AUDIO DIGITAL SIGNAL PROCESSING IN REAL TIME

by

Paul L. Browning

A problem report submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

West Virginia University

1997

Approved by _____

Chairperson of Supervisory Committee

_____

_____

_____

Date _____

Abstract

AUDIO DIGITAL SIGNAL
PROCESSING IN REAL TIME

by Paul L. Browning

Most modern desktop computers are equipped with audio hardware. This hardware allows audio to be recorded as digital information for storage and later playback. This digital information can be manipulated to change how the audio sounds when played back. Several digital audio "effects" have become commonplace because of the flexibility and fidelity of digital editing. Sufficient processing power and hardware facilities would allow the manipulation of audio information in real time on a common desktop computer. Successful real-time processing requires a combination of efficient hardware, process scheduling, and efficient algorithms. This report is the result of an investigation of the specific hardware and software requirements for performing a common set of digital audio processing "effects" in real time under the Windows 95/NT platform.

TABLE OF CONTENTS

LIST OF FIGURES

Number *Page*

ACKNOWLEDGMENTS

The author wishes to thank the Internet community for valuable suggestions and comments on this project.

LEGAL NOTICES

GLOSSARY

Acoustics..... The study of the physical properties and behavior of audio waves.

A-D Converter ..... Analog-to-Digital Converter - an electrical device that converts an analog voltage to a digital value.

Algorithm ..... A well specified technique for performing a task.

Amplitude..... The magnitude of a wave. The amount of air displaced by each cycle of a sound wave.

All-Pass Filter ..... A unit that has a flat (constant) frequency response curve. Note that an all-pass filter does not necessarily preserve a sound, but has no effect on the intensity of the sound's frequency component intensities.

Analog..... In a continuous varying form, as opposed to digital. Analog signals and values are not restricted to any minimum unit of measurement.

Audio File..... A computer file that contains digital audio data.

Band-Pass Filter ..... A filter that allows only a certain range (band) of frequencies to pass through.

Base Delay ..... The fixed part of a delay that is summed with a modulated delay.

Buffer..... A designated area of the computer's memory that stores a chunk of data. Buffers are often used to simplify the procedure of exchanging audio data between a computer program an the computer's operating system.

Circular Buffer ..... An array of storage locations that has a "end" which "wraps around" to the beginning when the end is reached. Storing data items at the "top," before the "top" is advanced allows a fixed number of previous data items to always be available.

Cochlea ..... An organ in the ear that converts eardrum vibrations into nerve impulses.

Comb Filter ..... A unit that has a frequency response curve with evenly spaced symmetric peaks.

Complex Exponential .... A number with an exponent that is a complex number.

D-A Converter ..... Digital-to-Analog Converter - an electrical device that converts a digital value to an analog voltage.

Device Handle ..... A variable that is used to reference an audio device.

Dialog Box ..... A window that appears on the computer's display, enabling the user to view and adjust the data and settings contained in the window.

Digital ..... In a numeric or binary format, as to allow processing by a digital computer.

Digital Audio Data..... A series of numeric values that represent audio information.

Digital Audio Effect..... The manipulation of digital audio data, often to imitate naturally occurring acoustic phenomenon such as room echo.

Discrete Function..... A function that is defined only at specific points, rather than continuously.

D.M.A. ..... Direct Memory Access - DMA channels allow devices to read and write to a computer's memory without the intervention of the CPU. This provides much faster data transfer than if the CPU were required. Most audio hardware uses D.M.A. facilities to quickly move the large amounts of data required for digital audio.

D.S.P. ..... Digital Audio Signal Processing - the analysis and manipulation of digital audio data.

E.A.R. ..... "Edit Audio in Real time" - the software written to accompany this project.

Effect..... See Digital Audio Effect

Filter ..... A unit that changes the distribution of frequencies in an audio signal, according to a linear scaling of each frequency. Filters are often used to reduce or magnify certain frequency ranges.

F.F.T. ..... Fast Fourier Transform - an algorithm for converting a series of samples into a series of values that represents the frequency distribution of the waveform represented by the samples.

Frequency..... The number of complete cycles that occur per unit of time. A waveform that completes ten cycles per second has a frequency of 10 Hertz (Hz).

Frequency Analysis ..... The process of determining the distribution of frequency intensities in a specific audio signal.

Frequency Response..... The distribution of the intensity of the output frequencies of a unit that modifies an input signal. A frequency response curve, usually a smooth curve, shows which frequencies are blocked or reduced by the unit.

Full-Duplex ..... Having the ability to record and play audio simultaneously.

Gain ..... Multiplying an audio signal by a constant value. Identical to a volume control.

I.F.F.T..... Inverse Fast Fourier Transform - an algorithm for reversing the transformation performed by the F.F.T.

M.D.I..... Multiple Document Interface - a framework of Microsoft Windows functions and classes that provide support for applications and their "documents."

Modulated Delay ..... The part of a delay that varies. The amount and fashion that it varies, depends on the "modulation." Note that a modulated delay may be positive or negative.

Natural Sound..... Sound that occurs in the real world, rather than information that represents a sound. Waves of vibration in a medium of air.

Noise..... Unwanted or random sound. Usually at low levels.

P.C.M..... Pulse Code Modulation - the representation of audio as a series of numerical values, representing the position of an audio wave at varying points in time.

Pitch..... see Frequency

Playback ..... The process of converting a series of samples to an audio signal.

Pseudo-code..... An English-like description of an algorithm for ease of understanding.

Real-Time..... Able to keep up with a well-defined load of work without exceeding well-defined deadline or performance criteria for an indefinite length of time.

Recording ..... The process of converting an audio signal to a series of samples.

Sample ..... A single unit of PCM digital audio data, corresponding to the relative air displacement at a specific instant in time.

Sampling Rate..... The number of samples per second taken when recording digital audio data.

Triangle ..... A periodic waveform that linearly increases and decreases to a specific amplitude, forming a series of triangles of plotted.

Volume..... The loudness of a sound. The amplitude of a sound wave.

Zero Crossing..... The point at which a wave cross the point of equilibrium. With PCM data, samples at this point usually have a value of zero.

Chapter 1 - Project Summary and Overview

Summary of Project Objectives

I have always been interested in audio sound effects. Commonplace audio hardware allows the typical computer to produce many audio effects by digitally processing audio data. With the recent availability of audio hardware that is capable of playing and recording simultaneously, it should be possible to perform digital audio effects on a live audio stream. A particular audio effect should be possible for a live audio stream if:

A. An algorithm for processing an infinite length of audio data exists

B. The computer has sufficient computing performance.

This project resulted in the completion of the following tasks:

- Investigate the following “effects”: echo, reverb, chorus, flange, distortion, spectrum analysis, spectrum shifting and scaling, and filtering.
- Modify existing algorithms or create new algorithms that will perform the digital effects on a stream of audio data of infinite length.
- Implement the algorithms by creating software for the Windows 95/NT platform.
- Determine the criteria for the software to work in “real time”. "Real-time" will be defined as “an output audio stream keeps up with an input audio stream of infinite length.”

Summary of Project Results

The software that I created was named "E.A.R." (Edit Audio in Real time). I was able to develop algorithms for all the digital effects listed above. However, some effects require more computation than others. E.A.R. allows the effects to be performed in real-time with sufficiently fast computers. All effects can also be performed on prerecorded audio files with computers of any speed. Following introductions to the "fundamentals of digital audio" and "working with PC-based audio", each chapter summarizes the mathematical and theoretical basis of each effect. These summaries are followed by pseudo-code of the algorithms used in E.A.R., screen-shots, and a summary of any interesting issues encountered while implementing the effect. The series of effects is followed by a description of the inner-workings of E.A.R. and programming issues addressed during its creation. This report is concluded with a summary of work done feedback from the digital audio community, and possible future work.

I used the following hardware and software for the bulk of my work:

- Microsoft Visual C++ 4.0
- Dell Optiplex Pentium-Pro 200MHz Computer
- Sound Blaster 16 Sound Card
- Microsoft Windows NT 4.0
- Microsoft Windows 95

Chapter 2 - Fundamentals of Digital Audio

Physical Properties of Sound

Digital audio is the representation of natural sound (waves of vibration in a medium of air) as a set of digital information (a series of numbers). Sound is created when the air is disturbed, usually by a vibrating object. The vibrating object causes ripples of varying air pressure. Very little air actually moves anywhere, as the pressure change is propagated by the collision of air molecules, similar to the way ripples spread across the surface of a pond without causing water currents. (Pohlmann, 1993, 1-3) These "waves" of varying pressure cause the eardrum to move back and forth. The movement is carried from the eardrum to an organ called the *cochlea* by a series of tiny bones. The cochlea contains a series of over 10,000 different-sized hairs, which convert these vibrations to nerve impulses. The impulses are carried to the brain and decoded. (Carley and others, 1995)

The ear processes two characteristics of sound: *volume* and *pitch*. Natural sounds are composed of multiple pitches, each at a potentially different volume. A simple periodic wave, as shown in Figure 2-1, has only one pitch and volume. The volume or *amplitude* of the sound wave corresponds to the amount of air displaced by each oscillation or wave cycle. The pitch or *frequency* of the wave corresponds to the number of wave cycles per second. The different-sized hairs in the ear's cochlea respond to the specific frequencies present in the sound wave. The amount of hair vibration and the intensity of the resulting nerve impulse are proportional to the amplitude of the particular frequency. Figure 2-2 illustrates a simple example of two frequencies present in a combined single sound or waveform.

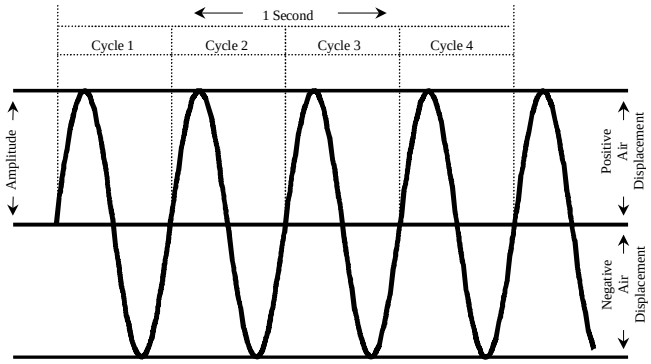

Figure 2-1: Amplitude and Frequency of a Simple Cyclic Waveform

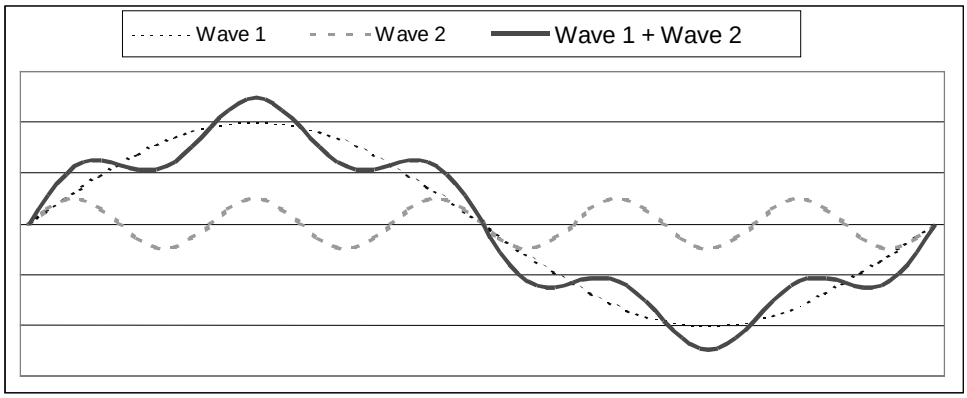

Figure 2-2: Combining Two Waves of Different Frequencies

Digital audio signal processing is the analysis and manipulation of audio waveform information, which can become a complex endeavor. Like many computer representations of real-world phenomena, the modeling of sound waves can be performed at any desired level of complexity. The complexity of natural acoustics is mostly due to the complexity of the natural world. As Figure 2-3 illustrates, even a small sample of a human voice waveform is very complex and is not completely periodic. Additionally, the physical properties of objects in a room and the air itself determine what frequencies are reflected or absorbed. Most people can listen to a sound from a familiar source while blindfolded and determine the size of the room. A well-trained ear can even determine the types of materials that comprise the walls.

Changing properties, such as the humidity and temperature, can affect the acoustics of a room. Additionally, the shapes and angles of the walls of a room determine the direction that the audio waves "bounce." Different waves that have bounced off of surfaces can collide, causing either an addition or cancellation effect. (Pohlmann, 1993, 11-12) This effect can easily be seen if a small object is dropped in a container of water. Not all areas of the water will have the same ripple pattern, due to the waves that are reflected from the container's walls.

The size of an object in a room can block or bend different frequencies. For example: holding a notebook between you and your home stereo will block out the highest frequencies, but you will still be able to hear the lower frequencies. The extent to which a computer can analyze or generate audio

waves is therefore dependent on the intricacy of the modeled environment. Computationally generating the sound of a full orchestra to near perfection would require extremely detailed knowledge of the individual instruments and the physical environment. Such a complex model would then need to track the reflection, bending, and interaction of audio waves about the room. In reality, such complex models are not practical, just as an engineer would not perform a simulation at the atomic level when designing a new type of machinery.

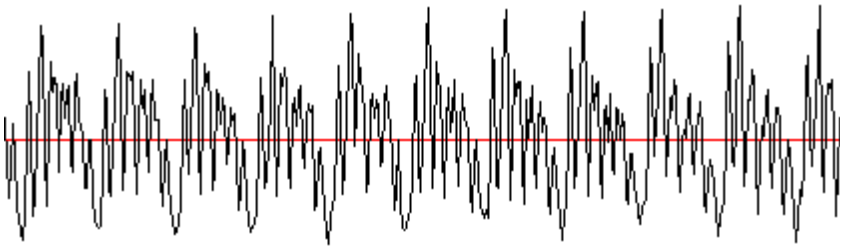

Figure 2-3: Sample Waveform of a Human Voice

Digital Representation of Sound

For a digital computer to process audio, a method of converting audio to and from the domain of digital information is required. The most common format of digitally representing audio information is *Pulse Code Modulation*. Typically, sound waves are converted to a series of numbers (PCM) as follows:

- A simple sine wave will be used as an example. Such a wave would be generated by an object vibrating in a sinusoidal pattern (similar to the pattern made by a whistle). The line through the center of Figure 2-4 represents normal atmospheric pressure. Portions of the sine curve above and below the centerline represent positive and negative pressure changes.

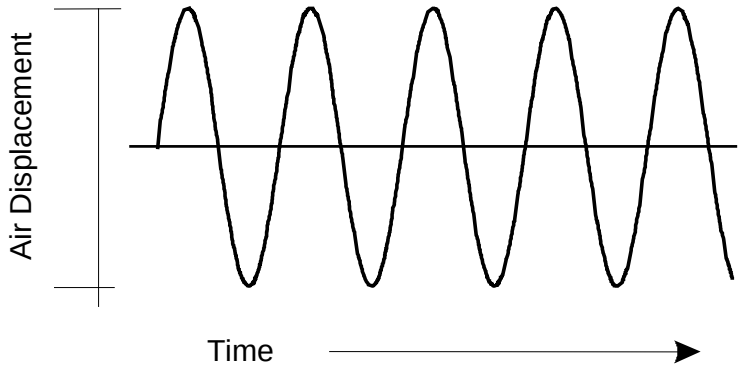

Figure 2-4: Original Audio Wave

- Next, a microphone is used to convert the audio signal (in the air) to an electrical signal. The microphone's output range is ±1 volt in Figure 2-5.

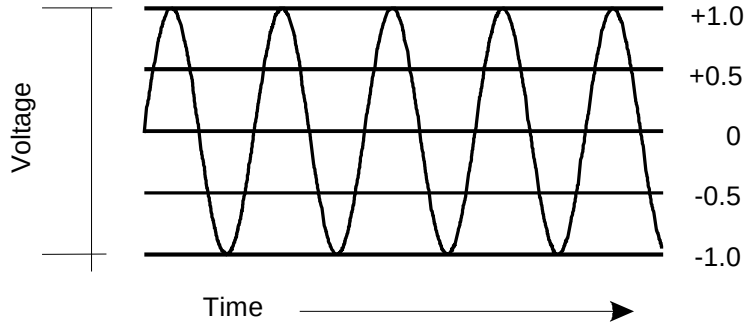

Figure 2-5 : Electrical Wave

- The analog electrical signal voltage is then converted to a numerical value by a device called an *analog-to-digital converter*. A 16-bit analog-to-digital converter, which has an output range of integers from –32,768 to +32,767, is illustrated in Figure 2-6.

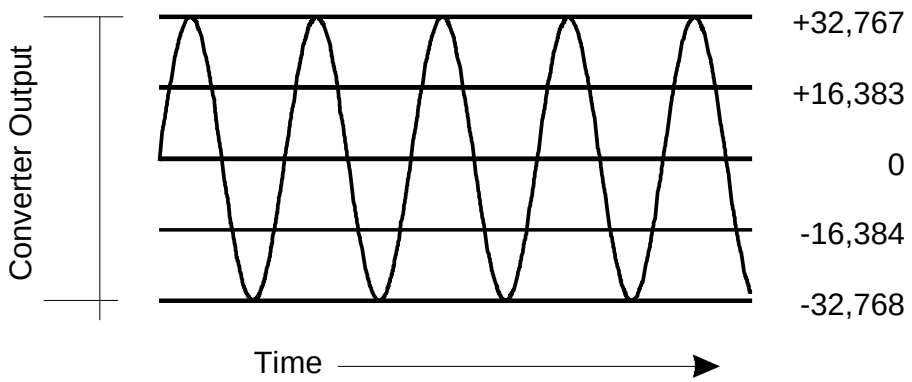

Figure 2-6: Analog-to-Digital Converter Output

- Because an infinite number of data points can not be recorded to characterize the waveform, a *sample* is taken at regular intervals. The number of samples taken per second is called the *sampling rate*. In Figure 2-4, 43 samples are taken.

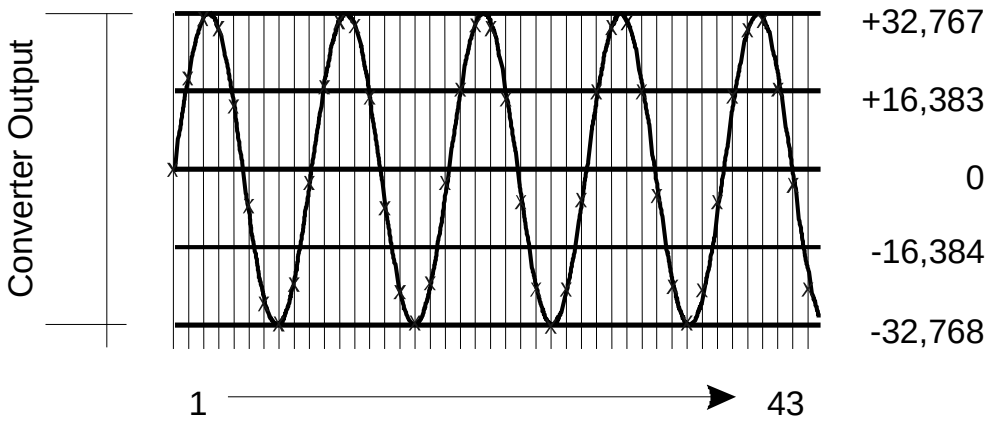

Figure 2-7: Division Into Samples

- The resulting series of 43 numbers represents the wave's position at each interval, as shown in Figure 2-8.

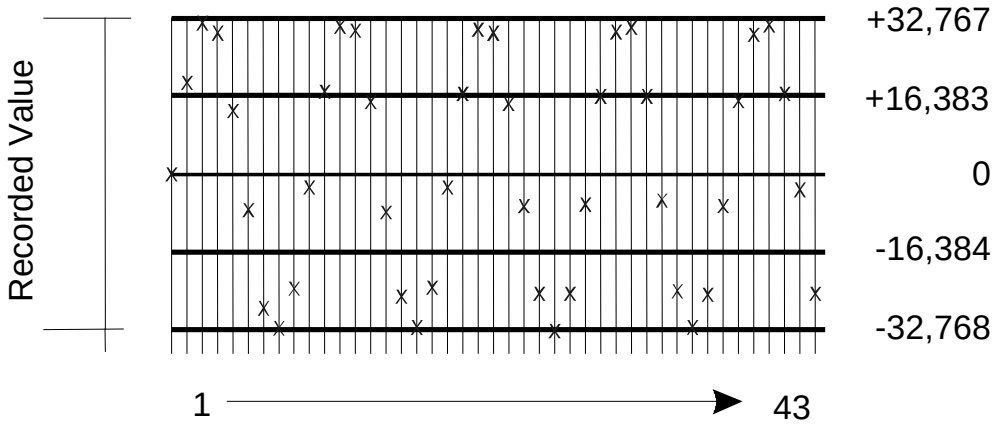

Figure 2-8 Resulting Sample Values

The computer can then re-construct the waveform by connecting the data points. The resulting waveform is illustrated in Figure 2-9.

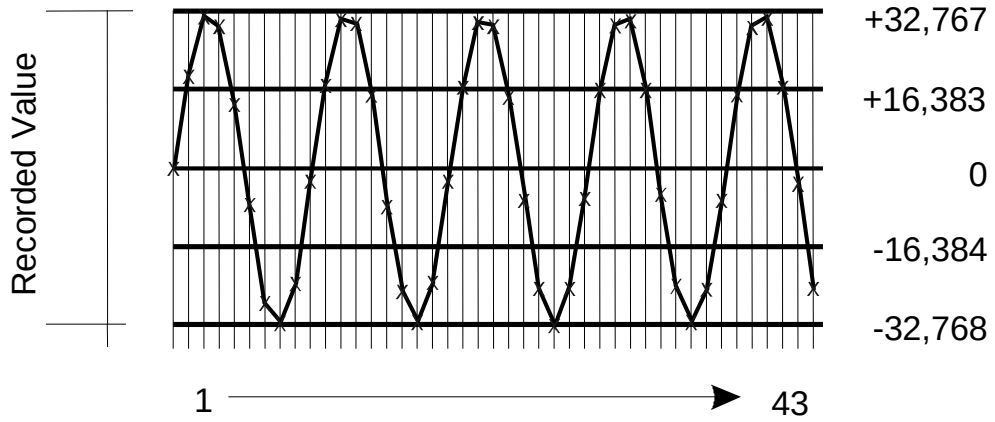

Figure 2-9: Reconstructed Waveform

Note that there are some differences between the original and reconstructed representations (Figure 2-4 and Figure 2-9):

A. The values that the analog-to-digital converter generates are integers and are therefore rounded.

B. The accurate reproduction of the shape of the wave is dependent on the number of samples recorded.

In general, any finite series of numbers (digital representation) can only represent an analog wave (real world representation) to a finite accuracy. It is important to note that most audio hardware does not produce reconstructed waveforms by traversing data points in a linear fashion (as was shown in Figure 2-9). The electrical characteristics of the equipment's *digital-to-analog converter,* the device that converts data points to corresponding voltage levels, usually result in a moderately smooth curved waveform.

Sampling Rates

When converting a sound to digital information, an important issue is the number of samples per second to be taken. The sampling rate necessary depends on the fidelity required.

According to Nyquist and Shannon, the sampling rate determines the maximum frequency information preserved. To create a wave with a frequency of *F* requires 2*F* samples per second. However, the *Nyquist rate* is not sufficient in most cases. If a sine wave of 500Hz (cycles per second) is sampled at a rate of 1000Hz, it is possible that all samples could be taken when the waveform is at a "zero crossing." Sampling at slightly higher rates than 2*F* can cause some "strange effects" of varying amplitude and added noise. (Currington, 1995) In general, the higher the sampling rate, the more accurately lower frequency components of a wave will be reproduced.

It is possible that selection of a sampling rate may have greater repercussions that discussed here. Ken Pohlmann (1995, 49-50) writes:

> *Before we close the book on discrete time sampling, we should mention a current hypothesis concerning the nature of time. We mentioned that time seems to be continuous. However, some physicists have recently suggested that, like energy and matter, time might come in discrete packets. Just as this book consists of a finite number of atoms and could be converted into a finite amount of energy, the time it takes you to read the book might consist of a finite number of time particles. Specifically, the indivisible period of time might be 1 X* 10^{-42} *second (that's a 1 preceded by a decimal point and 41 zeros). The theory is that no time interval can be shorter than this, because the energy required to make the division would be so great that a black hole would be created and the event would be swallowed up. If any of you out there are experimenting in your basements with very fast sampling rates, please be careful.*

Chapter 3 - Introduction to Digital Signal Processing with PCs Running Microsoft Windows 95 and NT

PC-Based Audio Hardware

The typical PC's audio hardware consists of an analog-to-digital converter (recording) and a digital-to-analog converter (playback) that is connected to special audio controller hardware. Although these are logically separate components, they are often found on a single silicon chip.

The computer's software is responsible for sending commands to the audio controller and allocating memory for audio data. The audio controller then transfers chunks (buffers) of audio data directly to or from the computer's RAM by the use of a Direct Memory Access (DMA) channel. DMA channels are much more efficient for this type of transfer, rather than involving the CPU in transferring large chunks of data. (Bartee, 1985, 382) The controller is also responsible for the fine-scale timing of the samples while playing or recording.

The audio controller and converter components are usually embedded in the PC's motherboard circuitry or located on a PC expansion board, called a sound card. Most audio hardware for PCs includes 8-bit or 16-bit converters and supports sampling rates of up to 44100Hz (44100 samples per audio channel per second).

Figure 3-1 illustrates how a program might play digital audio. The program packages audio data into buffers, which are transferred by a DMA channel to the audio hardware. Similarly, as shown in Figure 3-2, the audio hardware, by recording, generates a series of samples and stores them into RAM buffers by using a DMA channel.

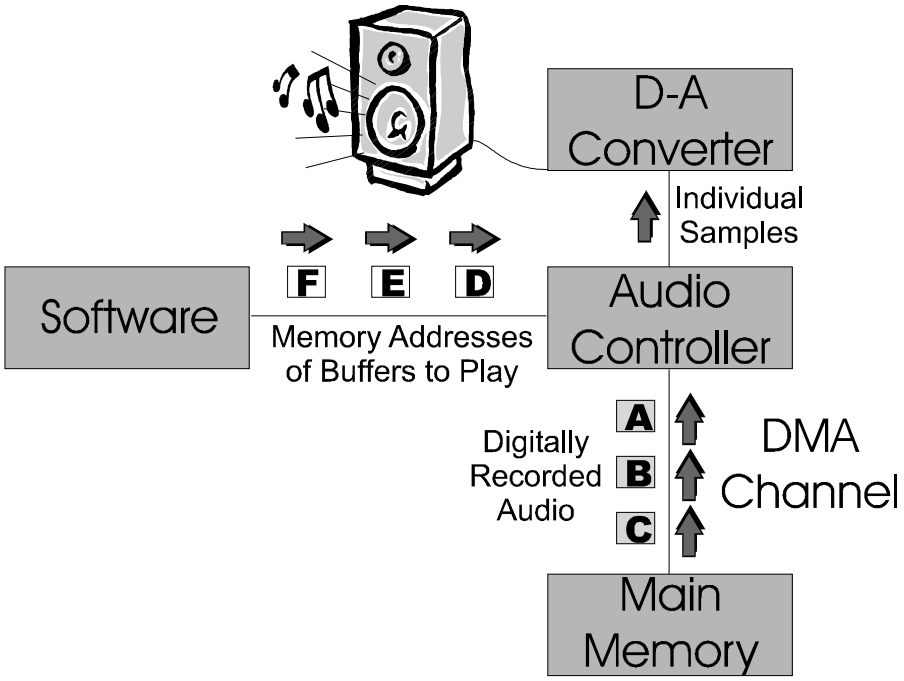

Figure 3-1: Digital Audio Playback

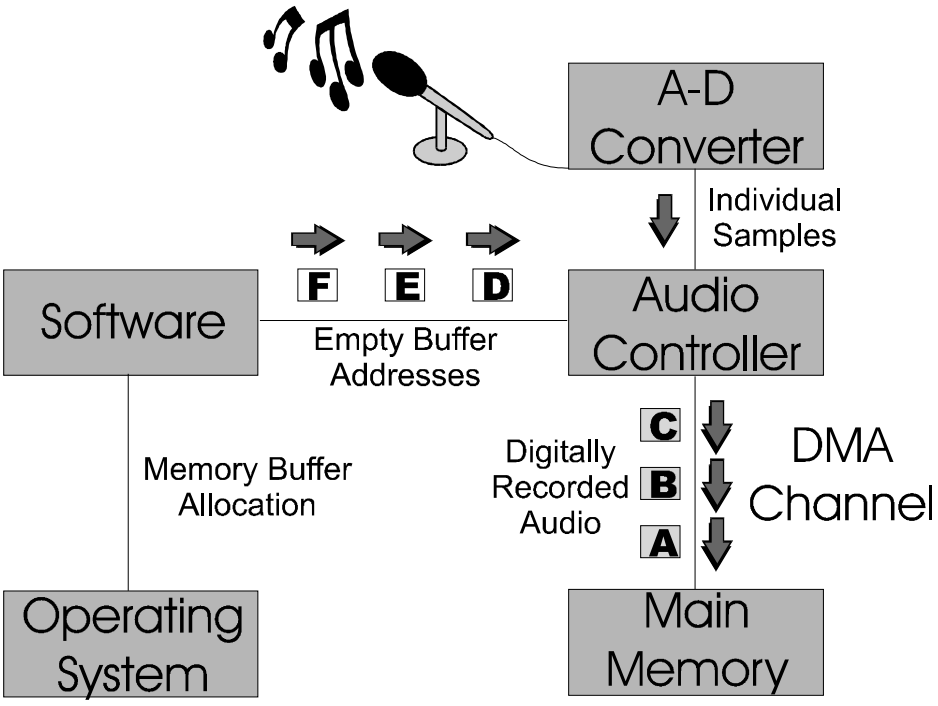

Figure 3-2: Digital Audio Recording

Many modern operating systems have built-in audio support, relieving the application programmer from the details of communicating with audio hardware. Instead, the programmer simply allocates memory buffers for recording or playback and relies on the operating system for proper sequencing and notification of audio events.

A series of recording and playback buffers can be used on two interconnected computers for a telephone-like application, as shown in Figure 3-3. Each of these computers must have *full-duplex* capability and a sufficiently fast connection to each other.

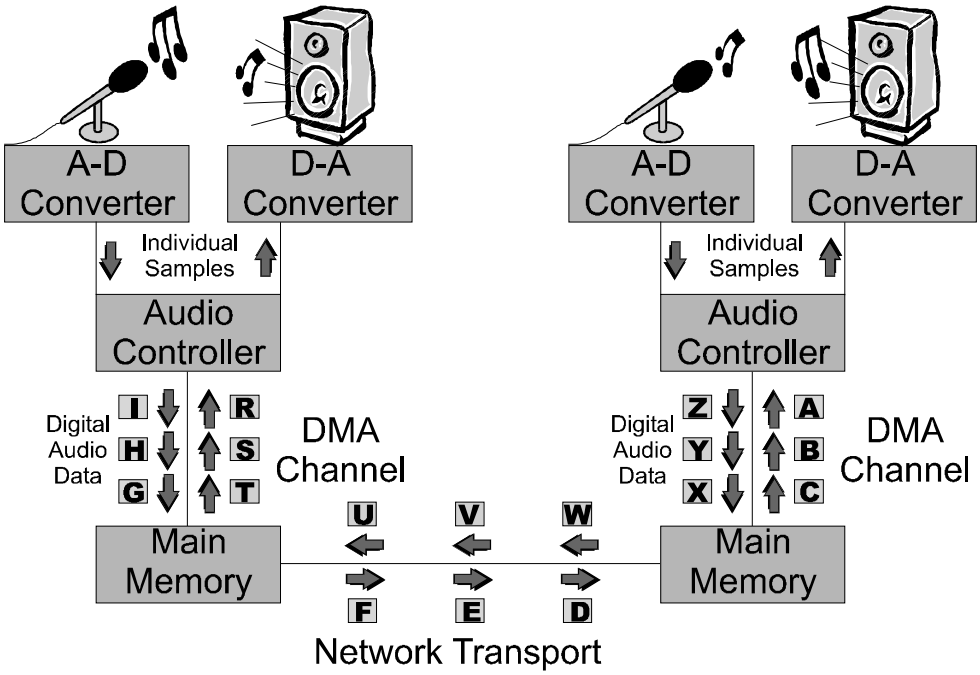

Figure 3-3: Full-Duplex Audio With Two Computers as a Telephone-Like Application

Figure 3-4 illustrates how a series of buffers can be recycled continuously to play and record at the same time. The buffers of data produced by the recording process can be modified or examined before using the data for the playback process. After a buffer has been used for playback, it can be reused as a recording buffer, assuming that the recording and playback buffers are the same size.

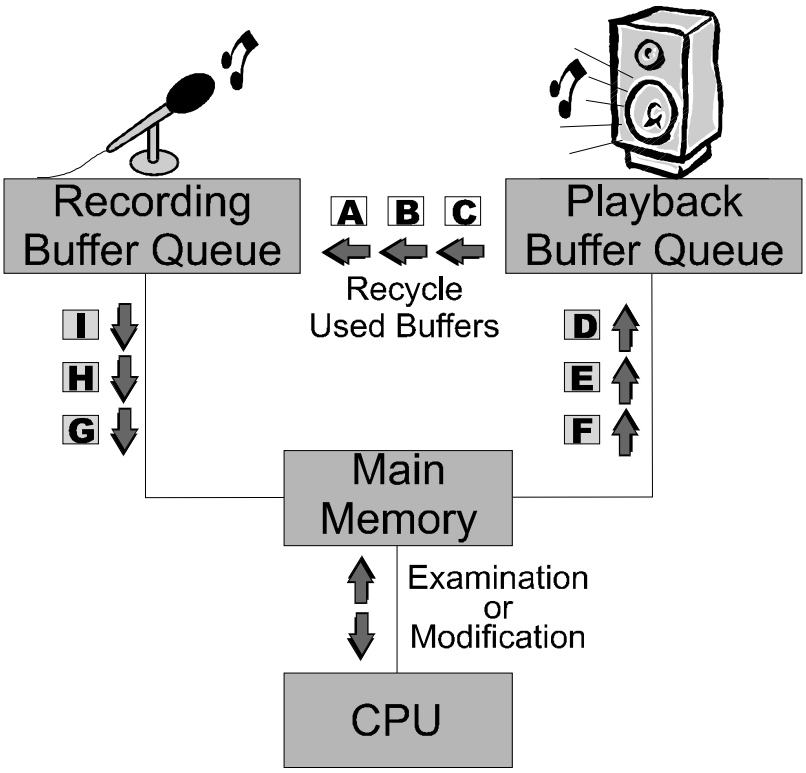

Figure 3-4: Full-Duplex Audio on a Single Computer by Recycling Data Buffers

Full-duplex audio is not without its snares. Not all PC audio hardware supports full duplex operation. Additionally, many audio equipment manufacturers do not supply full-duplex software for operating system support. This limitation can be overcome in either case by installing an additional independent sound card in a computer.

At this point, it becomes obvious that audio processing on a PC can not be "instantaneous." Even if the audio information does not need to travel across a network, a small delay is generated by packing data into buffers and transferring these buffers to and from memory. Specifically, the minimum delay between recording and playback in the previous example (Figure 3-4) would be: (*time to record a buffer + 2*time for a DMA transfer*).

A computer program is labeled as "real-time" if it must complete a certain task within specific time constraints (Deitel, 1984, 8). Similarly, "real-time" audio processing for PCs can be performed if the audio input and output can keep up with each other, without interruption, allowing some finite delay between recording and playback.

Working With Audio Under Microsoft Windows 95 and NT

Microsoft Windows supplies a rich set of audio functions for high and low level control. This text will focus on low-level functions, as they are necessary for examining and changing audio data. Working with audio hardware at the low level involves the following series of steps:

a) If an audio device, other then the default, is to be used, determine the device's number.

b) Referring to the audio device by number, get information about the device's capabilities.

c) Referring to the audio device by number, open the device. Opening a device produces a *device handle*.

d) Using the device handle, perform operations with the device.

e) Using the device handle, close the device.

a) Determining the Number of an Audio Device

The number of available audio playback or recording devices can be determined by using:

```
UINT waveInGetNumDevs(VOID);

UINT waveOutGetNumDevs(VOID);
```

b) Getting Detailed Information About a Device

After enumerating the available devices, details about each one can be obtained by using:

```
MMRESULT waveInGetDevCaps(UINT uDeviceID,
                          LPWAVEINCAPS pwic,
                          UINT cbwic);

MMRESULT waveOutGetDevCaps(UINT uDeviceID,
                          LPWAVEOUTCAPS pwoc,
                          UINT cbwoc);
```

The LPWAVEINCAPS and LPWAVEOUTCAPS structures look like this:

```
typedef struct {
    WORD      wMid;
    WORD      wPid;
    MMVERSION vDriverVersion;
    CHAR      szPname[MAXPNAMELEN];
    DWORD     dwFormats;
    WORD      wChannels;
    WORD      wReserved1; // padding
} WAVEINCAPS;

typedef struct {
    WORD      wMid;
    WORD      wPid;
    MMVERSION vDriverVersion;
    CHAR      szPname[MAXPNAMELEN];
    DWORD     dwFormats;
    WORD      wChannels;
    WORD      wReserved1;  // packing
    DWORD     dwSupport;
} WAVEOUTCAPS;
```

wMid, wPid, and vDriverVersion supply information about the specific hardware and driver versions of the device. This is usually not of interest to the programmer, but may be desired to display information for the user. szPname points to a textual name of the device. This information could be used to supply the user with a list of available audio devices. dwFormats contains information on the particular sampling rates supported. wChannels specifies whether the device supports mono or stereo output. dwSupport contains information about hardware-specific capabilities that not all devices support, such as separate left- and right-channel volume control. (Microsoft, 1995)

c) Opening an Audio Device

Once a device number has been selected, the following functions can be used to open the device, obtaining a *device handle*:

```
MMRESULT waveInOpen(LPHWAVEIN phwi,
                    UINT uDeviceID,
                    LPWAVEFORMATEX pwfx,
                    DWORD dwCallback,
                    DWORD dwCallbackInstance,
                    DWORD fdwOpen);
```

```
MMRESULT waveOutOpen(LPHWAVEOUT phwo,
                     UINT uDeviceID,
                     LPWAVEFORMATEX pwfx,
                     DWORD dwCallback,
                     DWORD dwCallbackInstance,
                     DWORD fdwOpen);
```

`phwi` and `phwo` are the device handles. `UDeviceID` is the numerical identifier of the device to be opened. `pwfx` contains details on the sampling rate and data format to be used. `dwCallback` and `dwCallbackInstance` contain information that indicate the desired action (nothing, execute a procedure, etc.) when an audio event (block finishes playing, recording, etc.) occurs. `fdwOpen` contains a set of flags to enable various options when opening the device. (Microsoft, 1995)

d) Audio Device Operations

To play a block of audio data:

1. Prepare the data block, using `waveOutPrepareHeader`
2. Send the prepared data block to the device, using `waveOutWrite`
3. Clean up the block preparation after it has played, using `waveOutUnprepareHeader`

To control the playback of audio blocks, use `waveInStart`, `waveInStop`, and `waveInReset`.

To record a block of audio data:

1. Prepare the data block, using `waveInPrepareHeader`
2. Send the prepared data block to the device, using `waveInAddBuffer`
3. Clean up the block preparation after it has been filled with recorded data, using `waveInUnprepareHeader`

To stop the recording of audio blocks, use `waveOutReset`.

e) Closing an Audio Device

When an audio device is no longer being used, it should be closed. The function for closing an audio input or output device is:

```
MMRESULT waveInClose(HWAVEIN hwi);
MMRESULT waveOutClose(HWAVEOUT hwo);
```

Chapter 4 - Delay, Echo and Reverb

Simple Delay and Echo

Delay, *echo*, and *reverb* are audio effects that are based on a fixed-length delay of a sound. To assist in illustrating these audio effects, this text will use the following symbols to construct diagrams of audio processing processes:

- A box represents a delay or filter
- A triangle represents signal amplitude scaling (gain control)
- A "plus" represents signal adding (mixing)

In general, *delay* and *echo* usually refer to a sound with a delayed version of the same sound added (Figure 4-1). Effects with longer delays are often called *echos*. The delay and echo effects have no exact equivalent in nature, as audio waves in a natural environment bounce back and forth multiple times (no surface completely absorbs a sound wave).

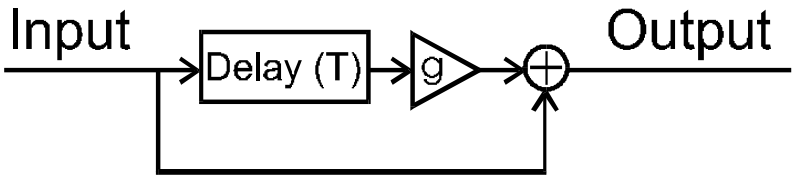

Figure 4-1: Simple Delay

As shown in Figure 4-2, the delay effect results in certain frequencies being "cancelled out," if the length of the delay is such that the output of the delay unit is an inverted form of its input. Because the frequency response curve for this effect consists of several evenly spaced symmetric peaks, like a comb, it is called a *comb filter response*. (Pohlmann, 1993, 402)

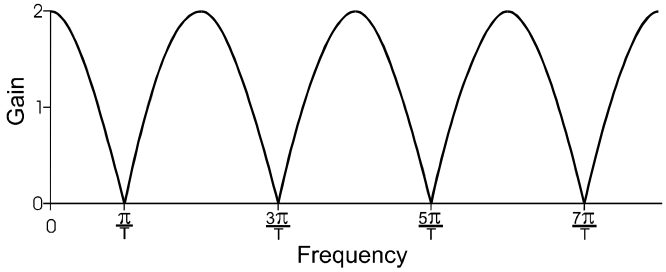

Figure 4-2: Frequency Response Curve of a Simple Delay Effect (Comb Filter)

Simple Reverb

Reverb attempts to imitate the natural reflections or *reverberation* of sound waves in the environment by adding feedback to the delay system. In a small room, for example, a sound might echo back and forth several times before being completely absorbed by the walls. In its simplest form, reverb is a delay with a feedback loop (Figure 4-2). (Currington, 1995)

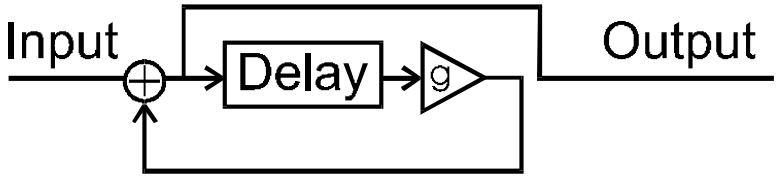

Figure 4-3: Simple Reverb

The recursive nature of reverb creates a more "spiked" comb filter, as shown in Figure 4-4. The longer the delay, the more peaks. (Pohlmann, 1993, 403)

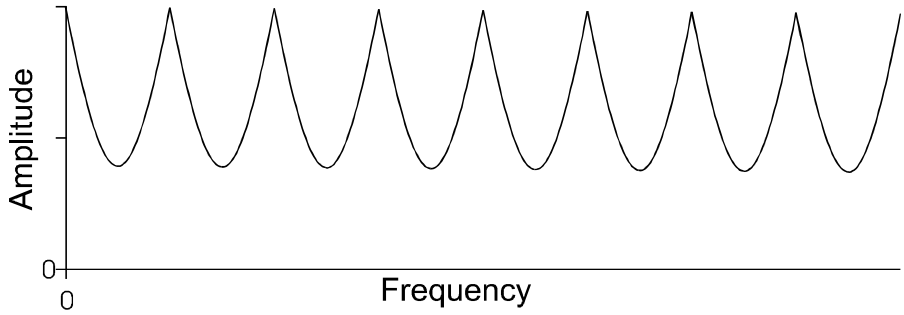

Figure 4-4: Frequency Response Curve of Simple Reverb (Comb Filter)

All-Pass Filters Using a Delay

A feed-forward path can be added to allow a feedback loop while maintaining a flat frequency response (Figure 4-5). This type of structure is called an "all-pass filter," (Lehman, 1996) and can be mathematically proven to have a uniform frequency response (Moorer, 1979, 14).

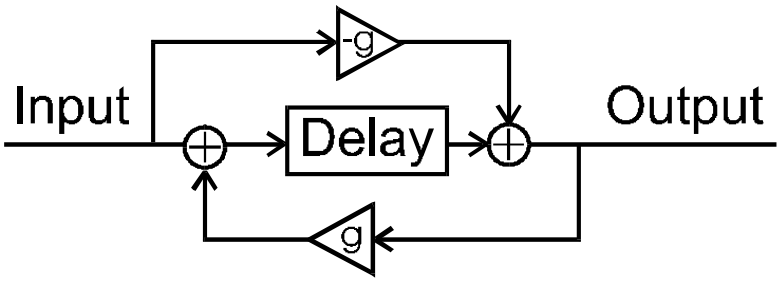

Figure 4-5: All-pass Filter

Complex Delay and Reverb Effects

Multiple delay units can be combined to create interesting effects. Figure 4-6 illustrates how three delay units can be chained together to create multiple delays.

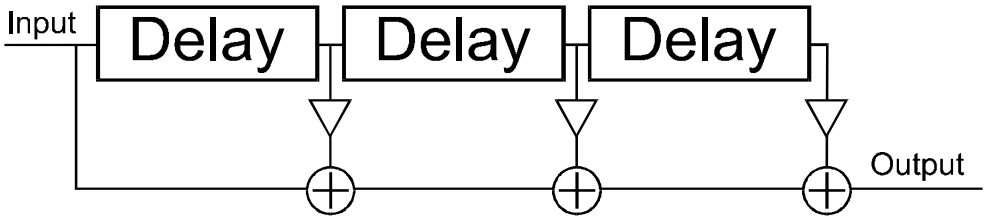

Figure 4-6: A Three-Tap Delay

As shown in Figure 4-7, delay units for separate audio channels can have their outputs crossed to produce "ping-pong" stereo effects. Note that technically, this is not a delay effect, because of the feedback loops involved. However, it is not technically reverb, as none of the delay units have a direct feedback loop.

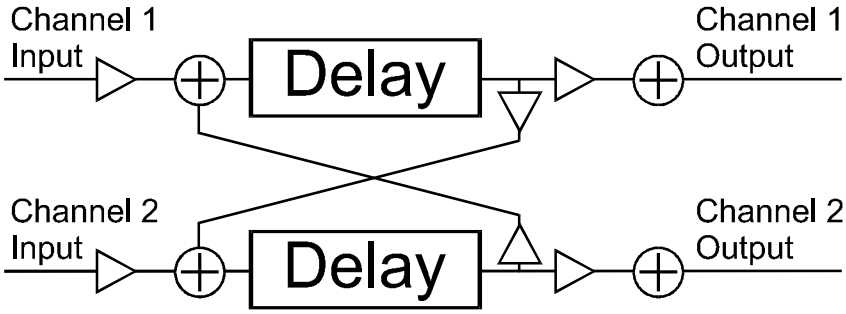

Figure 4-7: Ping-Pong Delay by Crossing Delay Units

Realistic Reverb

In a concert hall, sound waves bounce off of the walls, seats, and floor, creating a torrent of echos from these surfaces. Several people have sought to emulate this effect by studying the reflections caused by a single-source sound impulse in a room. In 1970, M.R. Schroeder reported on several ways to simulate room reverberation by summing the outputs of multiple reverb units. (Moorer, 1979, 14) He first suggested the chaining of several all-pass filters with a single feed-forward loop (Figure 4-8).

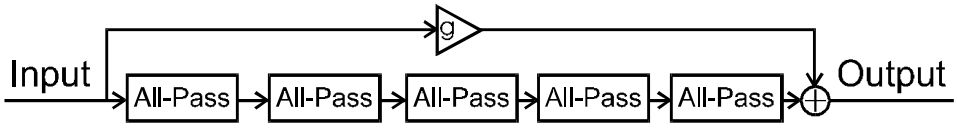

Figure 4-8: Schroeder's Suggested Reverberation Generator (All-Pass Filters)

Moorer (1979, 15) points out the following problems with the results obtained by this configuration:

1. "The decay did not start with a dense sound and die out slowly in an exponential manner. In fact, the higher the order, the longer it took for the density to build up to a pleasing level. This produces the effect of a lag in the reverberation, as if the reverberation followed the sound by some hundreds of milliseconds."

2. "The smoothness of the delay seemed to be critically dependent on the choice of the parameters involved: the gains and delay lengths of the individual unit reverberators. Just changing one of the delay lengths from its prime number length to the next larger prime number, which could be a change as small as 2 samples, has been noticed to occasionally make the difference between a smooth-sounding decay and a ragged-sounding decay."

3. "The tail of the decay showed an annoying ringing, typically related to the frequencies implied by some of the delays. This produced a somewhat metallic sound that was generally found objectionable."

Another configuration suggested by Schroeder was a summation of three comb filters passed into a chain of all-pass filters, as shown in Figure 4-9.

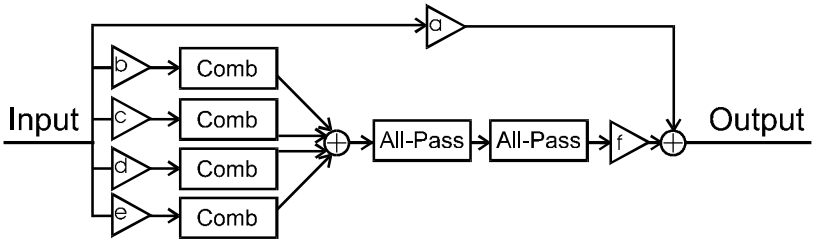

Figure 4-9: Schroeder's Suggested Reverberation Generator (Comb and All-Pass Filters)

Moorer (1979, 16) lists the following undesirable characteristics of this configuration:

1. "Any attempt to reverberate very short, impulsive sounds, such as drum strokes, gave distinct patterns of echos rather than smooth reverberant sounds. The sound was somewhat like flutter echo in rooms with parallel walls."

2. "The decay continued to show a metallic sound, especially with longer reverberation times."

Moorer (1975, 27) concludes, after examining several potential reverb simulation models, that current simulated room reverberation "does not sound at all like real rooms." He attributes this to the following facts, which make simulating room reverberation a complicated task:

1. "The effect of diffusion of echos due to irregularities in the reflecting surfaces"

2. "The fact that the spectrum of the echo is modified by the reflection in a manner that depends strongly on the angle of incidence"

Echo and Reverb Algorithms with Pseudo-Code

Echo and Reverb use a fixed-length delay. Therefore, to implement echo or reverb, the current sample value must be added to a previous sample value. The number of samples to "look back" will depend on the delay length. I used a circular buffer with a size greater than the maximum "look back" length. For each sample to be processed, the echo and reverb algorithms simply store the current sample value in the circular buffer and then combine this value with the "look back" value, generating the output value.

Simple Echo Pseudo-Code

Here is an object-based pseudo-code for simple echo. Note that *Set_Parameters* must be called before any samples are processed. *New_echo_delay* is in milliseconds, and *new_echo_ratio* is a percentage (0-100). To process a series of samples, simply call *Process* with the sample as a parameter. The return values of *Process* are the samples with echo added.

```
Variables
      buffer           Array[ECHO_BUFFER_SIZE] of samples
      write_position   integer
      read_position    integer
      echo_ratio       integer

Constructor()
      fill elements of buffer with value 0
      write_position = 0

Set_Parameters(  new_echo_delay    integer,
                 new_echo_ratio    integer,
                 new_sampling_rate integer)
      echo_ratio=new_echo_ratio
      read_position=(write_position-new_delay*
                     new_sampling_rate/1000)+ECHO_BUFFER_SIZE)
                     MOD ECHO_BUFFER_SIZE

Process(x sample)
      buffer[write_position]=x
      return_value=x+buffer[read_position]*echo_ratio/100
      write_position=(write_position+1) MOD ECHO_BUFFER_SIZE
      read_position=(read_position+1) MOD ECHO_BUFFER_SIZE
      return return_value
```

This effect is extremely fast, as only a few computations per sample are required. However, very high sampling rates can require a substantial amount of processing. If we assume that an average of 10 calculations per sample are needed for this effect, processing stereo at a 44,100 Hz sampling rate (CD quality) will require 882,000 calculations per second to provide this effect! These calculations are in addition to the overhead required to play and record the samples (DMA transfers, operating system calls, etc.).

Simple Reverb Pseudo-Code

The pseudo-code for simple reverb is very similar. The key difference is in *Process*. The feedback loop is implemented by storing the output value in the circular buffer, rather than the original sample (as was done with the echo effect).

```
Variables
      buffer          Array[ECHO_BUFFER_SIZE] of samples
      write_position  integer
      read_position   integer
      echo_ratio      integer

Constructor()
      fill elements of buffer with value 0
      write_position = 0

Set_Parameters( new_echo_delay    integer,
                new_echo_ratio    integer,
                new_sampling_rate integer)
      echo_ratio=new_echo_ratio
      read_position=(write_position-new_delay*
                    new_sampling_rate/1000)+ECHO_BUFFER_SIZE)
                    MOD ECHO_BUFFER_SIZE

Process(x sample)
      return_value=x+buffer[read_position]*echo_ratio/100
      buffer[write_position]=return_value
      write_position=(write_position+1) MOD ECHO_BUFFER_SIZE
      read_position=(read_position+1) MOD ECHO_BUFFER_SIZE
      return return_value
```

Echo and Reverb Effects in E.A.R.

These algorithms were easy to implement in C++. I created dialog boxes to allow the user to enter a delay in the range of 1-1000ms (Figures 4-10 and 4-11). This meant that my circular buffers had to be at least 1000ms long. At a sampling rate of 44,100 Hz, this is a length of 44,100 elements. Because the audio device works with only 8-bit or 16-bit numbers, some range limiting is done to prevent overflows from occurring. In the future, I will probably combine echo and reverb into a single effect, allowing the feedback to be adjustable.

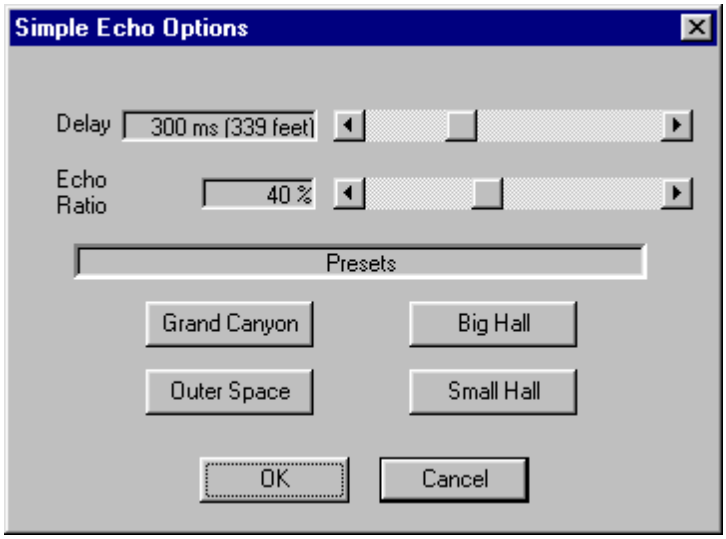

Figure 4-10: "Echo Parameters" dialog box for E.A.R.

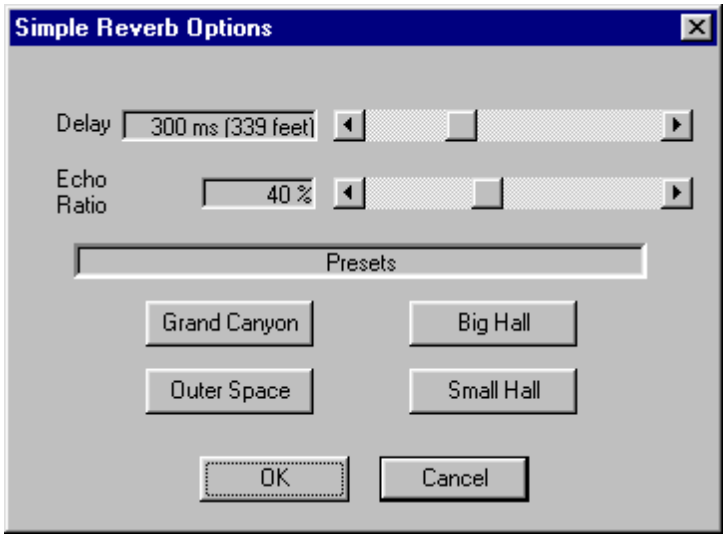

Figure 4-11: "Reverb Parameters" dialog box for E.A.R.

I derived the "preset" values from experimentation and personal preferences.

Chapter 5 - Chorus and Flange

Chorus and flange are two very different sounding effects that are produced by similar processes. Chorus usually describes an effect that sounds like multiple instances of the same sound. A chorus effect might be used to add depth to a single instrument or make a single audio source sound like multiple sources. Using a delay unit with a feedback loop, similar to reverb, creates simple chorus. However, chorus uses a varying delay. As illustrated in Figures 5-1 and 5-2, controlling a simple delay unit with the output of a simple waveform generator can produce a constantly varying delay. Typical waveforms include triangle and sinusoidal.

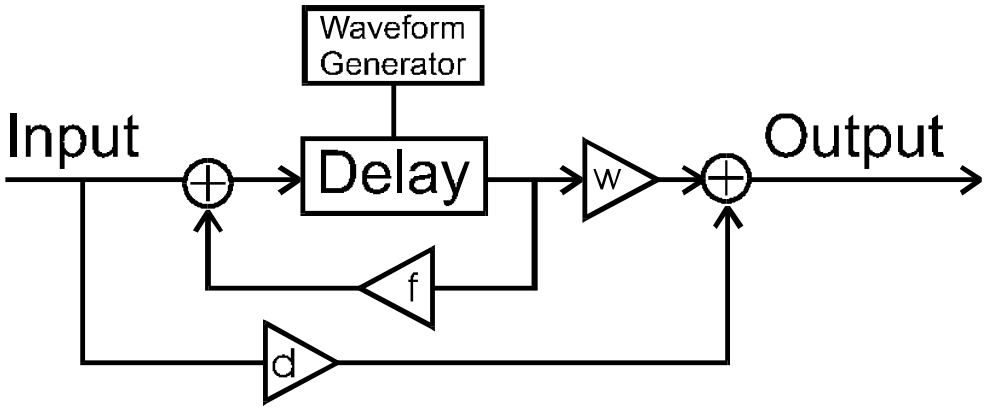

Figure 5-1: Simple Chorus and Flange Unit

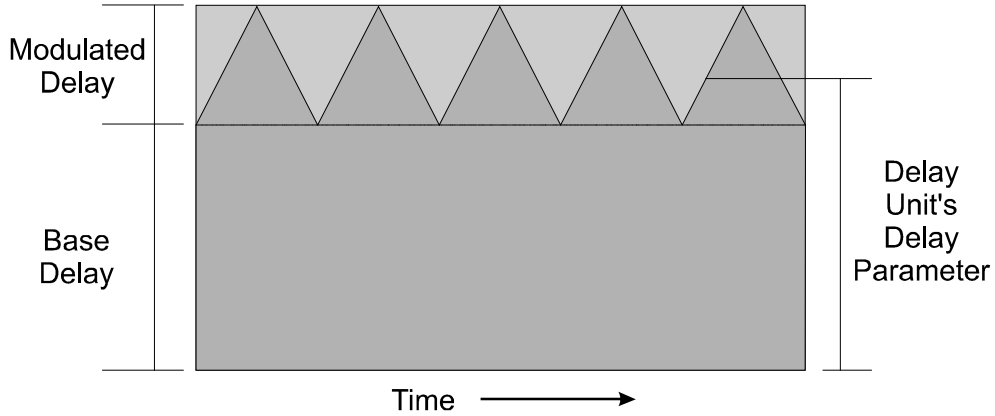

Figure 5-2: Generating a Varying Delay With a Simple Waveform Generator

With the chorus effect, the varying delay corresponds to the slight variations between two "identical" audio sources. For example: if a person sings with a recording of herself, there will be slight pitch differences, as the voices slide in and out of tune with each other. Similarly, there will be a slight, varying delay between the two sounds. Suppose, for simplicity, that the delay is in the range of 60 to 90ns and slowly increases or decreases until one of these boundary points is reached, as shown in Figure 5-2. If the delay is decreasing, the delayed version of the sound is "stretched out" in time, making the pitch lower than the original. When the delay is increasing, the delayed version of the sound is "squeezed together," having a higher pitch than the original sound.

Flange is the same process, with a much shorter delay (typically 15-35ms), and very different effect. The modulated delay creates a "sweeping" effect. The flange effect's name originated from its discovery. Thorderson (1997) writes:

> *Folklore has it that the Beatles discovered the Flanger (variations and accuracy of this tale cannot be verified, so just enjoy the story...OK?). The story goes that they were using a tape machine for their delay (the record head puts sound on the tape, and the playback head...located after the record head...would play the sound back after the original sound...the delay time depended on the tape speed and the distance between the record and playback heads) and JL had the deck running at 15ips for a short, slap back delay.*
>
> *He happened to touch the edge (or 'flange' pronounced flanj) of the tape reel and the pitch of the sound varied some. Being the genius he was, he began tinkering with the sound. One of the results of their experimentation included feeding the delayed, pitch modulating sound back at the tape deck...so much so that it almost began to feedback out of control. The sound started to curl, and sound like it was in a tube...the engineer probably started to reach to fix the problem but JL stopped him...a new sound had just been born. And since the sound is created by gently grabbing and releasing the flanges of the tape real, the sound became known as FLANGING.*

Chorus and Flange Algorithm with Pseudo-Code

Flange uses a delay with a varying, but bounded length. Thus, a circular buffer can be used to provide a "look back" mechanism. The primary difference between the reverb algorithm and the chorus/flange algorithm is the addition of a varying delay.

Simple Chorus/Flange Pseudo-Code

In this object-based pseudo-code for chorus and flange, linear interpolation is performed to allow non-integer delay values (i.e. look back 90.25 samples). This provides a much smoother effect. Note that *Set_Parameters* must be called before any samples are processed. *New_delay* is in milliseconds, corresponding to the "base delay" of the effect (Figure 5.2). *New_depth* is also in milliseconds, corresponding to the "sweep depth" or "modulated delay" of the effect. *New_rate* is the number of "sweep cycles" per second that occur or the reciprocal of the waveform generator's frequency. *New_lfo_waveform* specifies the type of waveform that is used to regulate the delay (triangle or sine). New*_dry*, *new_wet*, and *new_feedback* are percentages (0-100) that set the corresponding mixing levels (Figure 5-1). *Invert_feedback* and *invert_mixing* allow the feedback and wet signals to be inverted. They should have values of +1 or -1. To process a series of samples, simply call *Process* with the sample as a parameter. The return values of *Process* are the samples with the effect added.

```
Variables
      buffer             Array[ECHO_BUFFER_SIZE] of samples
      input_pointer      integer
      echo_ratio         integer
      invert_feedback    integer
      invert_mixing      integer
      feedback           integer
      dry                integer
      wet                integer
      delay              integer
      depth              integer
      rate               floating point
      waveform           WAVE_TRIANGLE or WAVE_SINE
      sampling_rate      integer
      delay_offset       integer
      middle_offset      integer
      samples_per_cycle  integer
      cycle_position     integer
```

```
Constructor()
      fill elements of buffer with value 0
      input_pointer = 0

Set_Parameters(  new_delay           integer,
                 new_depth           integer,
                 new_rate            floating point,
                 new_lfo_waveform    WAVEFORM_TYPE,
                 new_sampling_rate   integer,
                 new_dry             integer,
                 new_wet             integer,
                 new_invert_feedback integer,
                 new_invert_mixing   integer,
                 new_feedback        integer)

      invert_feedback=new_invert_feedback
      invert_mixing=new_invert_mixing
      feedback=new_feedback
      dry=new_dry
      wet=new_wet
      delay=new_delay*new_sampling_rate/1000
      depth=new_depth*new_sampling_rate/1000
      rate=new_rate
      waveform=new_lfo_waveform
      sampling_rate=new_sampling_rate;
      delay_offset=new_sampling_rate*new_delay/1000
      middle_offset=new_sampling_rate*(new_delay+new_depth/2)
                  /1000;
      // middle_offset is = delay_offset + 1/2 depth offset
      // needed for sine LFO calculation,
      // because the sine function
      // varies between -1 and +1, not 0 and 1

      samples_per_cycle=new_sampling_rate/new_rate
      cycle_position=0


Process(x sample)
      x2=x;
      if waveform=WAVE_TRIANGLE then
            if cycle_position<samples_per_cycle/2 then
                  offset=input_pointer-delay_offset-
                         depth*2*cycle_position/
                         samples_per_cycle
            else
                  offset=input_pointer-delay_offset-
                         depth*2*
                         (samples_per_cycle-cycle_position)/
                         samples_per_cycle
```

```
if waveform=WAVE_SINE then
      offset=(input_pointer-depth*
             sin(cycle_position/samples_per_cycle*2PI)-
             middle_offset+CHORUS_BUFFER_SIZE)
             mod CHORUS_BUFFER_SIZE

      // get a second offset to do linear interpolation
      offset1=truncate_to_integer(offset)
      offset2=(offset+1) mod CHORUS_BUFFER_SIZE

      // mix the dry part of the signal
      x2=x2*dry/100;// determine delay unit output
      x3=buffer[offset1]*(1-offset+offset1)+
         buffer[offset2]*(offset-offset1)

      // add wet signal to dry signal for
      // final output
      x2=x2+invert_mixing*x3*wet/100

      // add feedback to original signal and store
      //in delay buffer
      store=x+invert_feedback*x3*feedback/100
      buffer[input_pointer]=store

      // bump up counters
      input_pointer=(input_pointer+1)
                    mod CHORUS_BUFFER_SIZE
      cycle_position=(cycle_position+1)
                    mod samples_per_cycle


      return x2;
```

This effect is a little bit slower than echo or reverb, because more computations per sample are required. Because the number of computations is linearly greater, most computers that can handle the echo or reverb effects in real time should not have much difficulty with chorus/flange.

Chorus and Flange Effects in E.A.R.

This algorithm was easy to implement in C++, once the pseudo-code was worked out. The signal inversion options can be used to prevent out-of-control feedback. I created a dialog box for E.A.R. to allow all of the chorus/flange parameters to be set (Figure 5-3). Some really interesting things happen when you use high modulation rates.

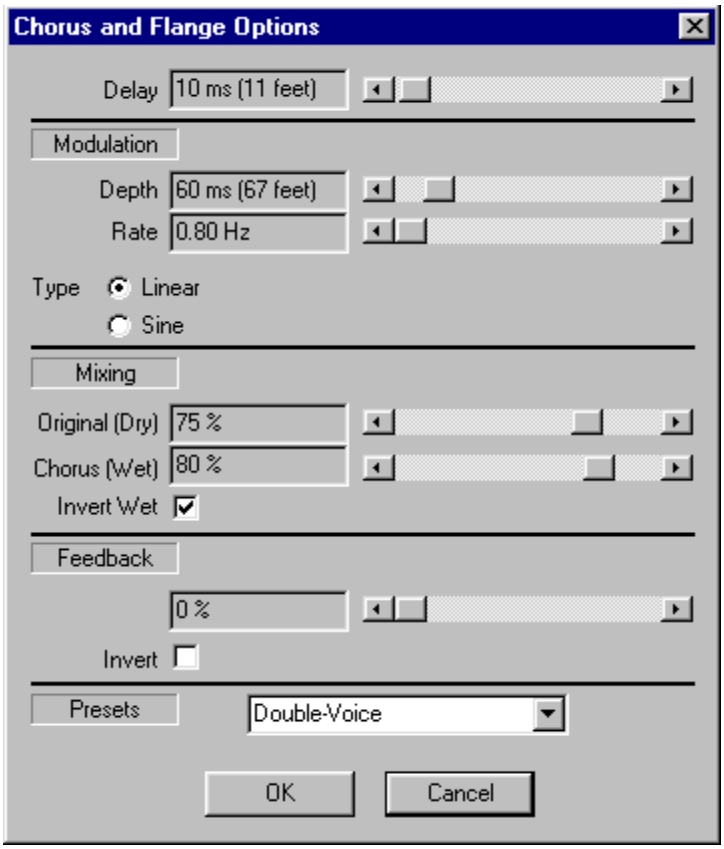

Figure 5-3: "Chorus and Flange Parameters" dialog box for E.A.R.

Chapter 6 - Distortion

Distortion is the modification of an original sound. Thus, any audio effect qualifies as "distortion." Outside of this definition, "distortion" is often the collection of audio effects associated with modifying a sound's amplitude without the use of any temporal information. Because no temporal information is needed for distortion, each audio sample can be independently modified.

Distortion can be used to simulate the effects of an amplifier that is overdriven, and is frequently used with guitars. As shown in figure 6-1, the input signal is mixed with the "distorted" signal. The "distorted" signal is identical to the original signal, unless it exceeds the "distortion threshold." When the signal exceeds the distortion threshold, it is changed to the "clamping level." For example: suppose we had a distortion threshold of 10 and a clamping level of 20. Any signal that reaches 10 or more units from equilibrium is converted to a signal of level 20 (Figure 6-2).

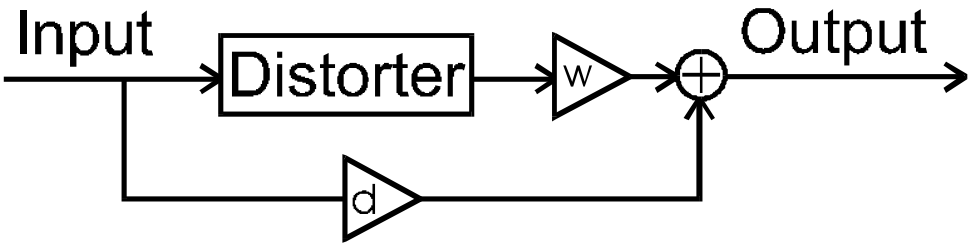

Figure 6-1: Simple Distortion Unit

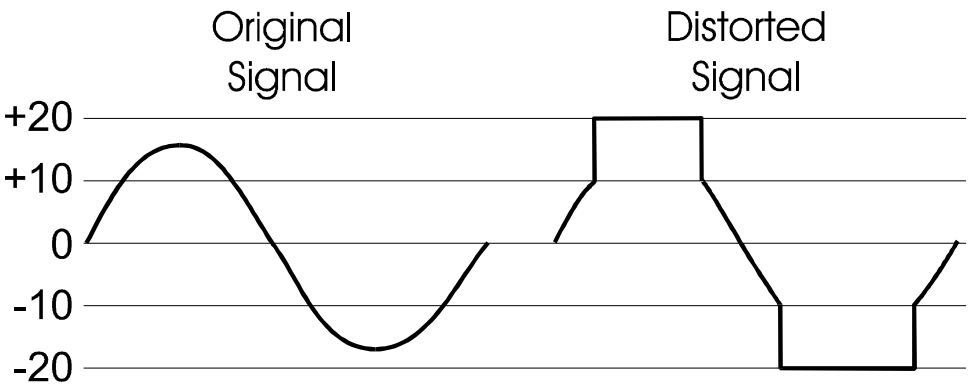

Figure 6-2: Distortion of an Audio Wave

Another form of distortion is called "noise gating. " A noise gate ignores all parts of a waveform below a specified threshold. This allows only louder sounds, or parts of sounds, to pass through the distortion unit (Figure 6-3). Softer sounds would be suppressed completely. An interesting effect is produced for sounds with waveforms that frequently cross the noise gate's threshold.

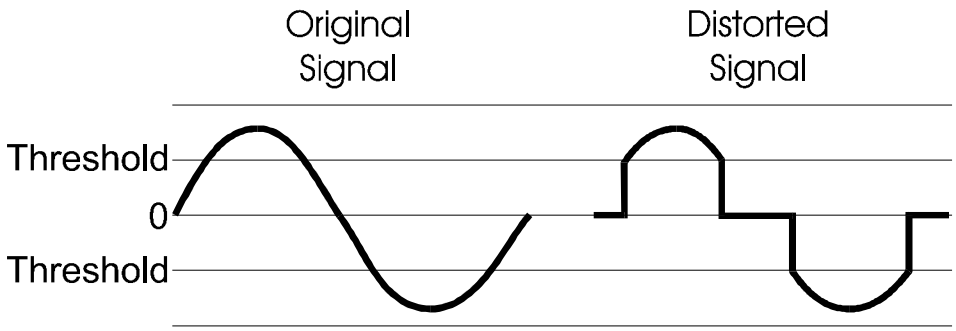

Figure 6-3: Noise-Gate Processing of an Audio Wave

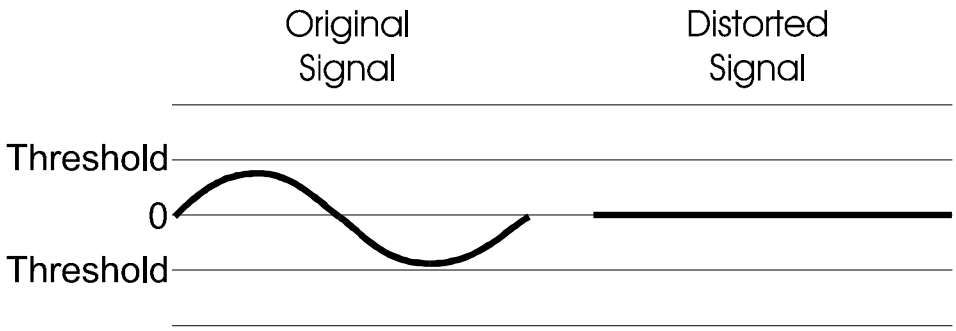

Figure 6-4: Noise-Gate Suppression of a Low-Volume Audio Wave

Distortion Algorithm with Pseudo-Code

Distortion is a really simple algorithm. Each "distorted" sample can be computed with only the value of the original sample. The original sample changes only if it is on the wrong side of the "threshold."

Simple Distortion Pseudo-Code

In this object-based pseudo-code for distortion, I allowed for noise gating. Note that the threshold level applies to positive and negative sample values. *Set_Parameters* must be called before any samples can be processed. *new_gate* is TRUE if noise gating is to be performed. *New_clamp_level* is the "threshold level," in the range 0-MAX, where MAX is the maximum possible sample value. For 16-bit samples, this would be 0-32,767. *New_dry_out* and *new_distorted_out* are mixing percentages (Figure 6-1) in the range 0-100.

```
Variables
      gate              boolean
      dry_out           integer
      distorted_out     integer
      threshold_level   integer
      clamp_level       integer

Constructor()
      fill elements of buffer with value 0
      input_pointer = 0

Set_Parameters(  new_dry_out         integer,
                 new_distorted_out   integer,
                 new_threshold_level integer,
                 new_clamp_level     integer,
                 new_gate            integer)

      dry_out=new_dry_out
      distorted_out=new_distorted_out
      threshold_level=new_threshold_level
      clamp_level=new_clamp_level
      gate=new_gate

Process(x sample)

      // dry signal gain
      x2=x*dry_out/100;

      dist=(double)x;

      // perform distortion if threshold is exceeded
      if dist>=(threshold_level) AND (not gate) then
            dist= clamp_level
      if dist<=(-threshold_level) AND (not gate) then
            dist=-clamp_level
```

```
if  dist<=(threshold_level)) AND
    dist>=(-threshold_level))&&(gate)) then
      dist= clamp_level;
      if (x2<0) then dist=-dist

// distorted signal gain
dist=distorted_out*dist/100

// mix dry signal with distorted signal
x2=x2+dist

return x2;
```

This effect requires about the same computational speed as echo or reverb. In my opinion, it is the least interesting of the effects that I investigated. The noise gating can be used to eliminate low levels of noise (hissing) from silent parts of a sound.

Distortion Effect in E.A.R.

This algorithm was easy to implement in C++, once the pseudo-code was worked out. Some boundary checking was needed to limit the range of output values. I created a dialog box for E.A.R. to allow all of the distortion parameters to be set (Figure 6-5). A closer approximation to the distortion caused by overdriven electrical components (guitar amplifiers) could be made by more smoothly going from the “threshold value” to the “clamping value” over multiple successive samples. Similarly, better noise elimination during periods of silence could be done with noise gating that requires several successive samples below the “threshold value” before going to the “clamping value.”

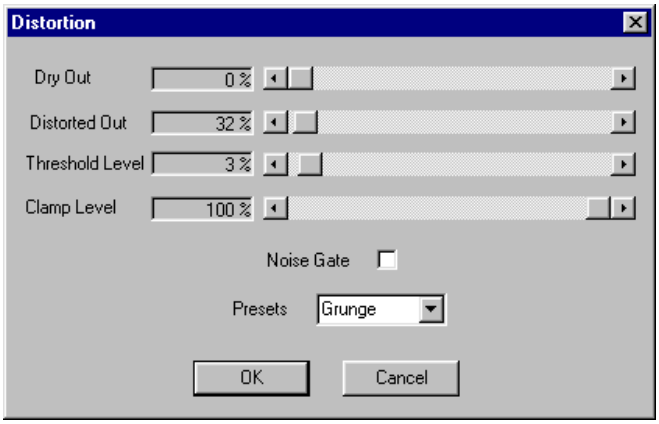

Figure 6-5: "Distortion Parameters" Dialog Box for E.A.R.

Chapter 7 - Working in the Frequency Domain with the Fast Fourier Transform (FFT)

Overview of the FFT, IFFT, and Associated Effects

Because the human ear responds to the intensity of the individual frequencies of sound present, it is often desirable to work with such information. The Fast Fourier Transform (FFT) is an algorithm that allows a finite-length array of sound samples to be converted into an array of frequency intensity information. The inverse FFT (IFFT) converts frequency intensity information back into sound samples. The FFT and IFFT can be used together to change the frequency composition of a finite-length sound. The next section provides an overview of the mathematical basis for converting any periodic function into its frequency components (the Fourier Transform). However, this conversion only works with continuous functions (real sound waves). The following section explains how this mathematical basis can be used to work with discrete functions (functions that are defined only at specific points), such as the discrete functions that are represented by a series of audio samples (the Discrete Fourier Transform). Finally, a shortcut, called a "butterfly operation" allows the Discrete Fourier Transform to be performed in $O(N \log_2(N))$ time complexity, rather than $O(N^2)$ (the Fast Fourier Transform). While the FFT is a well-known algorithm, understanding how it works and how it can be used requires a good understanding of all these concepts. This chapter is concluded with my implementation of the FFT, IFFT, and audio effects that use these transformations.

The Mathematics of the Sound Spectrum

As previously stated, natural sounds are typically composed of multiple frequencies. Suppose that we look at a short section of a sound. For the sake of analysis, we can ignore the rest of the sound. If we assume that the short section of the sound is periodic (repeats over and over endlessly), we can apply the mathematical principles that were developed by Fourier, Bernoulli, and Euler (Zill, 1989, 192).

For any periodic function (waveform) $x(t)$ which has a period of length T, the following is true:

1. *x(t)* can be represented as the sum of a series of sinusoidal waveforms.

2. Of the sinusoidal waveforms, the lowest possible frequency is $f_0 = 1/T$.

3. $\omega = \frac{2\pi}{T} = 2\pi f_0$ is known as *the first harmonic angular frequency* or *fundamental angular frequency*.

4. We can write the summed series of sinusoidal waveforms as
   $x(t) = \frac{a_0}{2} + \sum_{k=1}^{\infty} \left(a_k \cos(\omega nt) + b_k \sin(\omega nt)\right)$ for a certain set of a_k and b_k values.

5. The summation is called a "Fourier series," and the values of a_k and b_k are called "frequency coefficients," corresponding to the amplitudes of frequencies present in the waveform

6. Calculus reveals that

$$a_n = \frac{2}{T} \int_{\frac{-T}{2}}^{\frac{T}{2}} f(t)\cos(\omega nt)dt \quad \text{and} \quad b_n = \frac{2}{T} \int_{\frac{-T}{2}}^{\frac{T}{2}} f(t)\sin(\omega nt)dt .$$

(Zill, 1989, 493)

These results can be applied to discrete functions (functions that are defined only at specific points), as well as continuous functions. If we have a sampling rate of *s* and a series of *N* samples:

1. The lowest possible frequency present is $f_0 = \frac{s}{N}$ and $w = 2\pi f_0$
2. The series of *N* samples can be represented as a set of values of a_n and b_n, such that $x(t) = a_0 + \sum_{k=0}^{N-1} \left(a_k \cos(\omega kt) + b_k \sin(\omega kt)\right)$.

3. The series of N samples can be converted to a set of complex coefficients, y_n, where $y_n = a_n + b_n i$.

4. Integration, to calculate each value of $y_{p,}$ can be performed by computing the sum: $y_n = \sum_{k=0}^{N-1} x_k \left(\cos\left(\frac{2\pi nk}{N} \right) + i \sin\left(\frac{2\pi nk}{N} \right) \right)$ for *n* in the range *0..N-1*.

5. y_0 corresponds to the total value of all the input samples.

6. $y_1..y_{n/2-1}$ correspond to the frequency components present in the waveform, in steps of f_0.

7. The magnitude (amplitude) of each frequency component can be computed as: $magnitude = \sqrt{real\_part^2 + imaginary\_part^2}$

8. The phase angle of each frequency component can be computed as: $phase = \tan^{-1}\left(\frac{imaginary\_part}{real\_part} \right)$

9. It would appear that calculating all values of y_n requires $O(N^2)$ time. However, some shortcuts can be taken to reduce the time complexity to $O(N \log_2(N))$.

(Cross, 1997).

The FFT Algorithm

a) Working With Complex Exponentials

For the sake of being more easily manipulated, the sum

$x(t) = a_0 + \sum_{k=1}^{\infty} \left(a_k \cos(\omega kt) + b_k \sin(\omega kt) \right)$ can be re-written by using

complex exponentials. (Ifeachor and Davis, 93, 50)

This is based on the Euler Identities:

$$\cos(\theta) = \frac{e^{i\theta} + e^{-i\theta}}{2},\ \sin(\theta) = \frac{e^{i\theta} - e^{-i\theta}}{2i}$$

Thus, $x(t) = a_0 + \sum_{k=1}^{\infty}\left(a_k \frac{e^{i\omega kt} + e^{-i\omega kt}}{2} + b_k \frac{e^{i\omega kt} - e^{-i\omega kt}}{2i} \right)$ and

$$x(t) = a_0 + \sum_{k=1}^{\infty}\left(\frac{1}{2}(a_k - ib_k)e^{i\omega kt} + \frac{1}{2}(a_k + ib_k)e^{-i\omega kt} \right)$$

If we let $y_n = \frac{1}{2}(a_n - ib_n)$, we can use the facts that a_n and b_n are the Fourier coefficients of periodic functions, the cosine function is an *even periodic function*, and the sine function is an *odd periodic function* to conclude that $a_{-n}=a_n$ and $b_{-n}=-b_n$ and

$$x(t) = a_0 + \sum_{k=1}^{\infty} y_k e^{i\omega kt} + \sum_{k=-1}^{-\infty} y_k e^{-i\omega kt}$$

$$x(t) = a_0 + \sum_{k=-\infty}^{\infty} y_k e^{i\omega kt}$$

Calculus reveals that $y_k = a_k + ib_k = \frac{1}{T}\int_0^T f(t)e^{-i\omega nt}dt$.

The complex form of the Fourier series is the most widely used form.

(Castillo, 1997)

When moved to the discrete function domain, $y_k = \sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi kn}{N}}$. The values of y_k are called the *Discrete Fourier Transform* (DFT) of the values of x_k.

(Ifeachor and Davis, 93, 57)

b) Butterfly Operations

If we let $W_n = e^{\frac{-i2\pi}{n}}$, then $y_k = \sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi kn}{N}}$ becomes $y_k = \sum_{n=0}^{N-1} x_n W_N^{kn}$.

If the series is broken up into even and odd numbered elements,

$$y_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} W_N^{2nk} + \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} W_N^{(2n+1)k}$$

$$y_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} W_N^{2nk} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} W_N^{2nk}$$

because $W_N^{2nk} = W_{N/2}^{nk}$, if follows that

$$y_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} W_{N/2}^{nk} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} W_{N/2}^{nk}$$

$$y_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} W_{N/2}^{nk} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} W_{N/2}^{nk}$$

If we let $x_1(m)$=EVEN SAMPLES=x(2n) and
$x_2(m)$=ODD SAMPLES =x(2n+1)

$$y_k = \sum_{n=0}^{\frac{N}{2}-1} x_1(n) W_{N/2}^{nk} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} x_2(n) W_{N/2}^{nk}$$

Thus, if there are an even number of sample values, the problem of computing y_k can be cut in half. However, an interesting relationship develops if $y_{k+\frac{N}{2}}$ is examined.

$$y_{k+\frac{N}{2}} = \sum_{n=0}^{\frac{N}{2}-1} x_1(n) W_{N/2}^{n\left(k+N/2\right)} + W_N^{k+N/2} \sum_{n=0}^{\frac{N}{2}-1} x_2(n) W_{N/2}^{n\left(k+N/2\right)}$$

$$y_{k+\frac{N}{2}} = \sum_{n=0}^{\frac{N}{2}-1} x_1(n) W_{N/2}^{nk} - W_N^k \sum_{n=0}^{\frac{N}{2}-1} x_2(n) W_{N/2}^{nk}$$

because $W_{N/2}^{n\left(k+N/2\right)} = W_{N/2}^{nk}$ and $W_N^{N/2} = -1$ (proofs follow)

Proof that $W_N^{N/2} = -1$	Proof that $W_{N/2}^{n(k+N/2)} = W_{N/2}^{nk}$
$W_N^{N/2} = \left(e^{\frac{-i2\pi}{N}} \right)^{N/2}$	$W_{N/2}^{n(k+N/2)} = W_{N/2}^{nk} W_{N/2}^{nN/2}$
$W_N^{N/2} = \left(e^{-i\pi} \right)$	$W_{N/2}^{n(k+N/2)} = W_{N/2}^{nk} W_{N/2}^{2nN/4}$
$W_N^{N/2} = -1$	$W_{N/2}^{n(k+N/2)} = W_{N/2}^{nk} -1^{2n}$ because $W_N^{N/2} = -1$
	$W_{N/2}^{n(k+N/2)} = W_{N/2}^{nk}$

With the results $y_k = \sum_{n=0}^{\frac{N}{2}-1} x_1(n) W_{N/2}^{nk} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} x_2(n) W_{N/2}^{nk}$ and $y_{k+\frac{N}{2}} = \sum_{n=0}^{2^1} x_1(n) W_{N/2}^{nk} - W_N^k \sum_{n=0}^{2^1} x_2(n) W_{N/2}^{nk}$, a *butterfly operation* can be constructed (Figure 8-1):

$X(k) = X_1(k) + W_N^k X_2(k)$ and $X\left(k + N/2\right) = X_1(k) - W_N^k X_2(k)$

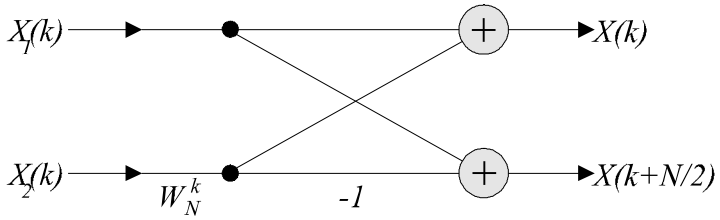

Figure 7-1: A Butterfly Operation (Numbers Along Edges Denote Multiplication)

c) The Algorithm

Note that butterfly operations allow the DFT computation to be broken in half. If there are 2^n samples, the DFT computation can be repetitively broken in half. For a 2-point DFT, the base-case, $y_0 = x_0 + W_2^0 x_1 = x_0 + x_1$ and $y_1 = x_0 - W_2^0 x_1 = x_0 - x_1$.

Thus, if there are 2^n samples, the DFT can be computed by the use of a series of butterfly operation stages (Figure 8-2).

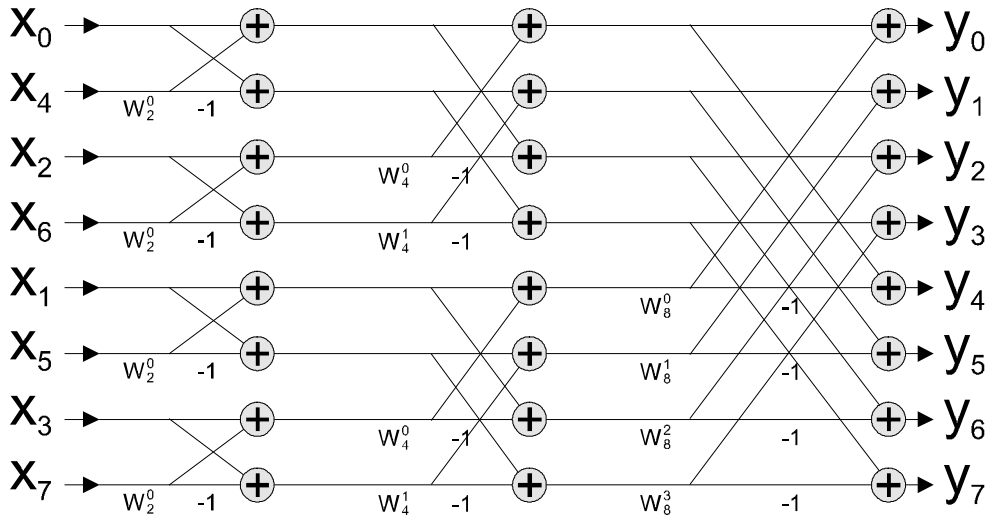

Figure 7-2: Use of a Series of Butterfly Operations to Compute an 8-Point FFT

This series of operations can be more easily implemented if the input values can be sequenced properly. Analysis of the binary representation of the input value subscripts reveals an underlying pattern. Specifically, the indices correspond to the reversed binary representation of the sequence number, as shown in Figure 8-3.

Required sequence for butterfly computation	Binary address of required sequence data	Bit-reversed addresses	Corresponding element of original data sequence
x_0	000	000	x_0
x_4	100	001	x_1
x_2	010	010	x_2
x_6	110	011	x_3
x_1	001	100	x_4
x_5	101	101	x_5
x_3	011	110	x_6
x_7	111	111	x_7

Figure 7-3: Sequence Re-ordering by Bit-reversal (Ifeachor, 73)

FFT Algorithm with Pseudo-Code

The algorithm would then look like this:

```
FFT(X,Y)                                X=inputs  Y=outputs
0   Bit-Reverse-Order(X,Y)
2   n=length[X]                         n is a power of 2
3   for s=1 to log2(n)
4       m=2^s
5       w_m = e^(2πi/m)
6       w=1
7       for j=0 to m/2-1
8           for k=j to n-1 step m
9               t=wY[k+m/2]
10              u=Y[k]
11              Y[k]=u+t
12              Y[k+m/2]=u-t
13          w=w w_m
14  return A
```

This algorithm runs in O($n \log_2(n)$) time.
(Cormen, 1990, 794)

Cooley (1965) and Tukey brought attention to the FFT, as they realized the computational potential that it presented. However, Runge and K¬nig were probably the first to devise it (Cormen, 800).

The IFFT

An alternate form of the Fourier transform is $y_k = \int_{-\infty}^{\infty} x_t e^{-i\omega nt} dt$.

Calculus reveals that $x_k = \frac{1}{2\pi} \int_{-\infty}^{\infty} y_t e^{i\omega nt} dt$.

This corresponds to $y_k = \frac{1}{N} \sum_{n=0}^{N-1} x_n e^{\frac{i2\pi kn}{N}}$ and $x_k = \frac{1}{N} \sum_{n=0}^{N-1} y_n e^{\frac{i2\pi kn}{N}}$, for discrete functions.

IFFT Algorithm with Pseudo-Code

This result applies nicely to the FFT algorithm, allowing the negation of a single term (line 5) and simple division (addition of line 0) to "reverse" the FFT.

```
FFT(X,Y)                                X=inputs  Y=outputs
1   Divide all elements of X by length[X]
2   Bit-Reverse-Order(X,Y)
2   n=length[X]                         n is a power of 2
3   for s=1 to log2(n)
16      m=2^s
17      w_m = e^(-2πi/m)
18      w=1
19      for j=0 to m/2-1
20      for k=j to n-1 step m
21      t=wY[k+m/2]
22      u=Y[k]
23      Y[k]=u+t
24      Y[k+m/2]=u-t
25      w=w w_m
```

Using the FFT and IFFT

Because the FFT reveals information about the frequency components of a finite length of sound, it can be used in any application where this information is desired.

1. Frequency Spectrum Display
2. Analysis of Musical Notes in a Recording
3. Tuning an Instrument

The IFFT allows the generation of cyclic waveforms, given the frequency components desired.

The FFT and IFFT can be combined to change the frequency components of a periodic signal. First, the FFT is performed. Second, the frequency coefficients are changed in some way, and finally, the IFFT is performed, resulting in the desired signal.

IFFT Algorithm with Pseudo-Code

Distortion is a really simple algorithm. Each "distorted" sample can be computed with only the value of the original sample. The original sample changes only if it is on the wrong side of the "threshold."

FFT Frequency Analysis Pseudo-Code

In this object-based pseudo-code for frequency analysis, it is assumed that the FFT conversion is already implemented.

```
Variables
      input_pointer;
      total_points;
      real_in[MAX], Ar[MAX];
      imag_in[MAX], Ai[MAX];

Constructor()

set_size(int size)
      total_points=size;
      input_pointer=0;

add(x sample)
      real_in[input_pointer]=x;
      imag_in[input_pointer]=0;
      input_pointer=input_pointer+1;


perform_FFT ()
      // Perform the FFT algorithm previously
      // described, using real_in and imag_in
      // as the complex inputs and using
      // Ar and Ai to store the outputs
      FFT()

      input_pointer=0

get_out_power(x integer)
      return sqrt(Ar[x]*Ar[x]+Ai[x]*Ai[x])
```

First, *set_size* is called with *SIZE*, a power of 2 <=MAX. This will set the size of the block for FFT functions. Next, *add* is called *SIZE* times. This puts the sample data in the buffer to be transformed. Next, *perform_FFT* is called to perform the actual transformation. The results of the transformation are retreived by calling *get_out_power(x)*, where x is in the range 0-(*SIZE*-1). This provides access to the intensity of each frequency possible. Each value of *x* corresponds to a specific frequency. Specifically, they are as follows:

1. x=0: Average of all samples

2. x=1 thru (*SIZE*/2): Frequency=x*sampling_rate/(2**SIZE*)

3. x=(*SIZE*/2) thru (*SIZE*-1): Frequency=sampling_rate*(1-*SIZE*)/(2**SIZE*)

Note that case 3 is the same as case 2, except the order is reversed.

If more than *SIZE* samples need to be processed, the *add*, *perform_FFT*, and *get_out_power* steps can be repeated.

This processing requires much more computational speed than the echo, reverb, or chorus effects. My experiments found that it required a Pentium-based processor to keep up with high sample rates. Windows NT seemed to work better than Windows 95 for this high-computation algorithm. I suspect that this is because Windows NT has a better process-scheduling algorithm than Windows 95.

FFT/IFFT Usage and Effects in E.A.R.

This algorithm was much more difficult than echo, chorus, or distortion to implement in C++. I started with several well-known public domain versions of the FFT conversion. I increased the code's efficiency by pre-computing some sine and cosine values that the algorithm needs to do the complex exponentials. I also pre-computed the bit-reversal indices for a performance increase. To make sure that my FFT implementation was correct, I created a "monitoring window" that displayed the first 128 values of the FFT results (Figure 7-4). The peaks could be used to determine what notes are present in a musical selection. The two bars are the average power levels of the left and right audio channels.

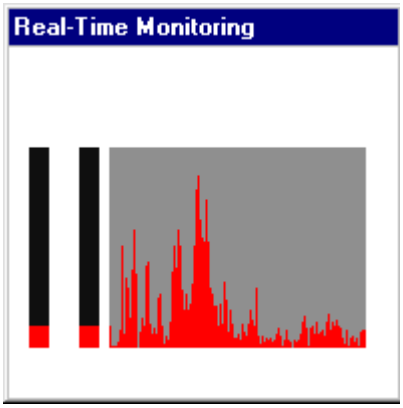

Figure 7-4: "Monitoring Window" for E.A.R.

After determining that my FFT implementation was correct, I created an IFFT implementation and verified that following an FFT with an IFFT produced the original data (a transformation from samples to frequencies to samples). Finally, I added some code to shift or scale the FFT results before the IFFT was performed. I created a dialog box that allows the user to specify how the frequency information is shifted or scaled (Figure 7-5).

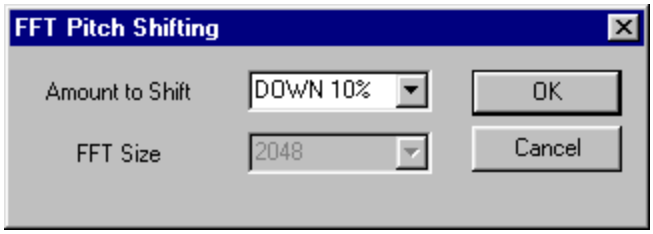

Figure 7-5: "Pitch Shifting Parameters" Dialog Box for E.A.R.

Shifting down the frequency components a certain percentage seems to work well. Shifting up, however, produces some undesirable "warble" and reverb-like effects. This is probably due to my "scale up" method. I plan to get some help from the digital audio community on this. Note that shifting the frequencies by a percentage preserves harmonic relationships (two notes that are an octave apart will still be an octave apart), while shifting by a constant amount does not.

Chapter 8 - Filters and Equalization

The FFT/IFFT combination seems like an ideal way to implement a "filter" that removes certain frequencies. However, from my experiences with pitch shifting, I decided to investigate a different method of filtering with the hope of finding a faster algorithm (FFT/IFFT requires a lot of computations).

FIR (Finite Impulse Response) filters work by multiplying an array of the most recent *n* data samples by an array of constants (called the tap coefficients), and summing the elements of the resulting array (Frohne, 1997). The computation of these tap coefficients is accomplished by an algorithm called "Remez exchange, " and is beyond the scope of this paper.

My investigation of FIR filters was for the purpose of implementing an "equalizer." Equalizers are a series of "band-pass" filters (filters that only allow a certain range of frequencies to pass through). My work was an extension of a digital equalizer developed at C.M.U. by Seet (1997).

Seet developed a C++ algorithm for a single band-pass filter with a frequency range that could be specified. He implemented an equalizer by creating multiple instances of the filter and summing their outputs.

Thus, for a single filter with *L* tap coefficients, b_n, $y_k = \sum_{n=0}^{L} b_n x_{k-n}$, where *y* is a filtered (output) sample and *x* is an original (input) sample (Siew, 1997). Seet's addition of a second filter with tap coefficients c_n would be: $y_k = \sum_{n=0}^{L} b_n x_{k-n} + \sum_{n=0}^{L} c_n x_{k-n}$. Realizing that this was equivalent to $y_k = \sum_{n=0}^{L} (b_n + c_n) x_{k-n}$, I modified Siew's algorithm so that the tap coefficients of each filter are added, resulting in a single filter. For an *n*-band equalizer, this results in a reduction of computation for each sample by a factor of *n*-1.

Graphic Equalizer Algorithm with Pseudo-Code

In this object-based pseudo-code for equalization, an unlimited number of band-pass filters can be used. First, *clear* must be called. Next, for each filter to add to the system, *init* must be called. *Gain_int* is the gain level (-100 to +100) of the band-pass filter being added. *Band* denotes the total bandwidth of the filter (range of frequencies covered). *Center* specifies the middle frequency of the filter. *Srate2* is the sampling rate being used, and *drymix* is the gain level (0 to 100) of the original signal. After the filter is set up, samples can be processed by calling *filt*.

```
Variables
      dry_mix           floating point
      b                 array[SIZE] of floating point
      f                 array[SIZE] of floating point
      x                 array[SIZE] of Samples

Constructor()

clear()
      for counter=0 to SIZE-1
            f[counter]=0
            x[counter]=0;


init( gain_int    integer,
      band        integer,
      center      integer,
      srate2      integer,
      drymix      integer)

      dry_mix=drymix/100;
      gain=gain_int/100;
      halfband=band/2
      w1=(center-halfband)/srate2
      w2=(center+halfband)/srate2

      // call the routine that calculates the fir filter
      // coefficients, placing them in the array "b"
      fir(SIZE,w1,w2)


      // add the new coefficient values to the
      // existing ones
      for counter=0 to SIZE-1
            f[counter]=f[counter]+b[counter]*gain
```

```
filt(input Sample)
      x[0] = input;
      output=0

      // calculate the sum
      for counter=0 to SIZE-1
            output=output+x[counter]*f[counter]

      // shift the sample array
      for counter=SIZE-1 to 1
            x[counter]=x[counter-1]

      output=output+input*dry_mix

      return output
```

This effect requires more computational speed than echo or reverb, but far less than FFT/IFFT-based effects. Most computers that can perform chorus/flange in real time can also perform equalization in real time.

Equalizer Effect in E.A.R.

This algorithm was more difficult than echo, chorus, or distortion to implement in C++. I started with Seet's algorithm for computing the tap coefficients. Next, I incorporated it into the previously described algorithm. Finally, I set up pre-defined filter frequency ranges and created a dialog box that allows the user to control the filter array like a 9-band graphic equalizer (Figure 8-1). I am very pleased with the results.

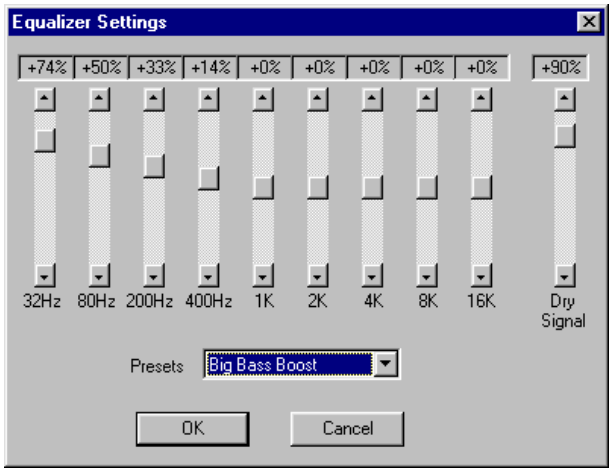

Figure 8-1: "Equalizer Parameters" Dialog Box for E.A.R.

Chapter 9 - Programming E.A.R. - Program Design

Overview of E.A.R.

E.A.R. (Edit Audio in Real time) is a program that allows digital editing of either a live stream of audio or a pre-recorded sound that is stored in a file. I used the following basic design principles:

1. All editing "effects" are implemented as objects. Each object has methods to set parameters, process data, etc. This allows easy code reuse, program consistency, and program readability.

2. The real-time and file processing routines use the same "effect" objects. If there is stereo processing of a monaural effect, two identical "effect" objects are used.

3. All buffer sizes are user-definable so that the "computer guru" can tweak performance.

4. The real-time "effect" settings are stored in memory and may be saved as new default values.

5. The last-used values for each file-based "effect" setting are saved in memory and may be saved as default values.

6. Only the standard sound and file I/O routines are used, as opposed to writing my own low-level software that might be faster, but less portable.

Playback and Recording

The same memory buffers are used for recording and playback when simultaneous recording and playback are occurring. (Figure 9-1) This reduces the overall RAM needed and eliminates the time that moving data from a recording buffer to a playback buffer would require.

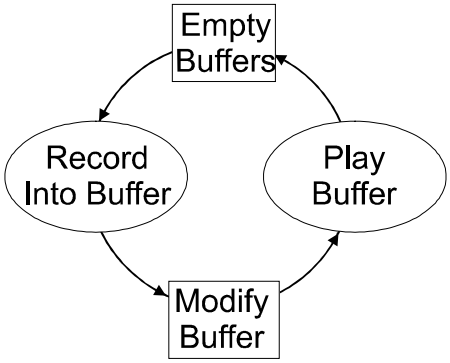

Figure 9-1: Recycling memory buffers for recording and playback

Playback of a loaded sound is simpler. The data is simply packaged into buffers, which are shipped off to the sound card drivers. Buffers are re-used, once they have played. (Figure 9-2)

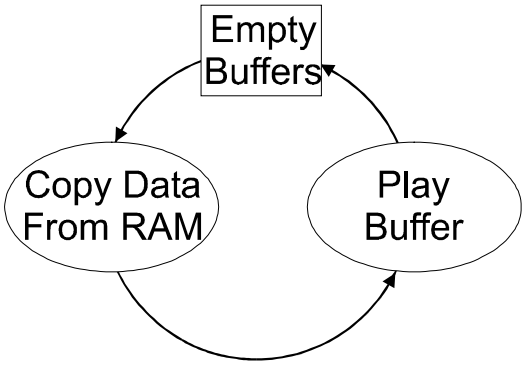

Figure 9-2: Recycling memory buffers for playback

Recording and Playback Buffer Variables in E.A.R.

Object or Variable	Purpose
`WAVEHDR buffers[TOTAL_AUDIO_BUFFERS]`	Contains information about the contents of each buffer (sample rate, memory location, etc.)
int buffer_status[TOTAL_AUDIO_BUFFERS]	Used to keep track of the status of individual memory buffers while playback or recording is occurring.
int status, status_flag;	Indicate the status of the recording/ playback system. Is it recording, recording and playing, idle, or in a critical section?

Playback and recording buffers can be almost any size. For simultaneous recording and playback, smaller buffers allow the delay between the input and output to be reduced. However, smaller buffers require more calls per second to operating system routines, and hence, more overhead. To satisfy all users of E.A.R., I have added dialog boxes to allow the user to "tweak" buffer parameters for his or her computer (Figure 9-3).

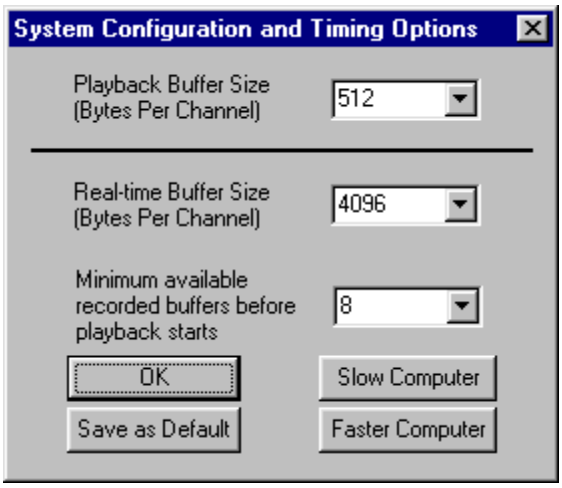

Figure 9-3: "Buffer Sizes" dialog box for E.A.R.

Additionally, I created a "Sound Device Parameters" dialog box for setting up separate input and output devices for users that have two sound cards rather than a single full-duplex sound card. For slower computers, the sampling rate can be lowered to allow real-time operation. (Figure 9-4)

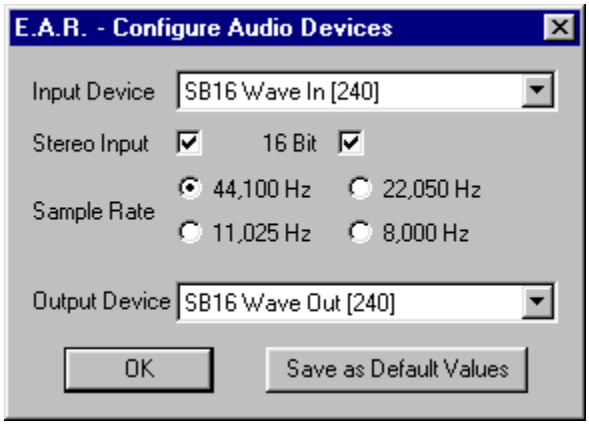

Figure 9-4: "Sound Device Parameters" dialog box for E.A.R.

Before implementing any effects, I added a facility for loading audio data from disk. This allowed the algorithms to be tested with static data. E.A.R. currently can read Sun's .AU and Microsoft's .WAV formats. Using Microsoft's MDI (Multiple Document Interface) facilities, the program treats each loaded sound as a "document." This allows multiple sounds to be loaded simultaneously. Each "document" has an MDI "view," which is a window that displays the document (Figure 9-5). The "view" class for my program includes methods that allow the user to select regions of the sound for processing, "zoom in," and perform clipboard operations.

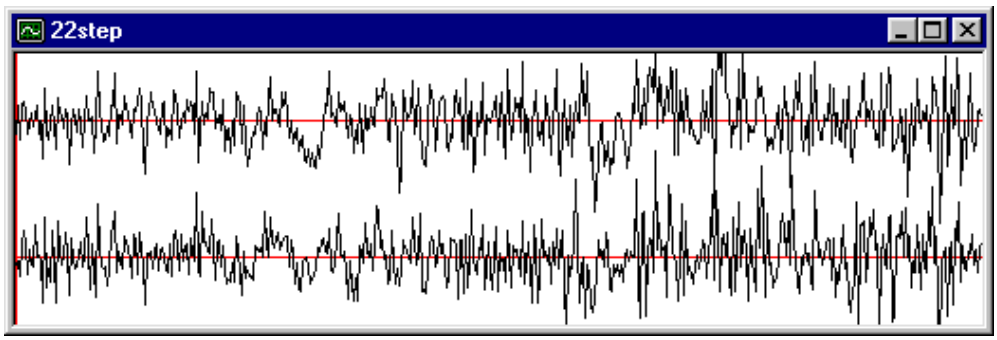

Figure 9-5: Audio "Document" view for E.A.R.

To make the program as intuitive as possible, most functions are available as a series of buttons. I created three "button bars" to categorize the functions. The first bar contains buttons that perform file operations, clipboard operations, and control file playback. The second bar contains buttons that enable the real-time effects and frequency display. The third bar contains buttons that control the file-based effects. The MFC interface directly supports these button bars (Figure 9-6), requiring only the button images and appropriate event-handlers from the programmer.

At this time, I am unsatisfied with several functions. The printing capability could be improved. Currently, the entire waveform in squashed onto a single page. When the waveform is drawn on the screen or printer device such that the number of samples represented is greater than the number of pixels available, my code simply skips some of the samples. This can have an unwanted effect of making the signal appear to be silent when it is not. The clipboard functions do not allow cut and paste operations between windows with different sampling rates, as no functions to perform sample-rate conversion have been implemented.

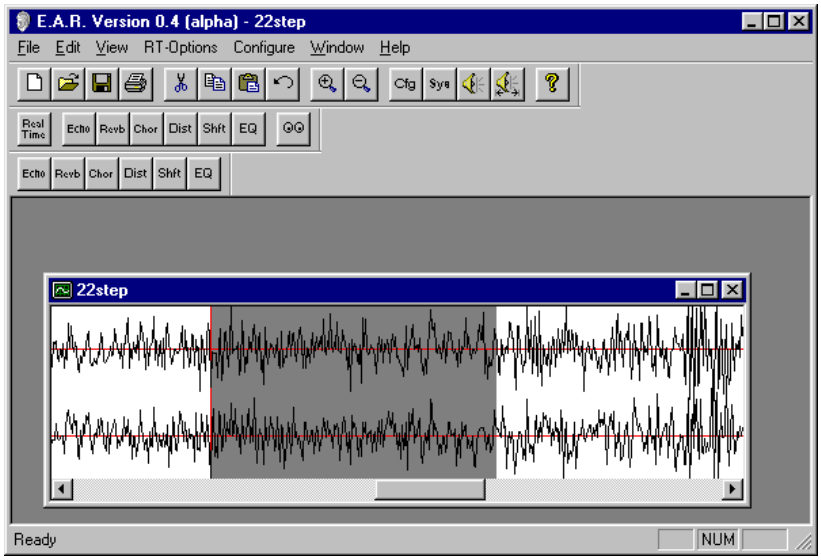

Figure 9-6: MDI design and button bars in E.A.R.

The MDI framework breaks a program up into several classes: program, documents, views, and windows. Figure 9-7 illustrates how the inner-workings of E.A.R. fit into this framework.

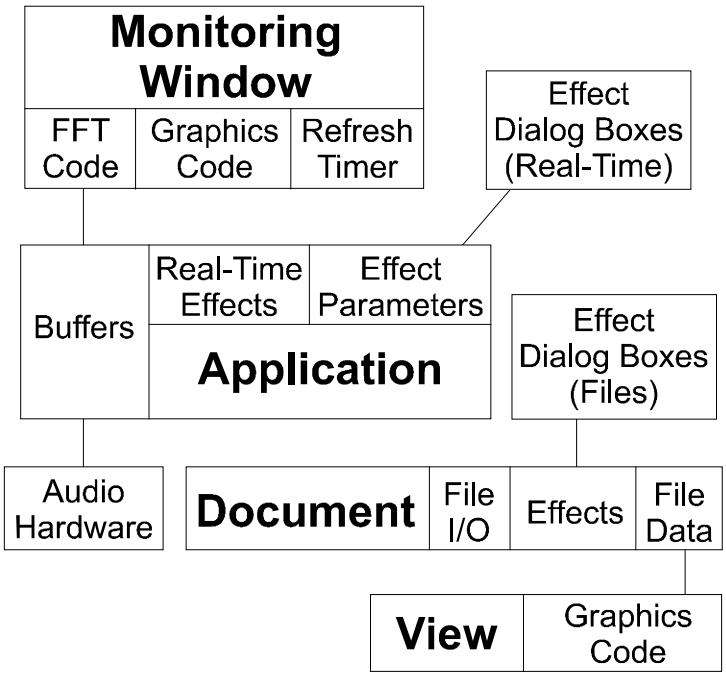

Figure 9-7: MDI framework of E.A.R.

Chapter 10 - Project Results

Overall Conclusions

- Real-time audio is possible with 486 or Pentium-based computers.
- Real-time audio processing is limited by the processing speed of the computer.
- FFT-based processing requires much more processing capability than most other digital effects (echo, reverb, chorus, flange, and distortion), requiring a Pentium-120 or faster to perform FFT-based effects in real time.
- Echo, reverb, chorus, flange, distortion, frequency analysis, and pitch shifting can be performed in real-time with a sufficiently fast computer. Thus, algorithms can be written to perform these operations on a data stream of infinite length.
- With Windows 95/NT, better multimedia performance can be achieved by using callback functions, rather than a timer-driven strategy for scheduling sound buffer I/O. I tried both methods while writing E.A.R.
- Typical computers can process real-time audio with a delay < 0.5 seconds
- Windows NT provides better performance than Windows 95 for multimedia tasks. I am attributing this to process scheduling algorithms, rather than any substantial device driver differences.

Improvements and Future Work to be Done

- Most effect algorithms can have their performance increased.
- The pitch shifting effect needs improvement. Some undesired "side effects" seem to be occurring. This may be partially because the FFT is based on periodic functions. The "windows" of data being processed are not actually periodic (they are in a series of differing windows of data). One solution that has been suggested is somehow "mixing" subsequent windows of data to reduce problems that occur at "window boundaries."
- There are many more "effects" that could be implemented (noise reduction, compression, etc.)
- E.A.R. could have functions for better file-based editing.
- The waveform display in E.A.R. needs to be improved.
- Currently, the real-time effects do not change as the effect parameters are adjusted. Instead, they change when the user presses the "OK" button.

References

Bartee, Thomas C. 1985. Digital Computer Fundamentals. 6th ed. McGraw-Hill, Inc.

Carley, Michael, Johan Nielsen, Chris Ruckman, Asbjoern Saeboe, Jesper Sandvad, and Andrew Silverman. "alt.sci.physics.acoustics FAQ." Oct. 1995. http://rainier.uofport.edu/~dhobbs/FAQ.HTM (9 Sept. 1997)

Castillo, Steven. "Complex Form of the Fourier Series." Feb. 1997. http://emlab2.nmsu.edu/classes/ee313/fssumm/node4.html (5 Nov. 1997)

Cooley J.W. and J.W. Tukry. An algorithm for the machine calculation of complex Fourier series. *Mathematics Computation*. 19. 297-301. 1965.

Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Revest. Introduction to Algorithms. The MIT Press, Cambridge, MA. 1990.

Cross, John. "Fast Fourier Transforms." Nov. 1997. http://www.intersrv.com/~`dcross/fft.html (4 Nov. 1997)

Currington, Michael John. "Audio Effects Frequently Asked Questions." Mar. 1995. http://www.doc.ic.ac.uk/~ih/doc/audio/sampling.faq (9 Sept. 1997)

Deitel, Harvey M. 1984. An Introduction to Operating Systems. Rev. 1st ed. Addison-Wesley Publishing Company, Inc.

Frohne, Rob. "Sidebar: Description of FIR Filters." 1997. http://www.wwc.edu/~frohro/qex/sidebar.html (29 Nov. 1997).

Ifeachor, Emmanuel C., and Barrie W. Jervis. Digital Signal Processing: A Practical Approach. Addison-Wesley Publishers Ltd. 1993.

Lehman, Scott. "Digital Allpass Filter." 1996. http://www.harmonycentral.com/Effects/Articles/Reverb/allpass.html (7 Oct. 1997).

Microsoft Corporation. 1995. Win32 SDK. Online Help Reference. Microsoft Corporation.

Moorer, James A. "About This Reverberation Business." Computer Music Journal. 3, no. 20 (1979): 13-28.

Pohlmann, Ken C. 1993. Principles of Digital Audio. 2d ed. SAMS Publishing.

Seet, Darryl. "Parametric Equalizer." 1997. http://www.ece.cmu.edu/afs/ece/class/ee551/www/projects/s_97_2/equilizer.html (3 Nov. 1997).

Siew, Simon Hong Boon. "IIR and FIR filters." 1997. http://dspserv.sdsu.edu/~simon/firfilters.htm (29 Nov. 1997).

Thorderson, Randy. "Friday Tips." Oct. 1997. http://ee.tamu.edu/~clpastor/qfaq-3a.html (28 Oct. 1997).

Zill, Gennis G. A First Course in Differential Equations with Applications. 4th Ed. PWS-Kent Publishing Company, 1989.

INDEX