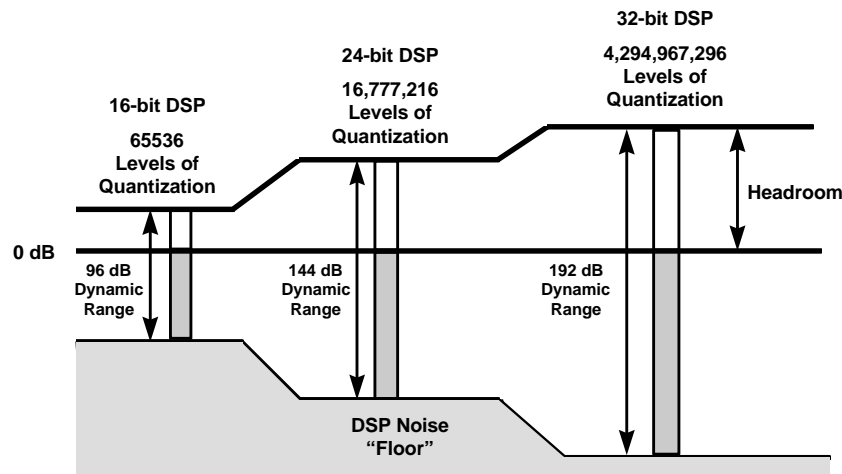
The maximum representable signal amplitude to the maximum quantization error for of an ideal A/D converter or DSP-based digital system is calculated as:

$$\text{SNR}_{A/D(RMS)}(\text{dB}) = 6.02n + 1.76 \text{ dB}$$

$$\text{Dynamic Range}(\text{dB}) = 6.02n + 1.76 \text{ dB} \approx 6n$$

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,	$16 \times (6 \text{ dB per bit}) = 96 \text{ dB}$
24 bit fixed point precision yields 144 dB,	$24 \times (6 \text{ dB per bit}) = 144 \text{ dB}$
32 bit fixed point precision yields 192 dB,	$32 \times (6 \text{ dB per bit}) = 192 \text{ dB}$

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 DSP 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

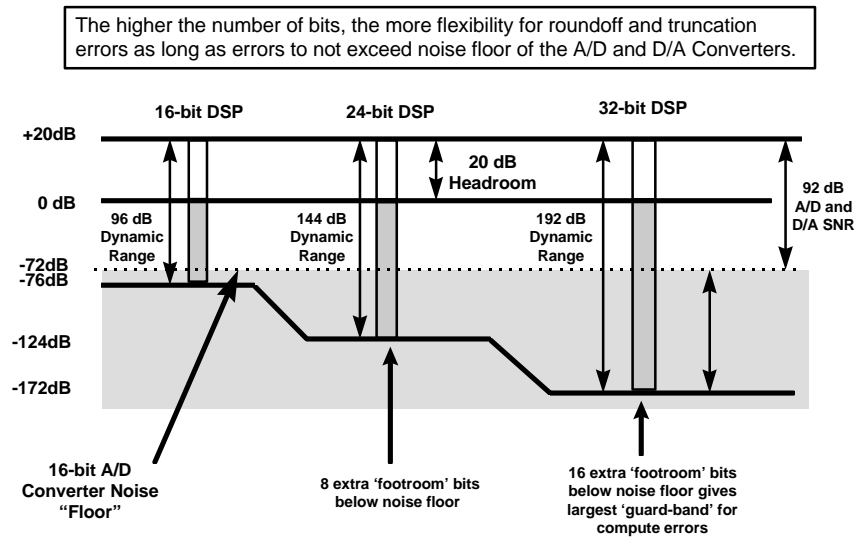
Fielder [38] demonstrates the dynamic range requirements for consumer CD audio requires 16-bit conversion/processing while the minimum requirement for professional audio is 20-bits (based on perceptual tests performed on human auditory capabilities). Dynamic range application requirements for high fidelity 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 ($6 \times 16 = 96\text{dB}$). 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 its 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

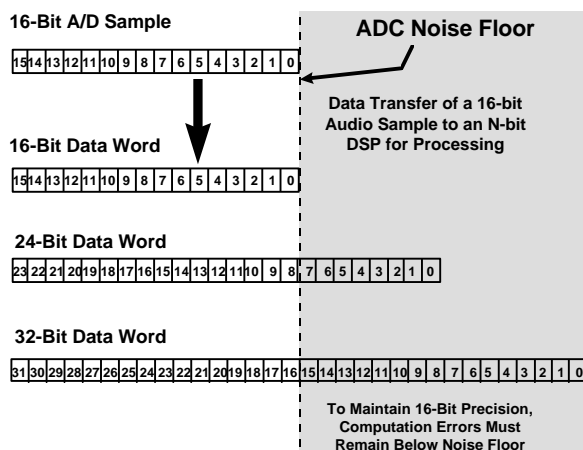
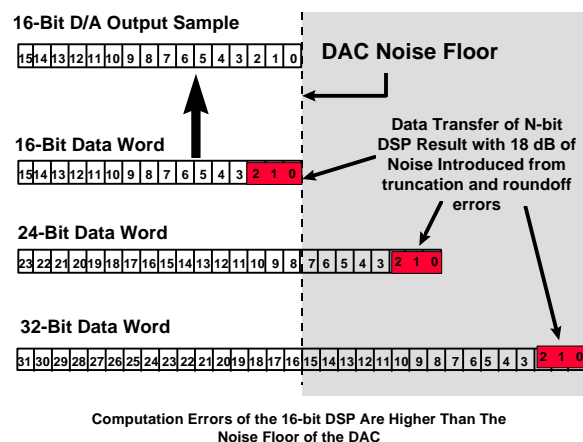


Figure 15.
16-bit D/A Output Samples with Finite Length Effects



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 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 rate. 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.

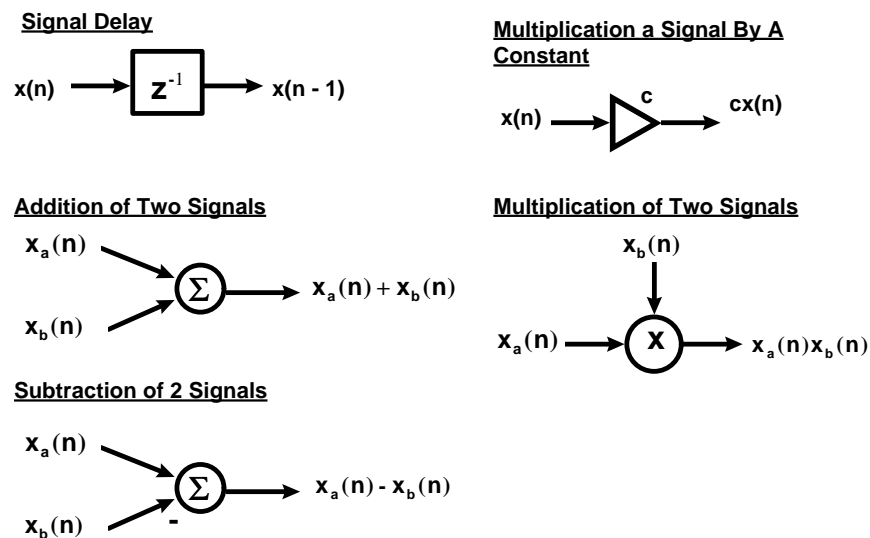
2. USEFUL DSP HARDWARE/SOFTWARE BUILDING BLOCKS FOR AUDIO

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.

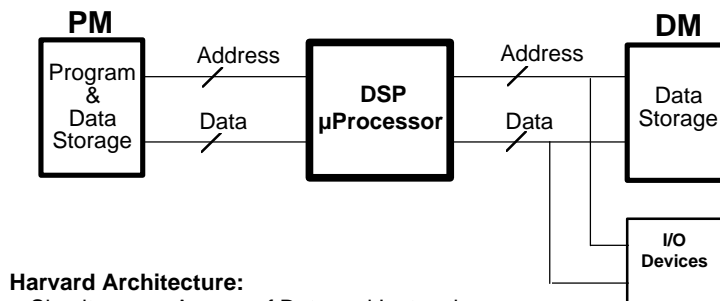
Multiply / Accumulates (MAC)

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

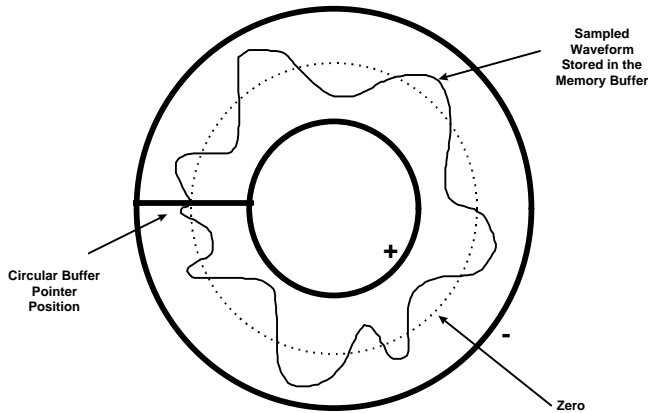
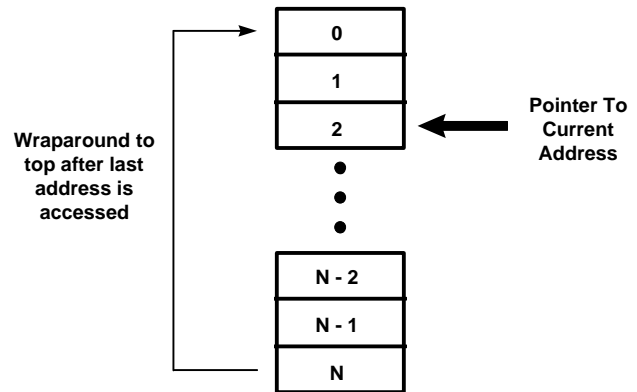


Figure 19.
Circular Buffer of Length N



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 known as *zero overhead*, meaning the branching, loop decrementing, and termination test operations are built into the control unit hardware, saving precious DSP cycles without having to include additional 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

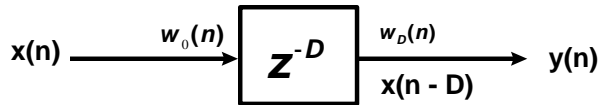
DSP algorithms usually process signals by either block processing or sample processing [2]. For *block processing*, data is transferred to a DSP memory buffer and then processed each time the buffer fills with new data. Examples of such algorithms are fast fourier transforms and fast convolution. The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

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

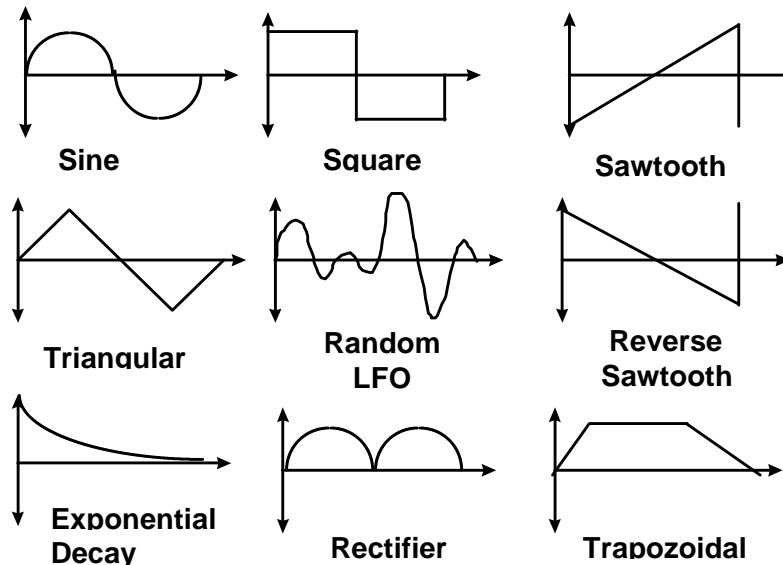
Usually, the sampling rate of the A/D or D/A converter is related to the rate at which the DSP's interrupt service routine is called for data processing. The DSP interrupt rate usually depends on the AD/DA converters since the converters are connected to the DSP's serial ports or are using a hardware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

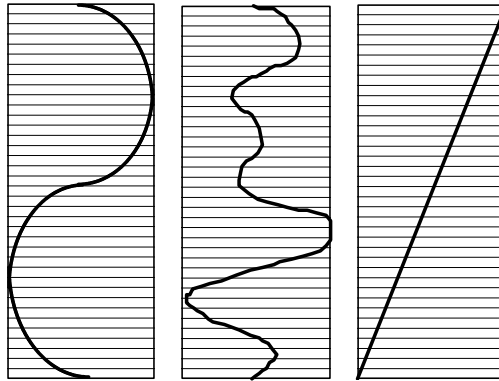
What is the best way to progress through the table? The general recommendation for accessing data from the table would be to declare the wavetable in the DSP program as a circular buffer instead of as a linear buffer (see some examples in Figure 22 below). This will allow the signal to be replayed over 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:** One 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

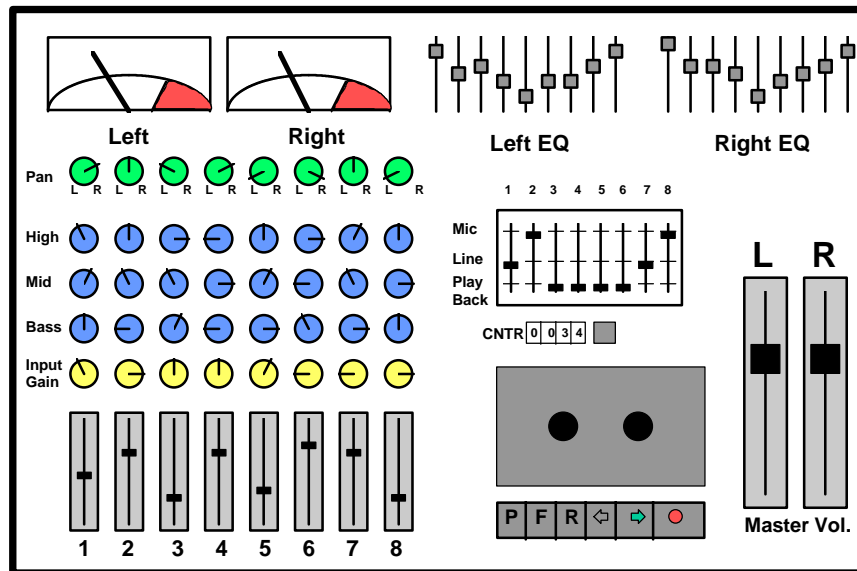


Figure 23 is an example 8 channel mixing console / recorder. Some of the features that are commonly found in mixers and multi-track recorders can be implemented with DSP instructions to perform functions that are often found in mixer equipment as the 8 track recorder shown above:

- 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 as 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:

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

The general mixing equation is:

$$y(n) = \frac{1}{N}[x_1(n) + x_2(n) + \dots + x_N(n)]$$

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:

$$y(n) = c_1x_1(n) + c_2x_2(n) + \dots + c_Nx_N(n)$$

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;

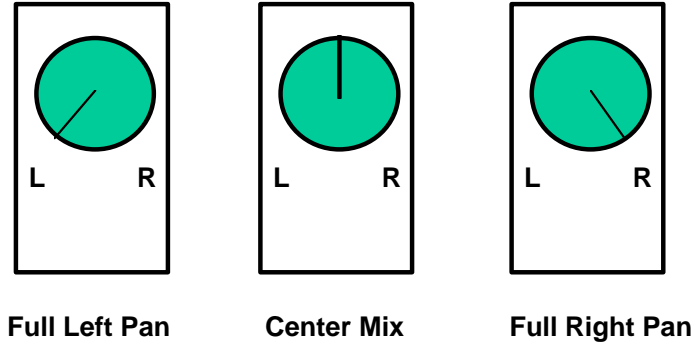
{----- write output samples to stereo D/A converter -----}
r0 = mrf;
dm(left_output) = r0;          {left output sample}
dm(right_output) = r0;         {right output sample}
```

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.

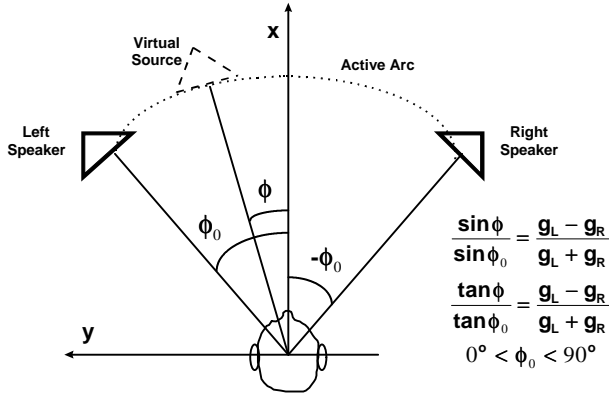
Normally, the stereo signal will contain an exact duplicate of the sampled input signal, although it can be split up to represent two different mono sources. Also, the DSP can also take a mono source and create signals to be sent out to a stereo D/A converter. Typical audio mixing consoles and multichannel 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



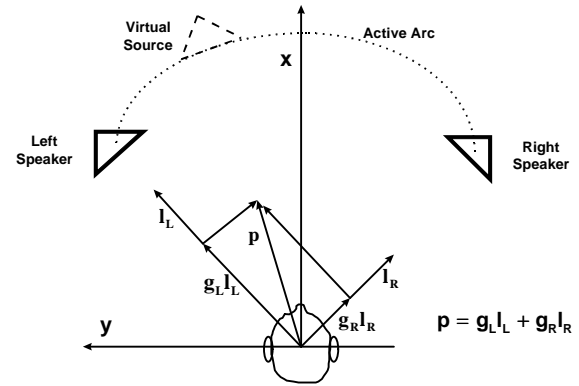
Many 4/8/12 track analog and digital studios contain a knob to pan an input source entirely to the left or right channel, or played back through both channels at an equal mix (with the pan control centered in the middle). To give the listener a sense of location within the output stereo field, the DSP 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]



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

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



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])

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

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:

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

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:

$$p = g_L l_L + g_R l_R$$

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

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

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.

$$\begin{aligned} G_L &= 1 \\ G_R &= 0 \end{aligned}$$

- 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.

$$\begin{aligned} G_R &= 1 \\ G_L &= 0 \end{aligned}$$

- 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:

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

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_{R-norm}
45°	45°	1	0	1	0
45°	30°	1.366	0.366	1	0.2679
45°	20°	1.8737	0.8737	1	0.4663
45°	10°	3.3356	2.3356	1	0.7002
45°	0°	1	1	1	1
45°	-10°	-2.3356	-3.3356	0.7002	1
45°	-20°	-0.8737	-1.8737	0.4663	1
45°	-30°	-0.366	-1.366	0.2679	1
45°	-45°	0	1	0	1

After normalization 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 used 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 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.

30° 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}

```

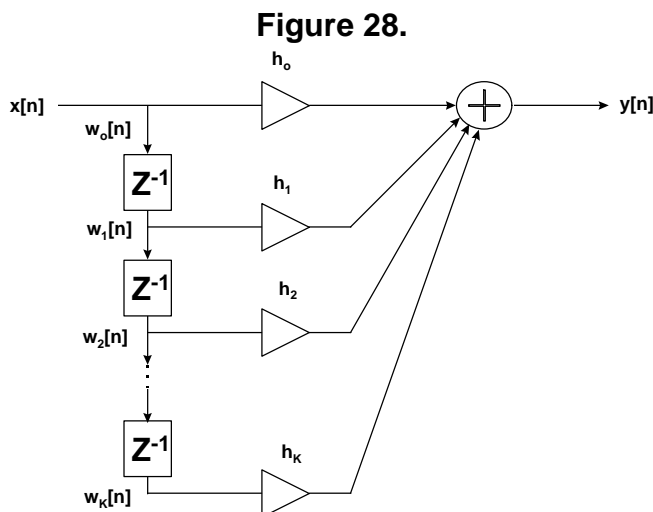
3.2 Filtering Techniques and Applications

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.



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

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 */

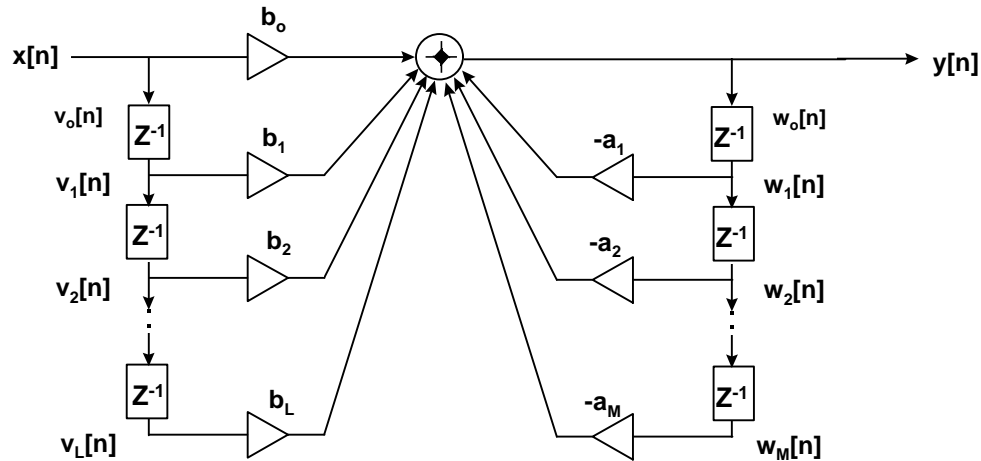
    MRF = MRF + R3*R4 (SSFR);      /* last sample to be computed */
    MRF = SAT MRF;

```

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.



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

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 COEFFICIENT 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 coefficients

# 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)*/

```

```

lcntnr=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 = mrf;
    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 = mrlf;
    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 ω_0 . 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 of r and R to the unit circle.

Figure 30.

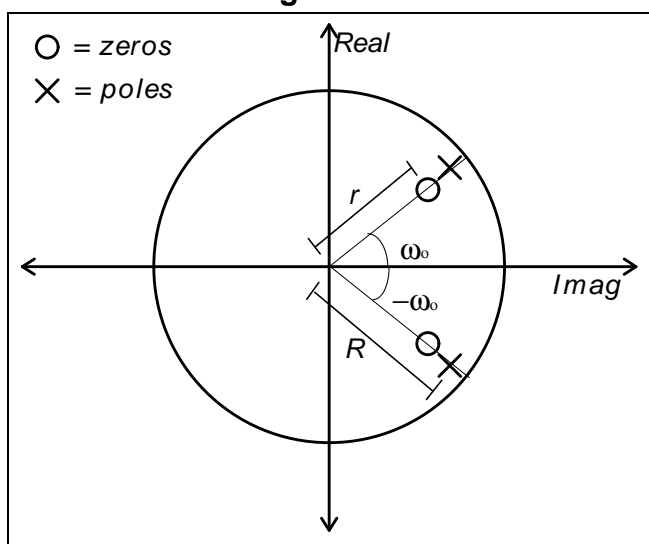


Figure 31.

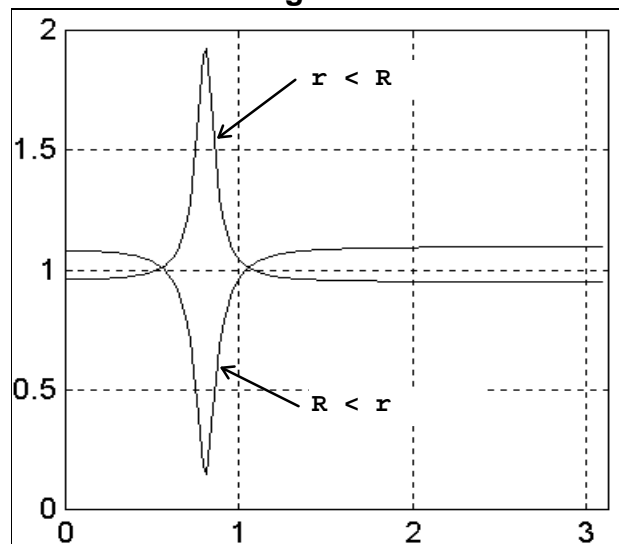


Figure 1 : Pole-Zero Plot of parametric Filter

Figure 2 : Frequency response of parametric filter

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

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

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 ω_o , r and R :

$$\begin{aligned} a_0 &= 1 & a_1 &= -2R \cos(\omega_o) & a_2 &= R^2 \\ b_0 &= 1 & b_1 &= -2r \cos(\omega_o) & b_2 &= r^2 \end{aligned}$$

These calculations are implemented in code example YYYYYY.

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 =  $\omega_o p$      $\omega_o$  = frequency
f1 = R
f3 = r
i9 = pointer to where coefficients are to be stored

outputs:
a1, a2, b1, b2 coefficients
```

```

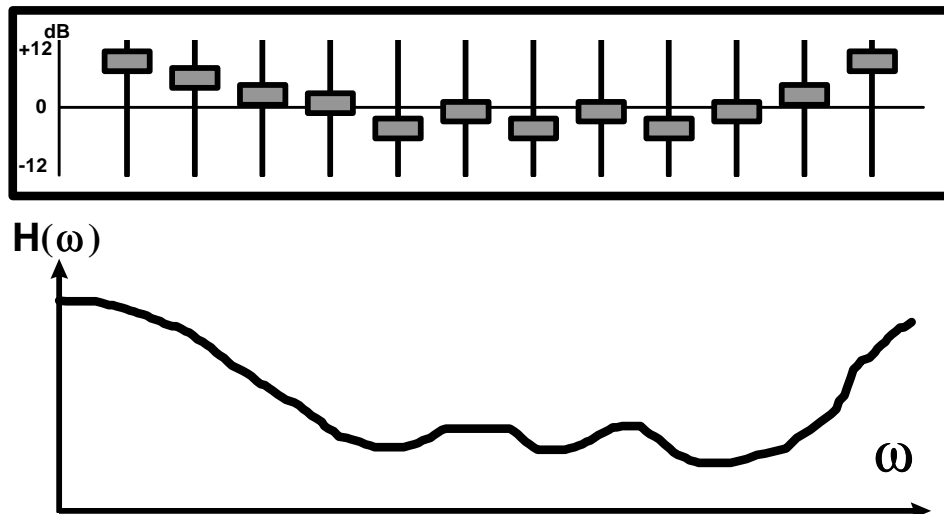
*/
ParametricCalc:
    call cosine (db);          // returns f0 as the cos(f0)
    nop;
    f4 = -2.0;
    f0 = f0 * f4;              /* -2*cos(V_o) */
    f4 = f0 * f3;              /* f4=-2r*cos(V_o) */
    pm(2,i9)=f4;               /* store b1 */
    f4 = f0 * f1;              /* f4=-2R*cos(V_o) */
    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.

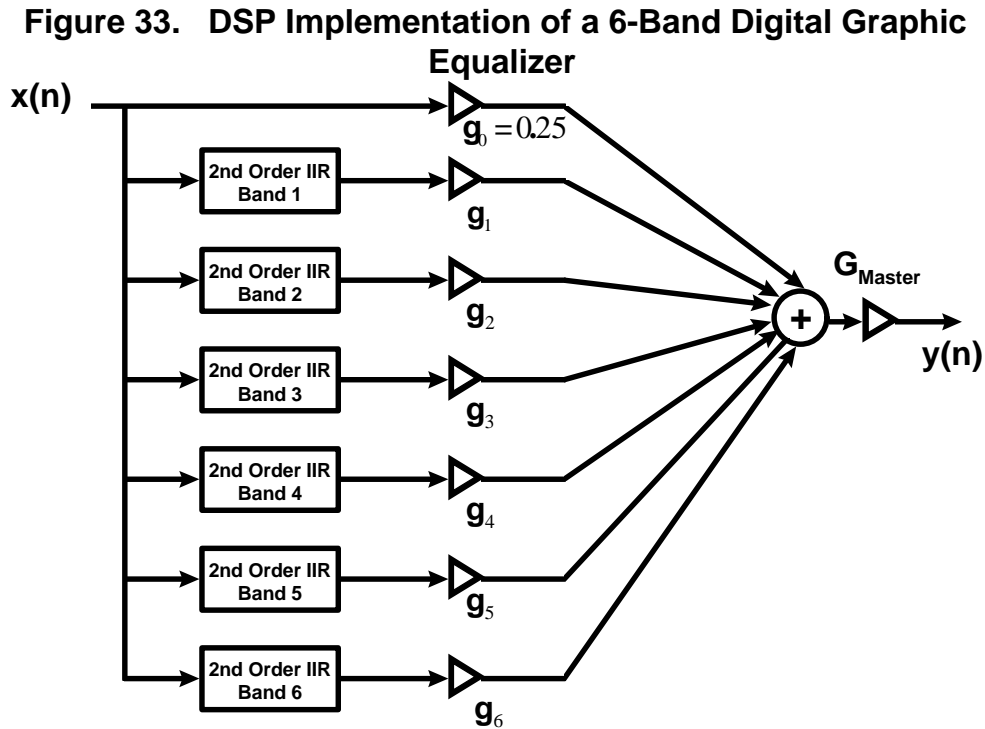
Figure 32. Typical 12 Band Analog Graphic Equalizer



Analog equalizers typically use 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. Resistors and capacitors are replaced by discrete-time filter coefficients, which are stored in a memory and can be easily modified.

Figure 33 shows an 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:

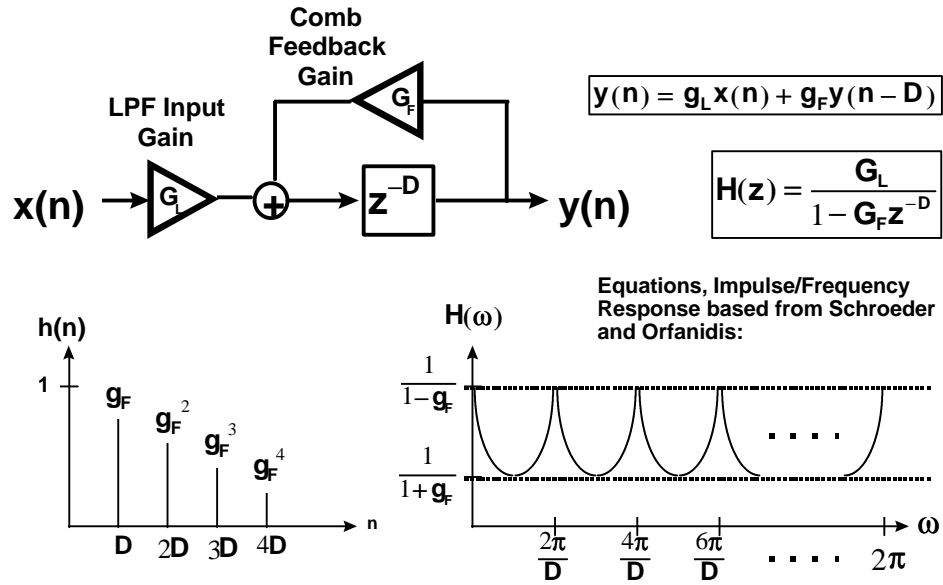
$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 - a_1 z^{-1} - a_2 z^{-2}} \quad y(n) = b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) + a_1 y(n-1) + a_2 y(n-2)$$



3.2.5 Comb Filters

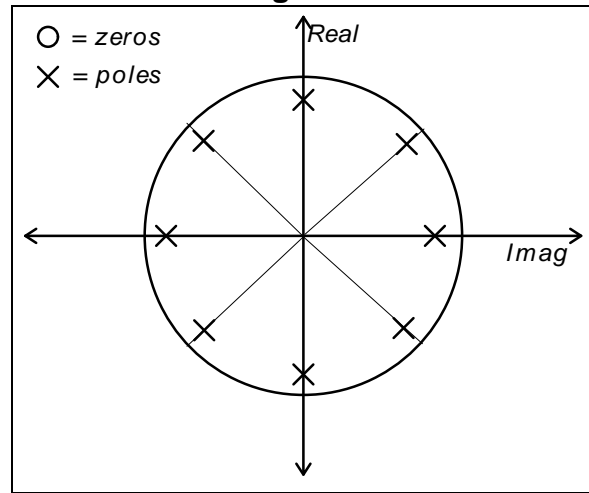
Comb filters are used for noise reduction of periodic signals, signal enhancement, averaging signals, and are inherent in digital audio effects such as delays, chorus and flange effects. Comb filters work by adding a signal with a delayed and scaled version of the same signal. This causes some frequencies 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:

$$\mathbf{H}_{\text{IIRcomb}}(\mathbf{z}) = \mathbf{b} \frac{1 + \mathbf{z}^{-D}}{1 - \mathbf{a} \mathbf{z}^{-D}}, \quad \mathbf{H}_{\text{FIRcomb}}(\mathbf{z}) = \frac{1}{N} (1 + \mathbf{z}^{-D} + \mathbf{z}^{-2D} + \dots + \mathbf{z}^{-(N-1)D}) = \frac{1}{N} \frac{1 - \mathbf{z}^{-ND}}{1 - \mathbf{z}^{-D}}$$

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

$$\mathbf{H}_{\text{FIR_MA}}(\mathbf{z}) = \frac{1}{N} (1 + \mathbf{z}^{-1} + \mathbf{z}^{-2} + \dots + \mathbf{z}^{-(N-1)}) = \frac{1}{N} \frac{1 - \mathbf{z}^{-N}}{1 - \mathbf{z}^{-1}}$$
$$\mathbf{H}_{\text{COMB}}(\mathbf{z}) = \frac{1}{1 - \mathbf{a}\mathbf{z}^{-D}}$$

21065L IIR-Comb Filter Example

39

```

        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; mrlf = 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 = mrlf;

    /* put y result into tap-0 of delay line */ ;
    r12 = mrlf; 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.

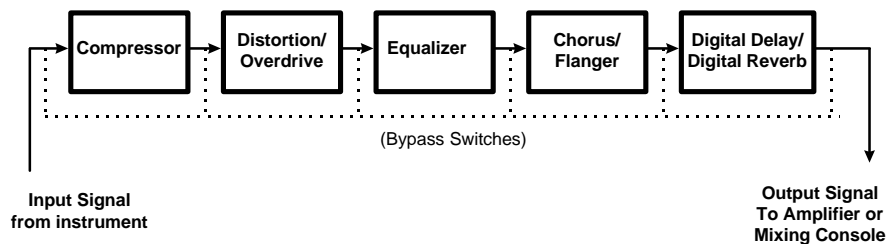
In addition to preventing fixed point overflow when doing simple addition of signals, scaling of input samples when performing fixed point digital filtering, or when going between consecutive filter stages of a higher order filter implementation.

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).

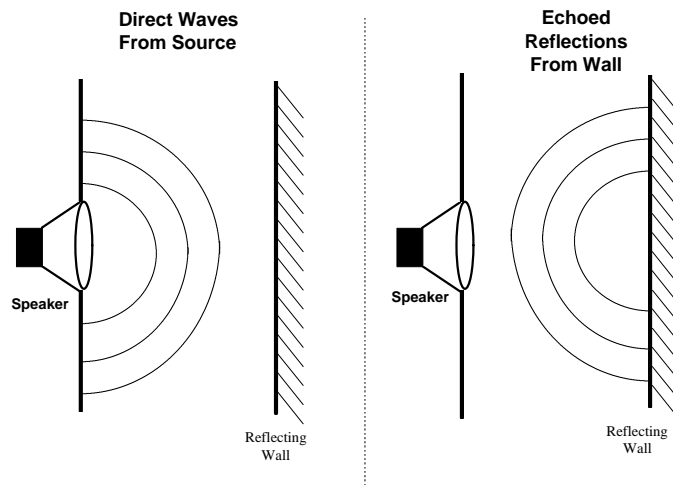
Figure 36.
Typical Signal Chain for Audio Multi-Effects Processors



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

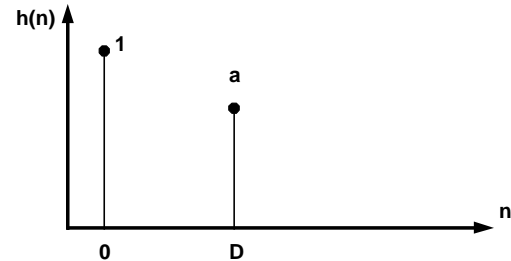
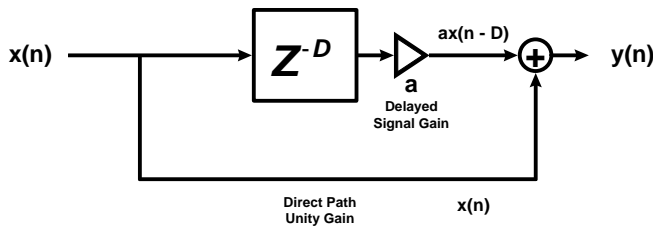


Figure 39. Delay (or Echo) with a Single Reflection

Single Reflection Delay

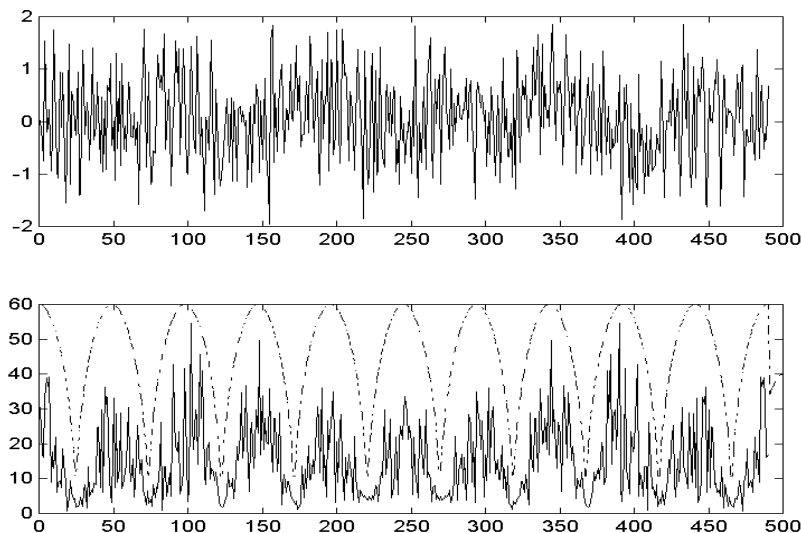
To create a *single reflection* (Figures 38 and 39) of an input signal, the implementation shown above is represented in the following difference equation [2]:

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

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 than 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 = mrf;
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

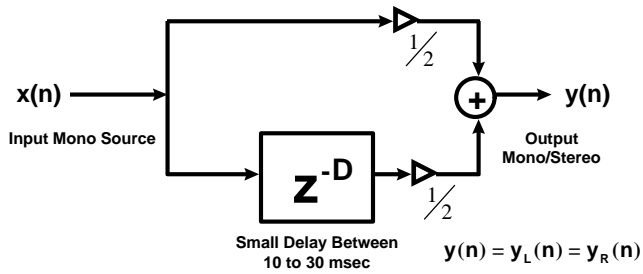
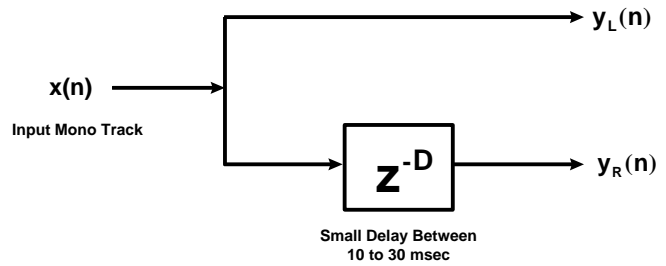


Figure 42.
Automatic Double Tracking / 'Stereo Doubling'



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 */

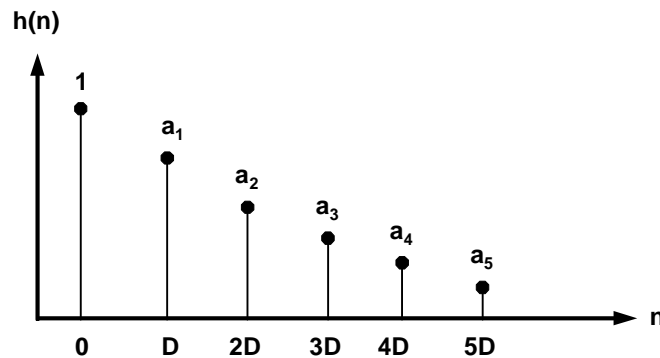
rts;                                  /* Return from Subroutine */

```

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.

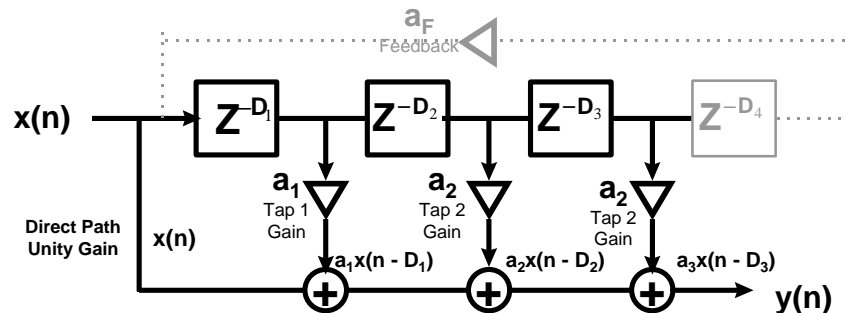
Figure 43.
Typical Impulse Response of Multiple Delay Effect



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

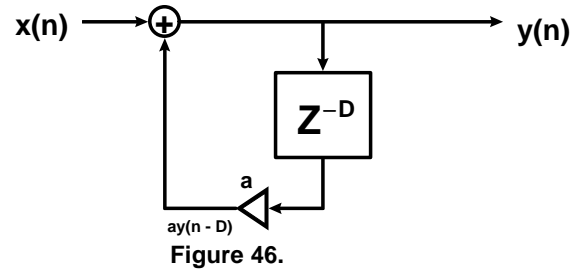
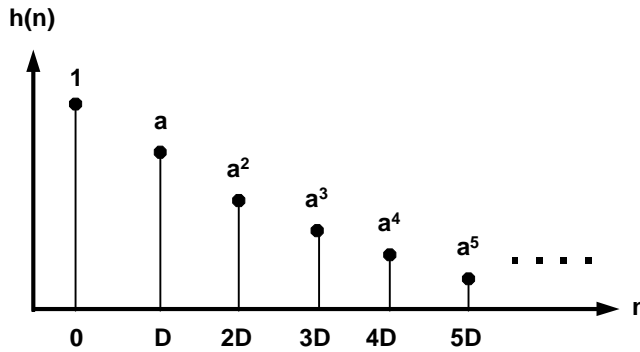


$$y(n) = x(n) + a_1x(n - D1) + a_2x(n - D2) + a_3x(n - D3)$$

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



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):

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

The transfer function is then:

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

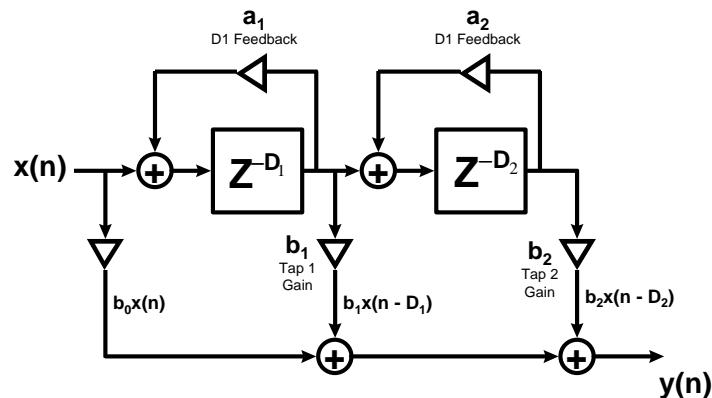
Represented as a geometric series, the transfer function becomes a recursive IIR comb filter:

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

Since an infinite number of delays are created as the response becomes an 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 produce 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:

$$H(z) = b_0 + b_1 \left[\frac{z^{-D_1}}{1 - a_1 z^{-D_1}} \right] + b_2 \left[\frac{z^{-D_1}}{1 - a_1 z^{-D_1}} \right] \left[\frac{z^{-D_2}}{1 - a_2 z^{-D_2}} \right]$$

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 = mrf;
dm(left_output) = r10;    {left output sample}
r10 = mrf;
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;
```

```

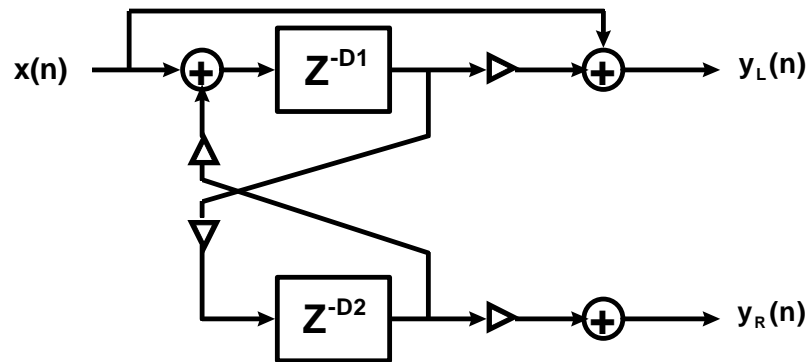
/* put input sample into tap-0 of delay line #1, postmodify address after storage */
r12 = mr1f; dm(i2, -1) = r12;
rts;

```

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 type effects 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(z) + 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;
    mrlf = r3;                    {mrlf = sLL}
    r2 = cL;
    mrf = mrf + r0 * r2 (ssfr);    {mrf = yL = cL * xL + sLL = left output}
    mrf = sat mrf;

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

```

```

mrf = 0;
mrlf = r4;
r2 = cR;
mrf = mrf + r1 * r2 (ssfr);
mrf = sat mrf;

r10 = mrlf;
dm(Right_Channel) = r10;

r2 = bL;
mrf = r0 * r2 (ssf);
r2 = aL;
mrf = mrf + r3 * r2 (ssf);
r2 = dR;
mrf = mrf + r4 * r2 (ssfr);
mrf = sat mrf;

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

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

r2 = bR;
mrf = r1 * r2 (ssf);
r2 = aR;
mrf = mrf + r4 * r2 (ssf);
r2 = dL;
mrf = mrf + r3 * r2 (ssfr);
mrf = sat mrf;

/* put value from MAC into delay line into tap-0 of right delay-line */
m3 = 0;
r12 = mrlf; 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

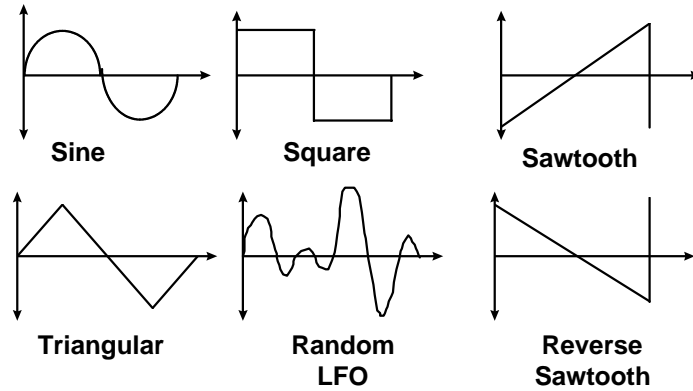


Figure 50. Delay-Line Modulation General Structure (J. Dattorro)

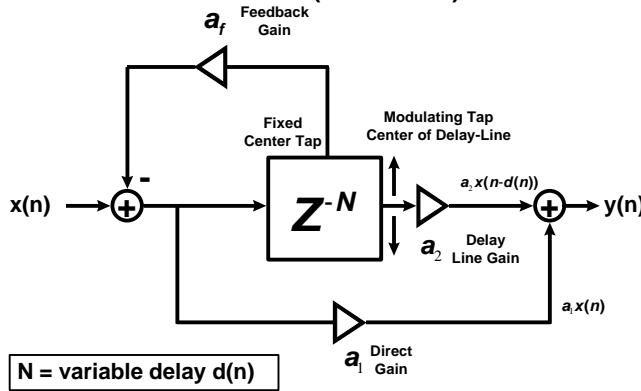
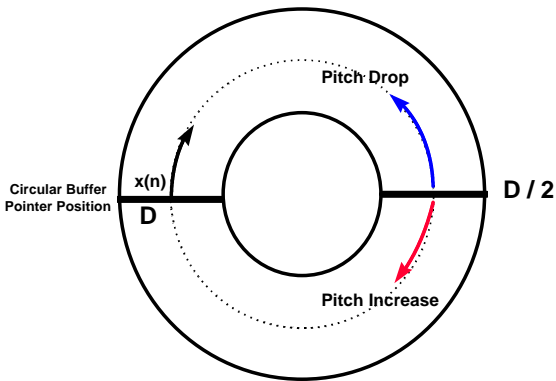


Figure 51. Visualizing The Result Of A Rotating Center Tap



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-reel tape decks and monitored from both playback heads of the 2 tape decks 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 tape 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

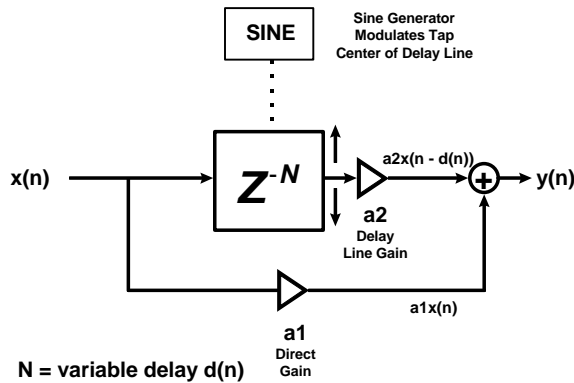
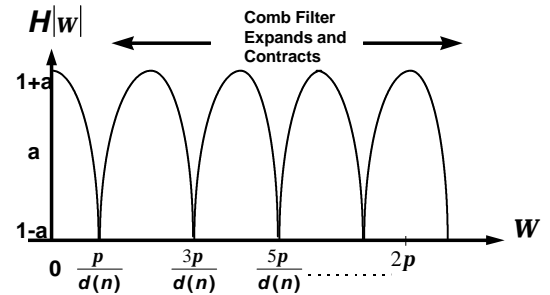


Figure 53.
Frequency Response of the Flanger



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 $\frac{1}{2}$ 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]:

$$d(n) = \frac{D}{2} [1 - \cos(2\pi n f_{\text{cycle}})]$$

Figure 54.
Sine Wavetable Circular Buffer Storage

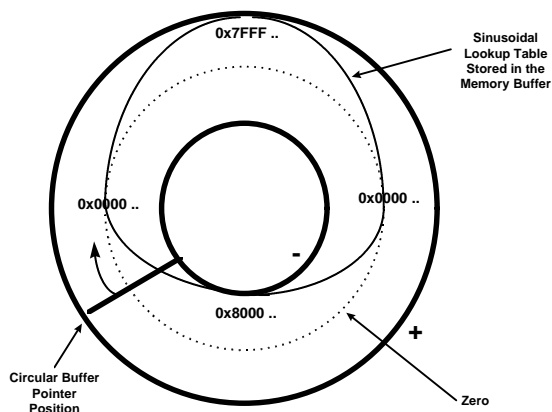


Figure 55.
Example 4 K-word Sinusoidal Wavetable

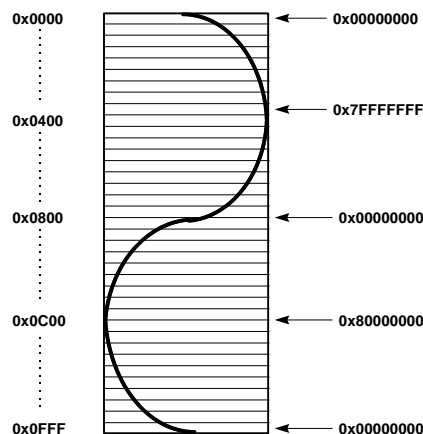
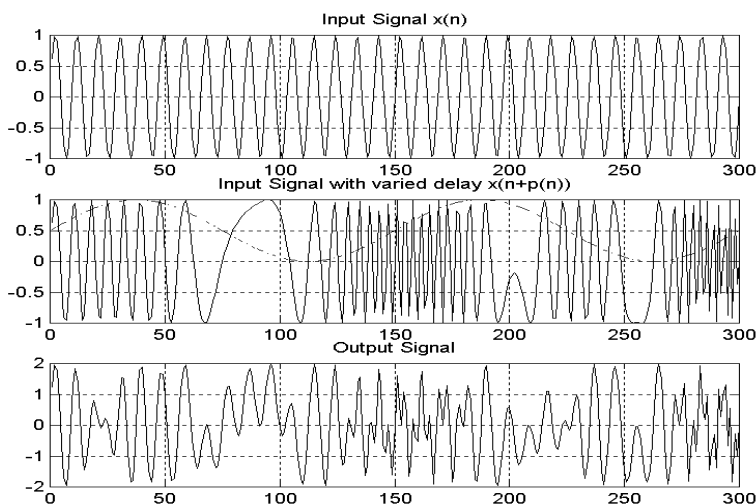


Figure 66.
"Flanger" Result of Adding a Variable Delayed Signal To It's Original



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. Society, Vol. 45, No. 10,
   October 1997

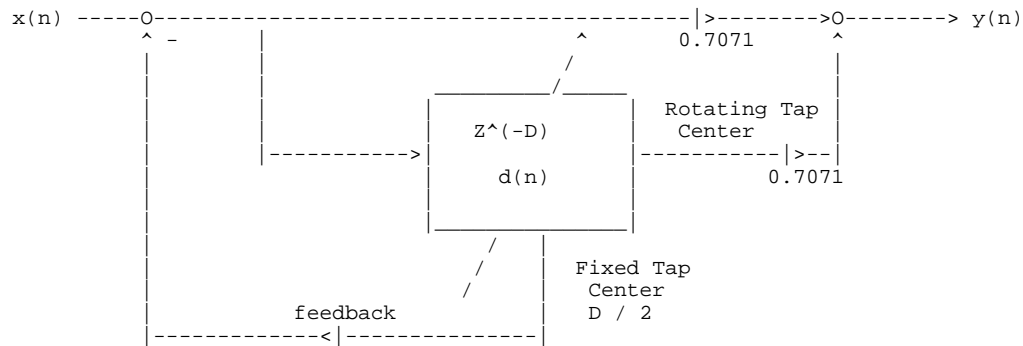
```

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 "def210651.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 */
  
```