SOUND ENHANCEMENTS FOR GRAPHICAL SIMULATIONS

Prepared by

Steven G. Villareal

B.S., Mechanical Engineering (1995)
University of Texas at Austin

Submitted to the Department of Mechanical Engineering in Partial Fulfillment of the Requirements for the Degree of Master of Science in Mechanical Engineering at the Massachusetts Institute of Technology

May 27, 1997

Author _____
Department of Mechanical Engineering
May, 1997

Certified by _____
Professor Thomas B. Sheridan
Thesis Supervisor
Professor of Engineering and Applied Psychology

Accepted by _____
Professor A. Sonin
Chairman, Department Committee on Graduate Students

SOUND ENHANCEMENTS FOR GRAPHICAL SIMULATIONS

Prepared by

Steven G. Villareal

Submitted to the department of Mechanical Engineering on May 27, 1997 in Partial Fulfillment of the Requirements for the Degree of Master of Science in Mechanical

ABSTRACT

With the advancement of processing power and visual displays, simulators are proving to be a cost effective tool for training workers and studying human behavior. Economics play a central role in determining the level of added realism. In general, slight additions in realism are accompanied by large increase in cost.

This research project provides programming tools for adding sound effects and other physical enhancements to graphical simulators in a cost effective manner. These tools are applied to enhance the realism of a high-speed train simulation used in human-factors studies. Sound effects are generated on a PC using digital playback and FM synthesis techniques. An ethernet software library is provided for linking the PC to a local area network running the simulation.

Thesis Supervisor: Professor Thomas B. Sheridan

Title: Professor of Engineering and Applied Psychology, Human-Machine Systems Lab

ACKNOWLEDGEMENTS

It has been a pleasure studying at MIT. During this short time, I have had so many positive experiences and met some truly great friends. The skills I have learned here will last me a lifetime.

The Human-Machine Systems Laboratory is one of the greatest labs at MIT. To my fellow lab member: Jay Einhorn, Mike Timmons, Mike Kilaras, Mark Ottensmeyer, Ed Lanzilotta, Shin Park, Jianjuen Hu, Suyeong Kim, Helias Marinakos, Jim Thompson, and Dave Schloerb, I would like to thank you for sharing your culture and ideas. Working with you has broadened my horizons and I hope I have broadened yours.

It wouldn't be MIT if there wasn't a group of "Tooler" to socialize with. So I thank the Toolbox Club: Larry Barrett, Razman Zambre, Bernardo Aumond, and Karl Iagnemma for always striving to do their best, no matter how many simulations or OP-amp circuits were needed! I wish all of you the best of luck in your careers.

To the Juicy Chicken Intramural Basketball Champions of 1997, thanks for supplying an outstanding season and stress outlet when times were rough; never underestimate the heart of a champion.

I would also like to thank the Department of Transportation Volpe Center's Dr. Jordan Multer and my thesis advisor Professor Sheridan for making this research project available for me. Thanks also to J.K. Pollard for his endless work with the train simulator hardware.

Last, but certainly not least, I want thank my wonderful parents Gilbert and Patricia Villareal for always supporting me and encouraging me to do my very best. They are the ones who instilled in me the philosophy that doing your best always pays off ... once again you are right.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1: INTRODUCTION

1.1 Background

1.1.1 High Fidelity In Simulators

Simulations have proven to be an effective tool in training workers and in studying human behavior. The level of realism need not be the same in both cases. Fidelity must be selected according to the target application. For example, one would expect a very complex flight training simulator but a less complex maintenance training simulation [5]. There is a tradeoff that must be made between adding realism versus cost. The demand for inexpensive simulators is high. In terms of training a person, a simulation can have a significantly lower operating cost as opposed to using real equipment [5]. In general, "small gains in realism often can be achieved only at relatively great incremental cost." [5].

The objective of this project is to create a realistic human interface for simulations used in human factors research in a cost effective manner. The M.I.T. Human-Machine Systems Laboratory (HMSL), under the direction of Professor Thomas Sheridan, routinely develops graphical simulators to test human-in-the-loop scenarios. Two HMSL systems currently in use are driving and high-speed train simulators. The latter is the focal point of this thesis.

One of the most challenging and expensive aspects of a simulator is developing the visual scenes. Both HMSL simulators generate graphics using Silicon Graphics workstations, which balance speed, graphical quality, and cost effectiveness. The ability to model the system, though necessary, is sometimes not enough to generate the most accurate experimental data. A subject participating in a behavioral experiment must also be aware of the surroundings and remain focused on the experimental task. During long

experiments, fatigue and boredom can distract the subject. By increasing situational awareness, the subject may remain vigilant. One way to raise situational awareness is by adding sound and other effects that increase the realism of the simulation.

1.2 Objective

It is important for moving vehicle simulators to give the subject an alternate sense of speed or motion aside from visual feedback. In the case of a car or train simulator, this is extremely important if reaction times or speed limits are invoked. Consider the routine act of driving a car. The speedometer is not the only source of speed information. We hear the tone of the engine, wind turbulence, and the tires moving across the pavement. Each physical cue contributes to a greater sense of speed awareness. For the train simulator, track noise is an important speed indicator. In addition to speed, we also need feedback on the level of power being added to the system. This feedback can come visually by a motor tachometer or current indicator and audibly by a modulating frequency. The latter is used for both the driving and train simulations.

The objective of this design project is to cost effectively add these important sounds into graphical simulations. In the process of adding sound effects to the Volpe high-speed rail simulator, a framework for adding sound to other simulations has also been established. This document provides programming tools for sound creation, networking communication, and external device interface.

Chapter 2: SOUND GENERATION

In theory, it is possible to reconstruct a complex sound from its frequency spectrum provided the original signal is band-limited. Given this information, all the harmonics and their relative amplitudes can be replaced by separately mixing pure tones each having an appropriate loudness; this is what is termed *sound synthesis*. In reality, however, mixing pure tones is not sufficient. As will be shown, a sound is not defined solely by its harmonic content. The transient behavior defined by *timbre* must also be reproduced. Electronic synthesizers accomplish this with relative ease.

There are four basic methods of reproducing sound: (1) additive synthesis, (2) subtractive synthesis, (3) frequency modulation, and (4) digital sampling. The first two will be described briefly for completeness, while the latter two are the focal point of this chapter. Before describing these different methods of synthesis, there are a few term that must be clarified.

2.1 Sound Characteristics

A number of terms will surface as one begins to read sound related literature. The tone, timbre, frequency, and vibrato differentiate one sound from another. It is therefore useful to be familiar with their definitions in order to recreate the desired sounds successfully. This section provides an introduction to some essential terms used when describing the creation of sound with frequency modulation (FM).

2.1.1 Pure Tones and Noise

Pure tones are to sound as atoms are to matter. They are the building blocks of sounds we perceive as being *musical*. Tuning-forks are devices that generate pure tones in order to

correct the musical notes played by various instruments like pianos and guitars. When the prongs of a tuning-fork resonate, the surrounding air vibrates, thus transmitting pressure waves to our ear-drums. The resulting sound waves are periodic and only vary in intensity. This can be simulated by the lightly damped mass-spring-damper (MSD) system shown in Figure 1.

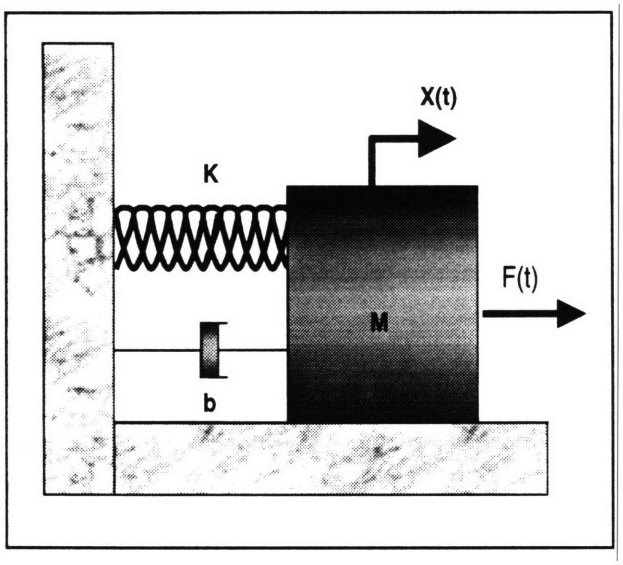


Figure 1. Mass-Spring-Damper System.

In this model, the position of the mass corresponds to the endpoint motion of a tuning-fork. To see how the system behaves by suddenly striking it with a hammer, we need to solve for the impulse response. This sets the system in motion thus revealing all the dynamics (*see Figure 2*).

Figure 2 shows the impulse response of the MSD system with a natural frequency (ω_n) of 20 Hz and a damping ratio(ξ) of 0.03. The system oscillates with a period T=50 msec and has an exponentially decaying amplitude. If humans could hear frequencies this low, we would hear a constant pitch but the loudness fades.

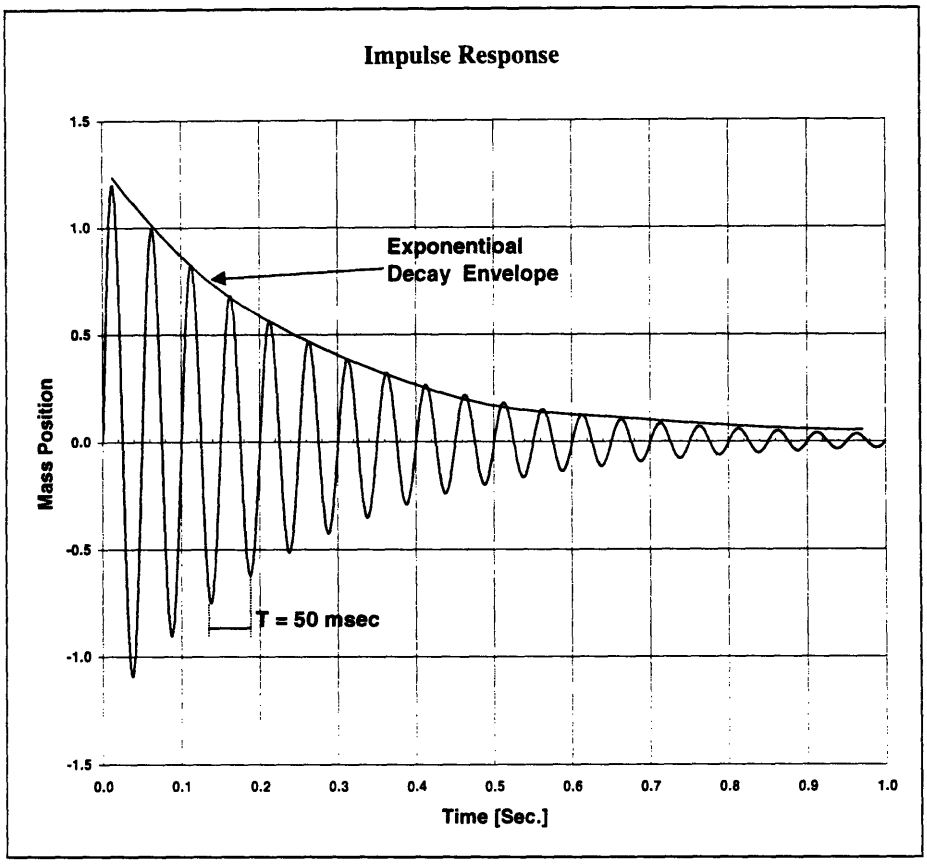


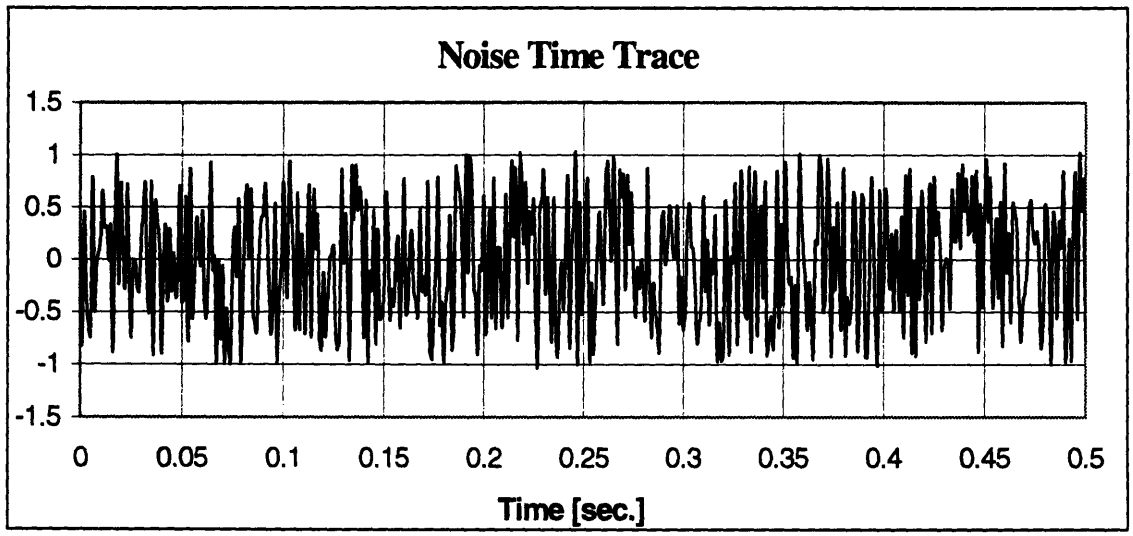
Figure 2. MSD impulse response ($\omega_n = 20$ *Hz*, $\xi = 0.03$).

Noise, as opposed to pure tone, is unwanted sound generated by a broadband signal. Both are sounds, but are differentiated with respect to frequency content. The frequency content of a sound wave can be determined easily using the Fourier Transform. According to the theory, all signals can be represented as the sum of sinusoids. For continuous time signals periodic in T_o, the following equation applies:

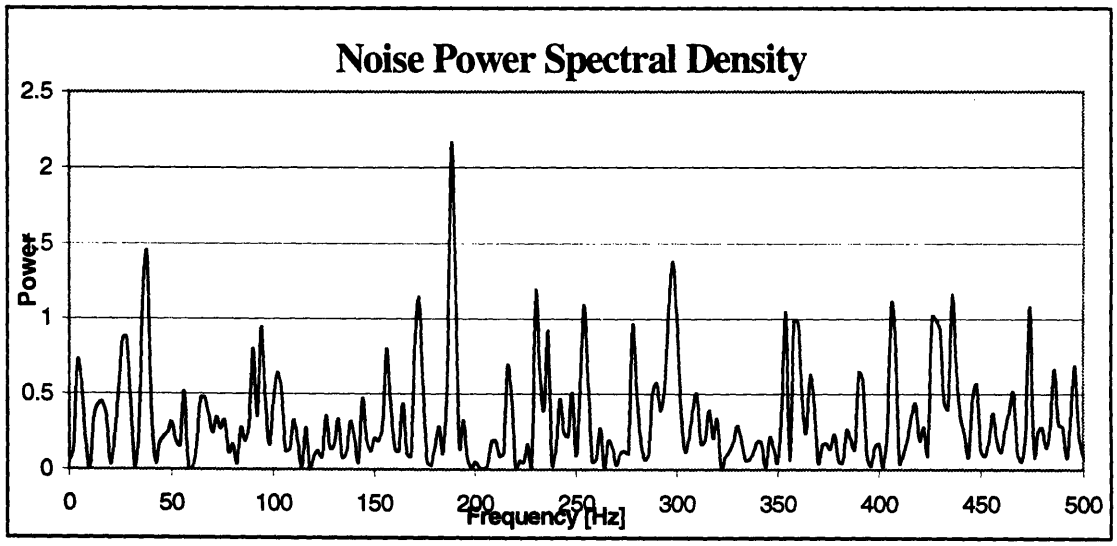
$$f(t) = \sum_{k=-\infty}^{\infty} a_k e^{j\omega_o kt} \;;\; \omega_o = \frac{2\pi}{T_o}$$
$$\text{where } a_k = \frac{1}{T}\int_{T_o} f(t)\, e^{-j\omega_o kt} dt \text{ and } e^{j\omega_0 kt} = \cos\omega t + j\sin\omega t. \qquad (1)$$

Each a_k is the corresponding amplitude of a component having frequency $\omega_o k$. Practically speaking, the Fast Fourier Transform (*FFT*) is the numerical routine commonly used to determine each a_k. Figure 3 shows the time trace (a) and power spectral density (b) of a noise signal obtained using the FFT. The FFT shows relatively uniform power over the

given frequency band and thus is referred to as *band-limited white noise.* The term "white noise" is used in analogy to white light, which is composed of all colors in the visual spectrum (*each of which can be described by a frequency*).



(a)



(b)

Figure 3. Noise time and frequency characteristics.

2.1.2 Period, Frequency, and Pitch

As previously mentioned, the period of a sound wave, T, is the time required to complete one full cycle. The frequency is calculated from the inverse of the period as follows,

$$f = \frac{1}{T} .$$

This is the fundamental parameter used in synthesizing sounds and can be easily measured; for example, with an oscilloscope. A sound is also defined by it's *pitch.* Unlike frequency, pitch is a subjective quantity requiring human subjects to evaluate it. Quite often, these two terms are interchanged, but it is the frequency that actually defines the pitch of a sound. Hence, if there is no clearly defined frequency, there is no clearly defined pitch. Therefore a tuning-fork tuned to a pitch of middle C will vibrate 261 times in one second, regardless of whether the sound is loud or soft. This is also true for all sounds with a frequency of 261 Hz, no matter how they are reproduced. For example, the hum of a diesel engine rotating at 261 rotations per second will also produce middle C. The pitch, or perceived frequency, is nonlinearly related to the actual frequency (*see Figure 4*). [6, pg. 66]. Humans tend to judge the pitch to be lower than the actual frequency after about 1000 Hz.

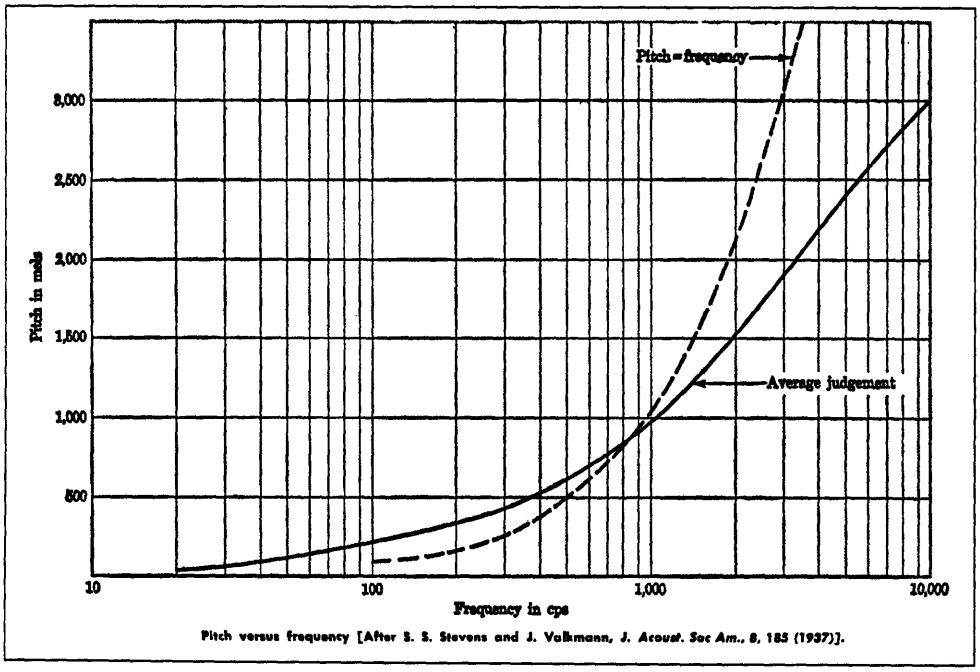


Figure 4. Pitch versus frequency.

2.1.3 Timbre

The last major sound characteristic is *timbre*. Simply stated, timbre differentiates one musical sound from another when the apparent pitch and loudness are the same. This is why we can distinguish between the sounds of a flute and a clarinet both playing the same note; they have different timbre. The harmonics of a sound play a major role in determining timbre. Complex sounds, unlike a pure tone, can be composed of many frequencies of varying intensity. In most cases, there is a dominant or *fundamental* frequency contributing most of the energy accompanied by less intense higher frequencies. The higher frequencies are called *harmonics* if they are an integral multiple of the fundamental. "Hemholtz proved that the timbre of a sound is determined by the proportions in which the various harmonics are heard in it." [6, pg. 71]

In addition to harmonic content, the sound envelope also affects timbre (*see Figure 5*). The envelope consists of three parameters: attack, sustain, and decay. The onset or *attack* of a sound determines the growth in loudness over time. "The characteristic tone of an oboe is due largely to its...attack." [6, pg. 71]

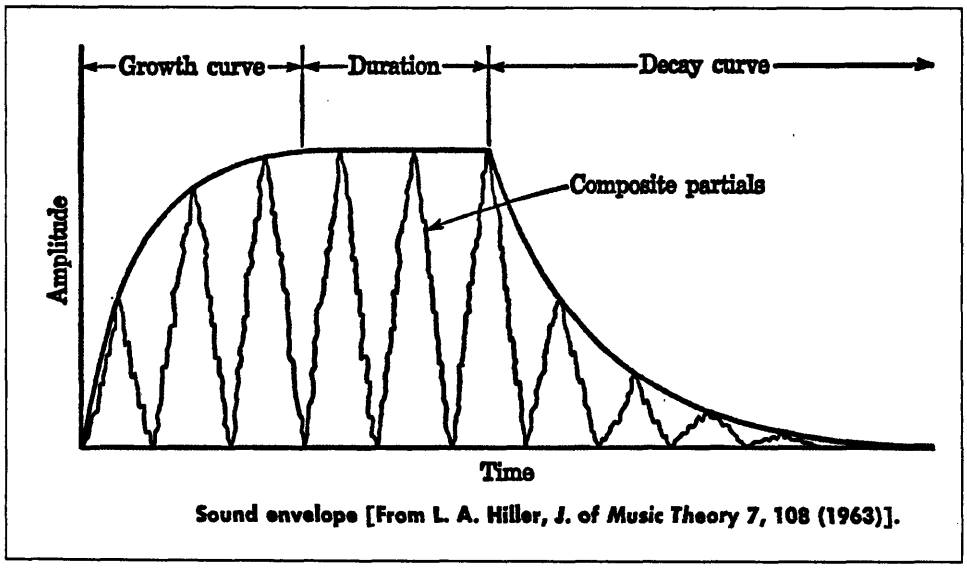


Figure 5. Sound Envelope.

The sustain time is the amount of time the sound remains in steady state. This time is critical to how the ear perceives the sound. Since the ear is a mechanically vibrating system, it too must reach a steady-state value in order to recognize a tone. "The minimum recognition time (duration threshold) is 4 ms for sine tones switched on suddenly." [10, pg. 111] A clicking sound will be heard if the duration is any shorter. Finally, the decay time is the transition from steady state to zero. This too is what allows one to distinguish between two sounds having the same steady-state timbre but different decay envelopes.

The next section will cover the hardware used to synthesize sounds. By using frequency modulation (*FM*), many sounds can be created each having different timbre and thus different sound envelopes.

2.2 Additive and Subtractive Synthesis

Additive and subtractive synthesis were the first analog methods used to reproduce sound. During additive synthesis, a complex sound is constructed from discrete frequencies corresponding to the desired harmonics. The building blocks come from three basic waveforms: (1) sine wave, (2) sawtooth, and (3) square wave. Each is generated by a voltage-controlled-oscillator. Using the Fourier Transform, it can be shown that each waveform has a unique frequency content. The square wave, for example, will have a frequency spectrum containing the fundamental frequency plus odd harmonics with amplitudes falling off at a ratio of $\frac{1}{3}$, $\frac{1}{5}$, $\frac{1}{7}$, etc. A sawtooth, on the other hand, contains both odd and even harmonics with exponentially delaying amplitudes. Any unwanted harmonics can be attenuated by appropriately filtering the output signal. To complete the synthesis, an amplitude modifier is placed in series with the filtered output to create the desired sound envelope as shown in Figure 6.

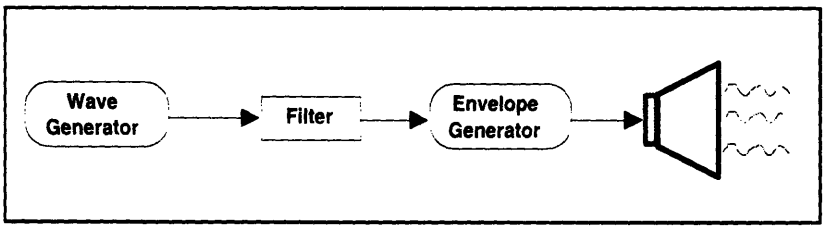


Figure 6. Additive synthesis block diagram.

As the name suggests, subtractive synthesis operates in the opposite sense of additive synthesis. In this case, noise, which contains many frequencies, is band-pass filtered to get only a desired frequency output. Ideally, white noise would be generated because it has uniform amplitude for all audible frequencies, but only band-limited white noise is realizable. To generate all the necessary harmonics, an array of parallel filters, each passing only a desired frequency, is setup and then summed at the output. An amplitude modifier can then modulate the output signal to produce the desired sound envelope (*see Figure 7*).

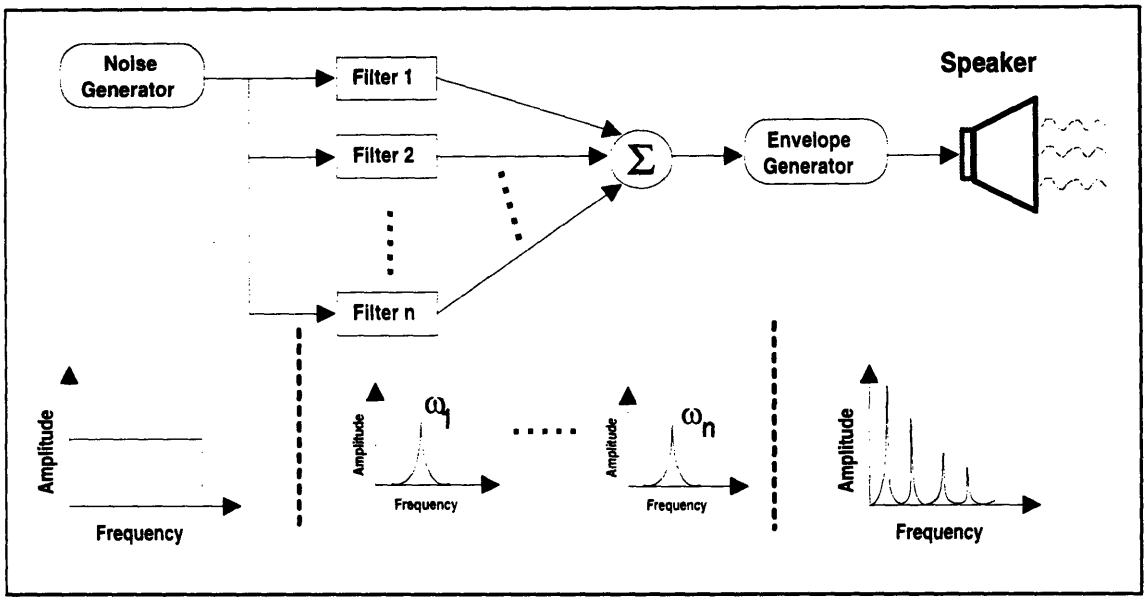


Figure 7. Subtractive synthesis block diagram.

2.3 Frequency Modulation

A third method of reproducing sound relies on *frequency modulation* (*FM*). By modulating the frequency of one waveform with another waveform, a rich spectrum of harmonics, not present in either of the two original waveforms, can be obtained. This scheme has two advantages over the previous two synthesis methods. First, FM is computationally superior. This method of adding harmonics is much faster than taking a signal and adding in its harmonics one at a time. Second, the parameters in FM are easy to change in real time thus allowing one to more easily imitate rapidly changing sounds. Figure 8 show a basic configuration for a simple FM system:

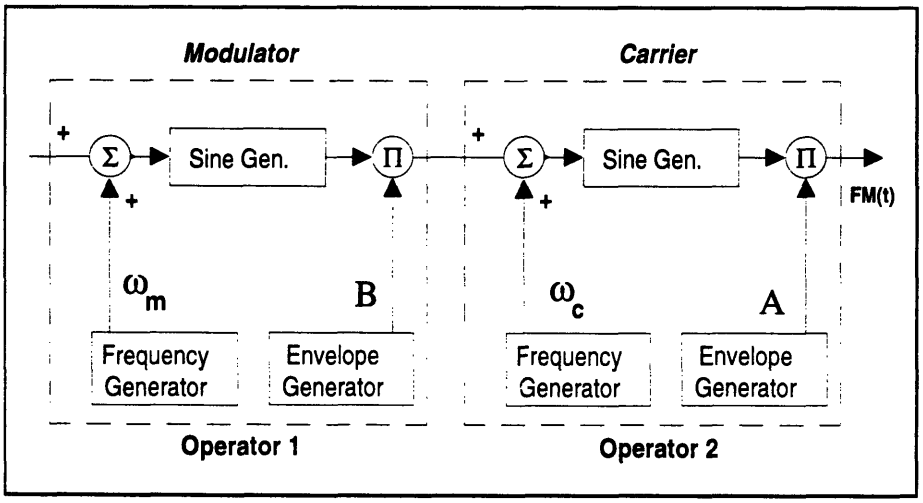


Figure 8. Simple 2-OP FM synthesizer.

The sine wave oscillators are called *operators* and perform three functions: (1) frequency generator, (2) envelope generator, and (3) sine wave generator. In this configuration, there is one *modulator* and one *carrier*. The arrangement of modulators and carriers is called an *algorithm*. The following equation expresses the algorithm shown in Figure 8:

$$FM(t) = A\sin(\omega_c t + B\sin\omega_m t). \quad (2)$$

The parameters A and B are the carrier and modulator amplitudes, respectively. The frequencies ω_c and ω_m are the carrier and modulator angular frequencies, respectively. Expanding equation (2), yields the following:

$$FM(t) = A[\sin\omega_c t\cos(B\sin\omega_m t) + \cos\omega_c t\sin(B\sin\omega_m t)]\ . \quad (3)$$

The two terms within this sum can be expressed as:

$$\sin(B\sin\omega_m t) = 2J_1(B)\sin\omega_m t + 2J_3(B)\sin 3\omega_m t + \ldots + 2J_{2n+1}(B)\sin(2n+1)\omega_m t + \ldots \quad (4)$$

$$\cos(B\sin\omega_m t) = J_0(B) + 2J_2(B)\cos 2\omega_m t + \ldots + 2J_{2n}(B)\cos 2n\omega_m t + \ldots \quad (5)$$

where $J_n(B)$ is a Bessel function of nth order [3, pg. 17]. Substituting (4) and (5) into (3) explicitly shows the harmonics:

$$\begin{aligned} FM(t) = A[&J_0(B)\sin\omega_c t + J_1(B)\{(\omega_c+\omega_m)t - \sin(\omega_c-\omega_m)t\} + \\ &+ J_2(B)\{(\omega_c+2\omega_m)t - \sin(\omega_c-2\omega_m)t\} \\ &+ J_3(B)\{(\omega_c+3\omega_m)t - \sin(\omega_c-3\omega_m)t\} + \ldots]. \end{aligned} \quad (6)$$

The key parameters that determine the harmonic frequency content are the carrier frequency and the ratio of carrier to modulator frequency. In addition, the amount of higher harmonics is determined by the amplitude of the modulator. For more information into the theory of Frequency Modulation, please see reference [3].

There are several companies that manufacture FM synthesis equipment. For this research project the SoundBlaster 16® (SB16) computer sound card is used to generate FM sounds. The selection criteria is based on performance, cost, and compatibility. To follow is a discussion of how to program the FM chip used by the SB16; the Yamaha OPL3 FM chip.

2.4 Programming the Yamaha OPL3 FM Chip

This section will cover only the essentials of programming the OPL3 FM chip used by the SB16. The objective is to provide enough information to get one started programming FM sounds. For more detailed information on all the card's functions, please see reference [11].

2.4.1 The Compiler

All the code presented in this document can be compiled using Borland C/C^{++} version 3.1 or higher. Some functions used in the code are Borland library functions and may not be available on other compilers.

2.4.2 Ports and Registers: Description and Usage

Because the OPL3 is a register-oriented chip, knowing how to access the registers is essential. "Registers are certain interfaces that are directly assigned to hardware. Commands that directly control hardware operation are placed in registers." [7, pg. 35] Each register has a unique hexadecimal address offset from the base I/O port address; base 16 numbers will be written with an "h" suffix. The base I/O port for the SB16 is set during installation or by jumpers on the card and can range from 220h-280h. This information is stored in the DOS environment variable "BLASTER" located in the *autoexec.bat* file. The function *detect_settings()* defined in the SMIX sound library (*see attached disk*) can be used to read and store the base I/O value. The OPL3 chip is accessed through ports 2x0h-2x3h, where "x" is a place holder for the 2nd number of the base I/O address (*i.e., 283h*). The following table indicates the function of each port address:

Table 1. SoundBlaster Port Identification.

PORT FUNCTION	BANK 0	BANK 1	PERMISSIONS
Address/Status	2x0h	2x2h	Read/Write
Data	2x1h	2x3h	Write Only

Writing register values to the Ports

The address port is used to "connect" to the requested register. After the request, the program must wait before writing data to the data port. To write a byte to a SB16 port, the following procedure should be used:

(1) Write the register address to the address port (*2x0h for bank0 or 2x2h for bank 1*).

(2) Wait at least 3 microseconds.

(3) Write the register value to the data port (*address port +1*).

The following code will execute the above tasks:

```
void timedelay(unsigned long clocks)
{
        unsigned long elapsed=0;
        unsigned int last, next, ncopy;

        outp(0x43, 0);
        last=inp(0x40);
        last=~((inp(0x40)<<8) + last);
        do
            {
              outp(0x43, 0);
              next=inp(0x40);
              ncopy=next=~((inp(0x40)<<8) + next);
              next-=last;
              elapsed+=next;
              last=ncopy;
            } while (elapsed<clocks);
}

void FM_Write_Output(unsigned port, int reg, int val)
{
        outp(port, reg);
        timedelay(8);          // delay about 3.3 microseconds
        outp(port+1, val);
        timedelay(55);         // delay about 23 microseconds
}
```

Register Layout

The OPL3 is separated into two register "banks". Each bank can generate nine independent 2-operator FM synthesizers referred to as *channels*. Table 2 gives the register set for the modulator (*OP Number 1*) and carrier (*OP Number 2*) of each channel.

For example, all the registers needed to program the modulator of channel 1 are 20h, 40h, 60h, 80h, and E0h. Registers A0h, B0h, and C0h affect both operators. Notice that the carrier register addresses are offset from the modulator registers by 3h. An additional 9 channels are available on bank 1 and have exactly the same register offsets; recall ports 2x2h and 2x3h control this bank. An example function that writes a value to register 20h (channel 1 modulator) of bank 0 is written as follows:

```
Void WriteRegValueBank0(int reg, int val)
{
        FM_Write_Output(baseio+0h, reg, val);
}
⋮
WriteRegValBank0(20h, 60h);  //write the value 60h to register 20h of bank 0
```

Table 2 . Register-to-Slot and Channel Relationship.

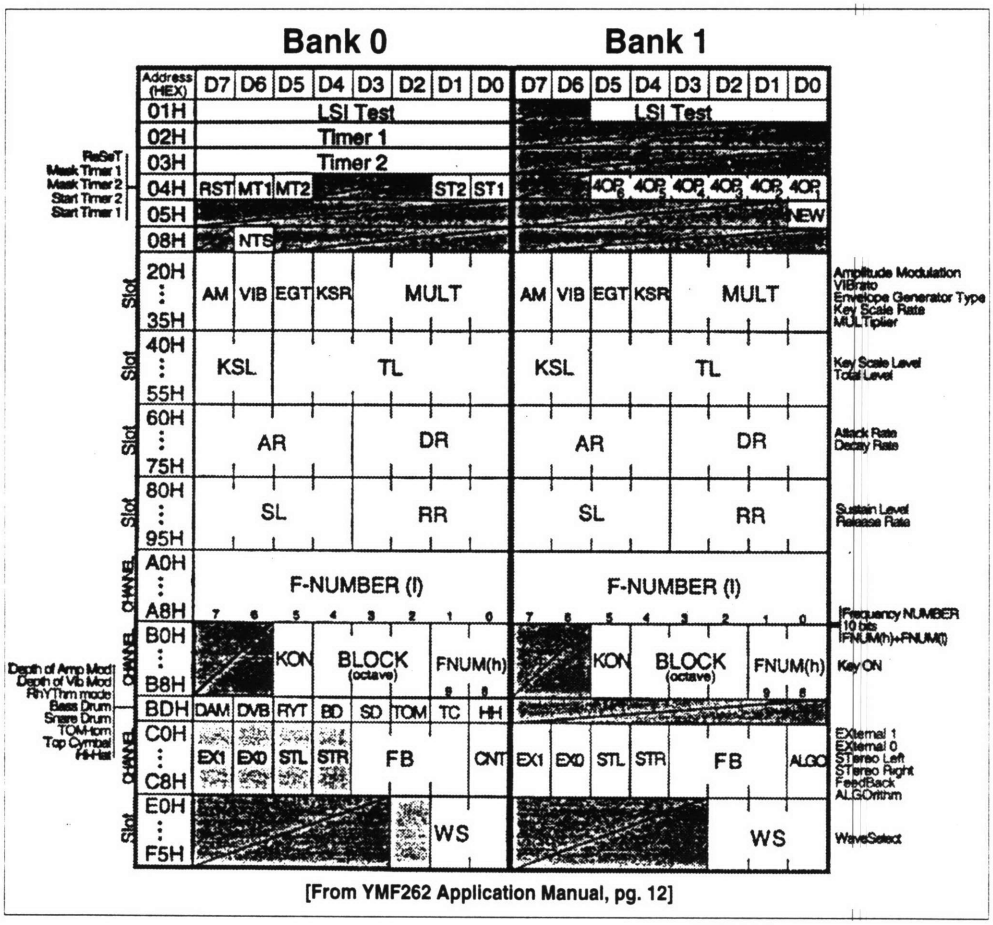
Channel Number (Decimal)	OP Number (Decimal)	Slot Number (Decimal)	REGISTER SET FOR SLOT (HEX)					Register Set for Channel (Hex)		
1	1	1	20	40	60	80	E0	A0	B0	C0
	2	4	23	43	63	83	E3			
2	1	2	21	41	61	81	E1	A1	B1	C1
	2	5	24	44	64	84	E4			
3	1	3	22	42	62	82	E2	A2	B2	C2
	2	6	25	45	65	85	E5			
4	1	7	28	48	68	88	E8	A3	B3	C3
	2	10	2B	4B	6B	8B	EB			
5	1	8	29	49	69	89	E9	A4	B4	C4
	2	11	2C	4C	6C	8C	EC			
6	1	9	2A	4A	6A	8A	EA	A5	B5	C5
	2	12	2D	4D	6D	8D	ED			
7	1	13	30	50	70	90	F0	A6	B6	C6
	2	16	33	53	73	93	F3			
8	1	14	31	51	71	91	F2	A7	B7	C7
	2	17	34	54	74	94	F4			
9	1	15	32	52	72	92	F2	A8	B8	C8
	2	18	35	55	75	95	F5			

[ref. YMF262 Application Manual, pg. 14]

2.4.3 Register Map for the FM Synthesizer

The register map for the FM chip used by the SoudBlaster®16 is shown in Table 3. These registers contain all the parameters that must be set in order to make an FM sound. This section will give the instrument table format used to store FM parameters and describe channel registers. Please refer to reference [11] for complete information on other registers.

Table 3. Register Address Map.



[From YMF262 Application Manual, pg. 12]

Instrument Table

The instrument table is an array used to store groups of FM parameters. Each row in the array defines a different FM sound. Sounds can be indexed according to their position in the array and easily loaded into the registers.

The function *Initialize_Sound_Timbre_2op_Mode()* defined in the high-speed train simulation code loads an FM sound from the instrument table in this manner (*see Appendix D*). Table 4 shows the format for a *row* in the table:

Table 4. Instrument Table Offset Format.

OFFSET (HEX)	DESCRIPTION
00	Modulator Sound Characteristics
01	Carrier Sound Characteristics Bit 7 : Amplitude Modulation (AM) Bit 6 : Vibrato (VIB) Bit 5 : Envelope Generator Type (EGT) Bit 4 : Key Scale Rate (KSR) Bit 3-0: Frequency Multiplier (MULT)
02	Modulator Scaling/Output Level
03	Carrier Scaling/Output Level Bit 7-6: Key Scale Level (KSL) Bit 5-0: Total Level (TL)
04	Modulator Attack/Decay
05	Carrier Attack/Decay Bit 7-4: Attack Rate (AR) Bit 3-0: Decay Rate (DR)
06	Modulator Sustain Level/Release Rate
07	Carrier Sustain Level/Release Rate Bit 7-4: Sustain Level (SL) Bit 3-0: Release Rate (RR)
08	Modulator Wave Select
09	Carrier Wave Select Bit 7-3: Clear Bit 2-0: Wave Select (WS)
0A	Feedback/Connection Bit 3-1: Feedback (FB) Bit 0 : Connection (CNT)
0B-0F	Reserved for Future Use

The abbreviations in parenthesis after each bit correspond to the labels in the register map. An example of using the instrument table to define two sounds is shown below.

```
/* Sample instrument table definition */
int inst[128][23] =        /*define the array dimensions */
{
/* (index #0: FM sound 1) */
{ 0x60, 0x60, 0x8E, 0x8D, 0xFF, 0xFF, 0x0F, 0x0F,
  0x00, 0x00, 0x0D, 0x00, 0x00, 0x00, 0x00, 0x00 },

/* (index #1: FM sound 2) */
{ 0x61, 0x70, 0x68, 0x00, 0xF2, 0x52, 0x0B, 0x0B,
  0x00, 0x00, 0x0A, 0x00, 0x00, 0x00, 0x00, 0x00 }
}
```

The numbers within the brackets ({...}) are hexadecimal. Using a two digit hex number conveniently breaks an 8 bit computer word into two groups of 4 bits. Here are some examples:

```
FFh = 1111 1111 binary
F0h = 1111 0000 binary
0Fh = 0000 1111 binary
60h = 0110 0000 binary
```

Breaking a computer word into high and low bits is convenient and used repeatedly throughout the code included in the appendices.

Frequency Control (Register A0h-A8h and B0h-B8h)

The frequency produced by the FM synthesizer is set by a 10-bit value (*F-NUMBER*) formed from 8 low bits (*F-NUM(l)*) in register A0h-A8h and 2 high bits (*F-NUM (h)*) in register B0h-B8h. The octave is determined by the 3-bit value (*BLOCK*) in register B0h-B8h. The FM sound for a given channel is on when the *KEYON* bit is 1 and off when 0.

REGISTER (HEX)	BITS							
	B7	B6	B5	B4	B3	B2	B1	B0
A0-A8	F-NUM (L)							
B0-B8			KEY ON	BLOCK			F-NUM (h)	
				2^2	2^1	2^0		

Equation 7 gives the F-NUM (*in decimal*) as a function of desired frequency *f* [Hz] and BLOCK number:

$$F-NUMBER=\frac{f*2^{(20-BLOCK)}}{50000} \quad . \quad (7)$$

For example, if a 277.2 Hz frequency is desired ($C^{\#}$) and BLOCK is 4, then

F-NUMBER=363d=0101101011b. Thus, F-NUM (l)=01 and F-NUM (h)=01101011b. If channel 1 is being programmed, the value 6Bh would be written to register A0h and 31h (*KEYON+BLOCK+F-NUM(h)*) would be written to register B0h.

Connection Algorithm, Feedback, and Left & Right Stereo (Register C0h-C8h)

The following table shows the function of each bit in the C0h-C8h register:

REGISTER (HEX)	BITS							
	B7	B6	B5	B4	B3	B2	B1	B0
C0-C8			STL	STR	Feedback			CNT
					2^2	2^1	2^0	

In 2-operator mode there are only two possible connection algorithms. The classical FM configuration is obtained by setting CNT to 0 (*see Figure 9 (a)*). This algorithm has self-modulation on operator 1, where the value of FB ranges from 0-7 according to bits 1-3.

If pure tones are desired, the value of CNT is set to 1 making the parallel connection shown in Figure 9(b).

When the SB16 is in OPL3-mode, bits 4 and 5 control the right (*STR*) and left (*STL*) audio output. Setting these bits high/low will turn the left or right audio output on/off.

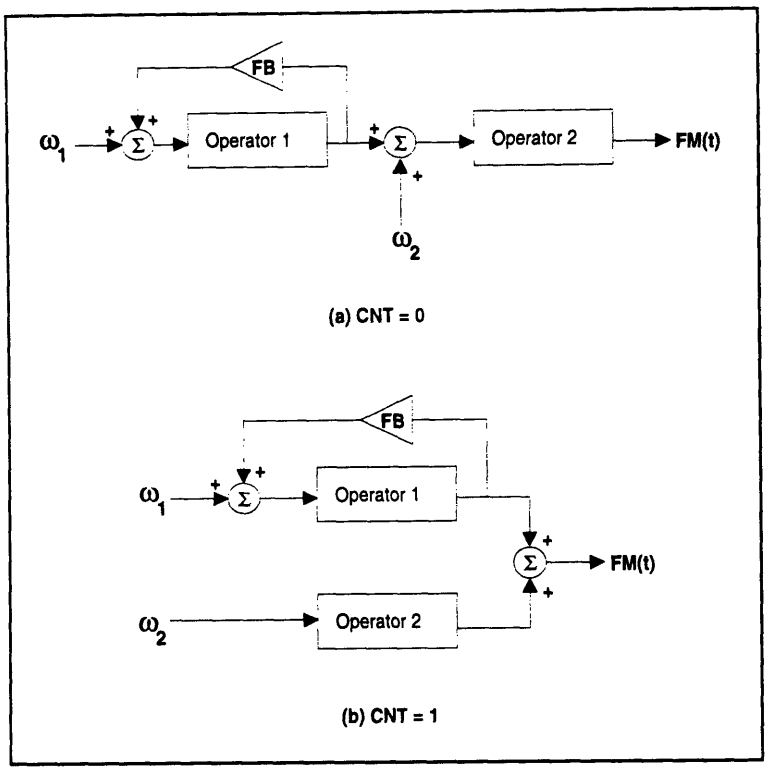


Figure 9. 2-Operator connection algorithms.

2.4.4 Initializing the SoundBlaster 16 Card

Before any FM can be programmed, the SB16 must be initialized. The following procedure initializes the sound card:

(1) Write the value 00h to register 01h of bank 0 to initialize the card.

(2) Write the value 01h to register 05h of bank 1 to set the card to OPL3 mode.

(3) Write the value 00h to register BDh of bank 0 to setup FM mode.

Refer to the function *InitializeFMsound()* in Appendix D.

2.4.5 Helpful Resources

Third party software exists to assist those not familiar with the effects of changing FM parameters. One such program, called *FMED 101*, is shareware and can be downloaded at http://www.cdrom-paradise.com/fmed.html. The shareware version of this program is included in the companion diskette. This program provides an interface that changes the

FM parameters during program execution so that one can experiment with different combinations. One can also do a search on the World Wide Web for "*SoundBlaster Utilities*" and find a variety of newsgroups and other shareware.

2.4.6 Sample Code

Appendix A and the companion diskette contain sample code that programs the SB16 FM chip. Borland C/C^{++} version 3.1 and higher will compile the code.

2.5 Digital Sound Software Library: SMIX

In cases where a complicated sound must be reproduced almost exactly, FM synthesis may not be the best approach. Playing a digitally sampled version can be used as an alternative, but not without a penalty. The main disadvantage of this method is that through sampling, the sound file must be played back with the recorded sampling rate. Otherwise the sound will be distorted. Sound recordings can also get quite large; for high sampling rates, a few seconds can exceed a megabyte, thus limiting the size and duration the sample.

Digital sampling involves discretizing a continuous time signal into quantization levels. The size of each level depends on how many bits are used by the analog-to-digital converter (ADC) and the voltage range. For example, a 12-bit ADC with a ±10V range will register 4.88 mV per level. Thus sampling accuracy is increased as the number of bits increases in the ADC. Stored digital sounds are converted to continuous time via a digital-to-analog converter (DAC). Aliasing and quantization noise should be considered when high quality recordings are desired.

The SoundBlaster 16 has the ability to play 8-bit mono or 16-bit stereo digitized sounds ranging from 5 kHz to 44 kHz in frequency. Programming the SB16 to play digital sounds is more complicated than generating FM sounds and requires knowledge programming interrupt service routines and the DMA controller 8237. For detailed information on DSP programming, see reference [2].

For those not familiar with this type of programming, the SMIX library, developed by Ethan Brodsky, provides a set of functions that initialize the DMA and allows digital recordings to be played back simultaneously with FM sounds.

The SMIX software allows 8 digital sounds to be played simultaneously at a maximum sampling rate of 44.1 kHz, changeable within the code. All digital sounds must be recorded at the same sampling rate and converted to a *raw* 8-bit unsigned format. Any file in the *wav* format can be converted to the *raw* format using the utility *wav2raw.exe.* Once all the desired sounds are in the raw format, they must be assembled into a single resource file, or sound library. To create a sound library, use the utility *sndlib.exe.* The user must assign a unique case-insensitive keyword to each sound. The keywords must then be assembled into a character array at the beginning of the program using SMIX as follows:

```
/* Character array of sound library keywords */
char *sound_key[NUMSOUNDS] =        /*NUMSOUNDS = n+1*/
     {
      "sndkey1",
      "sndkey2",
      :
      "sndkeyn"
     };
```

A desired sound is triggered by passing its array position to the function *start_sound()* within the main program. The sample program *smix.c* available on the included diskette should be used as a model to setup the rest of the program. The utility programs *wav2raw.exe* and *sndlib.exe* are also on the diskette.

2.6 Programming the SB16 Mixer Chip

The mixer chip within the SB16 contains a set of registers that control the master volume, FM sound volume, and digital sound volume. The register address port and data port are accessed through 2x4h and 2x5h, respectively. Table 5 shows the mixer chip register map.

Table 5. Mixer Chip Register Map

REGISTER OFFSET (HEX)	BITS							
	B7	B6	B5	B4	B3	B2	B1	B0
04	Digital Volume LEFT				Digital Volume RIGHT			
22	Master Volume LEFT				Master Volume RIGHT			
26	FM Volume LEFT				FM Volume RIGHT			

Each register has a 4-bit left and right side speaker volume control ranging from 0-15. If a 2 digit hex number is written to the register, the least significant digit will correspond to the lower 4 bits (*right side*) and the most significant digit will correspond to the upper 4 bits (*left side*).

Writing register values to the Ports

To write a value to the mixer chip register, use following the procedure:

(1) Write the register address to the address port 2x4h.

(2) Wait at least 3 microseconds.

(3) Write the register value to the data port 2x5h.

For example, to set the FM volume maximum on the left and 9 on the right, the value F9h should be written to register 26h. The function *Mixer_Control()* in Appendix D can be used for this purpose.

Chapter 3: COMMUNICATION

3.1 Networking

High-fidelity simulation requires considerable processing power for performing real-time dynamics and graphical calculations. Therefore, sound effects should be generated on an external system (*i.e. a PC*), thus minimizing computational overhead on the main processor. As a result, the external system must be networked with the workstation in order to obtain user input and the state of the simulation. The sound effects are then coordinated according to this information. Fast network communication is necessary to allow the sound system to be responsive to changes in simulation state, thus minimizing time lag. Ethernet is one networking standard that provides high transfer rates and reliable transmission.

Hardware and Software Requirements

The only hardware required to setup a network link between a PC and Unix workstation is an ethernet network adapter for the PC. Unix workstations are generally network-ready. If a network already exists, consult with the network administrator to select the card most compatible with the existing system. To use the communication software provided with this document, Novell LAN Workplace for DOS® must be purchased and installed on the PC. Other platforms, such as Artisoft's LANtastic or Novell Netware, will **not** work because a developer's kit library specific to LAN Workplace is used to develop the communication software. Note that this software is written in C++, so special linking of C and C++ code may be required.

The following section will briefly explain some networking basics and will provide the programmer information on how to use the ethernet software.

3.1.1 Software

The networking software in Appendix C is designed to setup a bi-directional ethernet link between a client and multiple host computers (*see Figure 10*). Data is transmitted through the networking cable (*co-axial, 10BaseT, etc*) and stored in a buffer located within the network adapter. The network adapter can be viewed as a black box that collects the data packets as a bit stream and then sorts the variables in a format defined by the programmer.

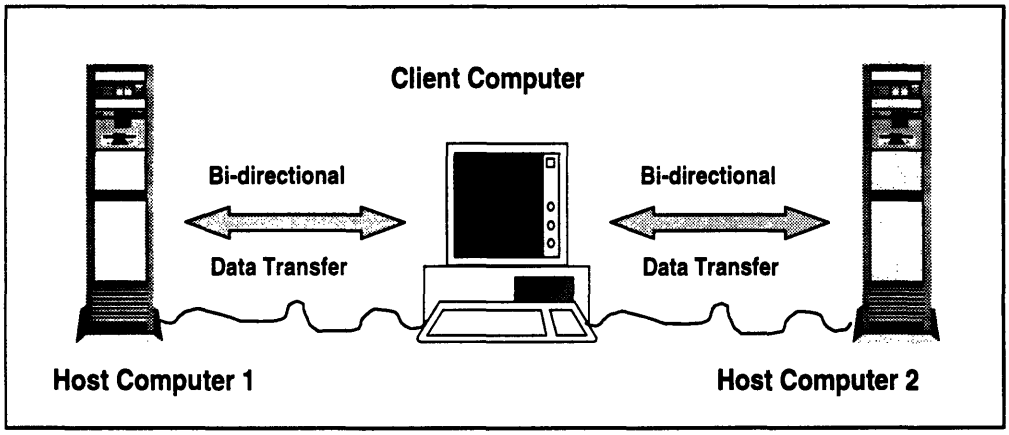


Figure 10. Client and multiple host network configuration.

Hand-Shaking

The manner in which data is exchanged between the client and host is extremely important. Consider the case where the main loop in the client program is twice as fast as the main loop of the host program. If both the client and host send data to the buffer indiscriminately, data will accumulate within the buffer because the host simply can not read the data stream fast enough. At some point, the buffer will reach it's maximum storage capacity and overflow. This is shown schematically in Figure 11.

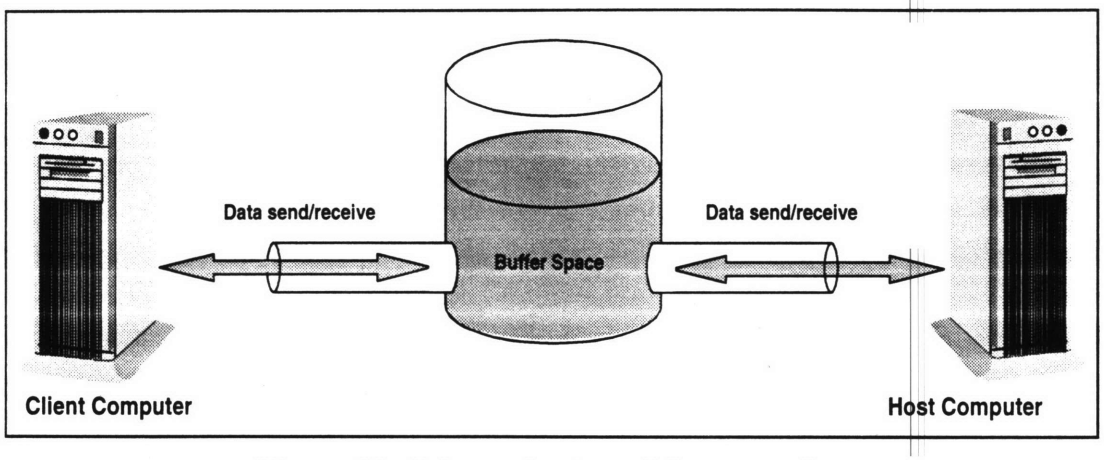


Figure 11. Schematic view of data transfer.

Buffer overflows can be prevented by operating the communication in a hand-shaking mode. In this way, the host sends data only after receiving data from the client. As a result, transmission occurs only when both the client and host are checking the network at the same time. Before the client writes data to the buffer, it checks to see if the host is ready to receive. It will wait for a maximum of 1 second before aborting the send. If the host checks the network within this 1 second window, a data exchange will occur.

Synchronization

As explained above, during hand-shaking mode data transfer occurs only when the client and host are checking the network simultaneously. Hence, the highest data throughput happens only when the host and client are exactly synchronized. Exact synchronization can happen only if the main control loops in the client and host programs have the same frequency. In most cases, however, this will not be the case due to differences in processor speeds and computational overhead. It is therefore wasteful in terms of processing time for the faster computer to check the network at the end or beginning of its loop. It is the programmer's responsibility to measure the frequency of each loop and adjust the bandwidth accordingly.

3.1.2 Sample Code

The easiest way to learn how to implement the ethernet code is by following the demo programs. It is assumed that the user has knowledge of using projects and makefiles. Appendix C and the attached disk provide C++ code demonstrating both single and multiple host communication with a PC. The PC client code, located in */network/client* on the disk, contains the project files (*.ide*) to compile the code under Borland C/C++ version 4.5. The SGI host code located in */network/host* contains the make files used to compile the code on a UNIX system.

The low-level communication functions are defined in *netclint.cpp* and *unixnet.cpp* for the PC and SGI workstation, respectively. The user should not need to alter these files.

Program Notes

Because the client and host are hand-shaking, it is the responsibility of the client program to initiate the data transfer. The host program must be started **before** the client program. This will open a socket and establish the link. Also, the definitions of *ECHO_PORT* and *BUFFER_SIZE* must be the same in both the client and host programs. If multiple host workstations are being used, the echo port defined within each host program must match the definition in the client program.

Data Transfer

The core functions are the data transfer functions defined in the client and host programs. The number and type of arguments should agree with each other. For instance, if the client computer is expecting two variables of type *float* and sending a variable of type *int*, the function declaration should be as follows:

```
CLIENT PROGRAM:
        void NetSendRecv( NetClient &enet, int dataout, float *data1in, float *data2in)
        {
          if(enet.iswriteready()) {
                sprintf(buffer_send,"%d",dataout); //send integer variable
                enet.send((char *)buffer_send, BUFFER_SIZE); //Data sent out of the pc
          }

          if(enet.isreadready()) {
```

```
            len=enet.recv((char *)buffer_recv,BUFFER_SIZE);
            sscanf(buffer_recv,"%f %f,data1in,data2in); // float data received from host
            if(len<=0) {
                        printf("\n zero receiving length");
                        exit(1);
            }
      }
      else {
            printf("\n not connected: read not ready \n");
      }

}
```

The host program must be modified in a similar manner to send two variables of type *float* and to accept a variable of type *int*:

HOST PROGRAM:

```
int networkSendRecv(float data1out,float data2out,int *datain)
{
   int rc=-1;

   sprintf(buf_send,"%3.2f %3.2f",data1out, data2out);
   strcpy(buf_recv, "N");
   if ( netserv.isconnected() ) {
     rc = netserv.recv(buf_recv,BUFFER_SIZE);
     if ( rc > 0 ) {
        sscanf(buf_recv,"%d",datain);  /*Scan in only if data is transfered!*/
        netserv.send(buf_send,BUFFER_SIZE);
     }
     else sprintf(buf_recv," Rc = %d \n", rc);
   }
   return rc;
}
```

Host IP Address

The host internet protocol (IP) address is the number used to uniquely identify the workstation on the network. The C^{++} class *NetClient* defines the low-level functions that link the client to a host. Each object of type *NetClient* declared within the main program creates a new host connection. The class has been written such that all objects of this type are initialized with the host IP address. Therefore, the host IP can be changed only by recompiling the code. If, however, there is only one host, the class *HostIP* can be used to change the host IP without having to recompile the program. This is useful when the client PC needs to link with either host 1 or host 2, but not both at the same time.

The object *HostIP* defined in *netclint.cpp* will allow the user to enter the host IP address on the command-line using the following syntax:

```
C:\ executable_-ipaddress
```

where "executable" is the name of the client networking program and the underscore (_) represents a space afterwards. The value *ipaddress* is either in the format *a.b.c.d*, where *a*, *b*, *c*, and *d* are numbers ranging from 0-255, inclusive or is the host name resolvable on the local network. If the command line argument is left off or if it can not be resolved by the network, the user is prompted a second time. The client program would use the following network initialization sequence: (*also see onehost.cpp*)

Within Client Main Program:

```
/*Create an object of type HostIP class initialized with the first string after executable */
 HostIP ipaddress(argv[1])
/*Create an object of type NetClient Class and initialize it with the IP address obtained by
ipaddress */
NetClient enet(ipaddress.returnIP());
```

Note that each host linked with this program will have to have the same echo port defined in the client program.

Chapter 4: HIGH-SPEED TRAIN SIMULATION

4.1 Background

For a number of years the Federal Rail Association (FRA) has contracted the Volpe National Transportation Systems Center to research improvements in locomotive control by looking at automation and information aiding. In order to perform human factors experiments, Dr. Edward Lanzilotta and Dr. Shumei Askey (*HMSL alumni 1995*) developed the high-speed rail simulator shown in Figure 12.

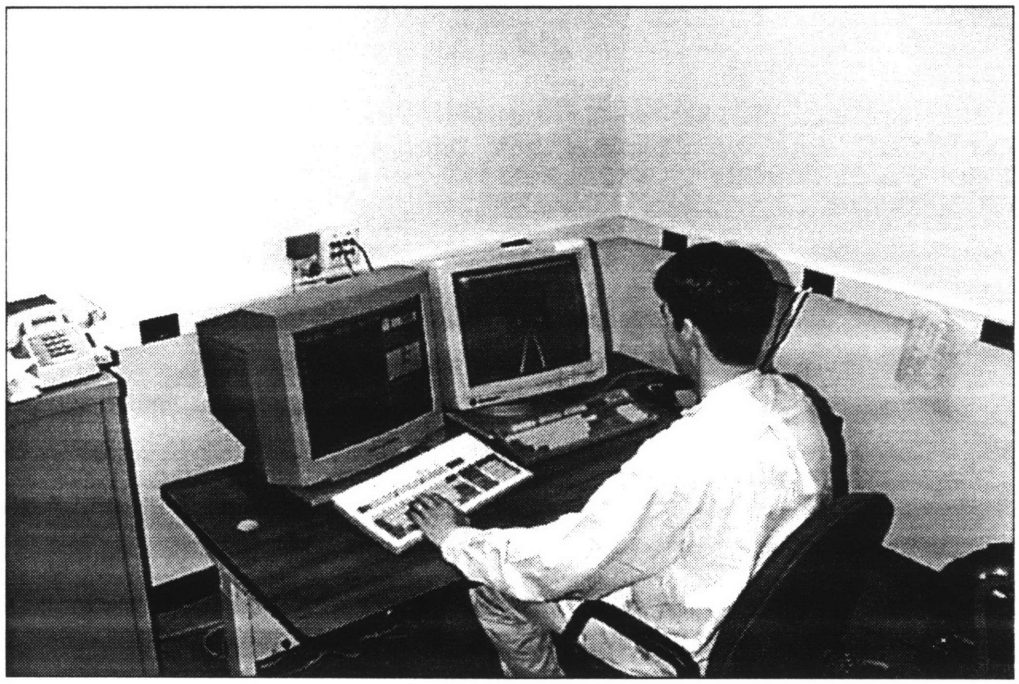


Figure 12. First train simulator setup.

The simulation uses three high-performance Silicon Graphics workstations. One is dedicated to calculating the train dynamics and producing the out-the-window (OTW) view. Another is used to create a dashboard/preview display and the third acts as central traffic control (CTC). Figure 12 shows the experimental setup used to collect human

factors data. The monitor on the left is the dashboard and the monitor on the right is the OTW view. The subject provides input through the keyboard and throttle joystick. Experiments, typically lasting 3 hours, would have the subject drive the train from one station to the next while recording operator responses to various train failures. Because of the length of each experiment, the project supervisors decided to pursue adding more fidelity to the simulation, thereby increasing situational awareness and vigilance.

This section will apply the information provided in Chapters 1-3 to the Volpe Center's train simulator. The external and internal train cab sounds are created on an external Pentium PC using a SoundBlaster 16 sound card and an amplified audio system. The networking software detailed in Chapter 3 is used to link this PC to the local network of SGI workstations. In addition, the physical interface is improved by creating external hardwired controls, a cabin enclosure, and a vibrating chair.

4.1.1 Desired Sounds

Increased fidelity in the simulator comes mainly by adding sound effects. The main external or environmental sounds are track, engine, and braking noises. These sounds are auxiliary sensory inputs that help keep the subject aware of the train state. There are also internal train cab sounds that can be added to increase realism. These are the dead-man alerter, speeding indicator, bell, and train horn sounds.

4.1.2 Physical Enhancements

In addition to adding sound, a preliminary cabin structure enclosure has also been constructed.. The structure isolates the subject from the large room and supports the dashboard display. Also, the user interface has been improved. This is done by replacing the keyboard interface with external control boxes. As a final (and experimental) enhancement, a 15" speaker driver is used to actuate the subject's seat. This is used to simulate the rumbling of the train cab .

4.2 FM Synthesis of Engine Throttle

One strategy in recreating the engine sound is to determine the dominant frequency that changes according to power level. This will correspond to the changing pitch the operator hears as the power level is varied. FM synthesis can easily generate this frequency with the benefit of providing a smooth transition between power levels. The rest of the complex spectrum can be played in the background digitally, thereby recreating the entire sound.

Digital Filtering

Separating the frequency components requires the use and knowledge of digital filtering. Therefore, issues of sampling and signal aliasing should be addressed. The goal is to extract the frequency band that is common to all the power levels and play them back as a separate group.

Finite-Impulse-Response (FIR) filters are commonly used and have the following pulse transfer function:

$$H(q) = b_0 + b_1 q^{-1} + b_2 q^{-2} + \cdots + b_n q^{1-n} \quad (8)$$

where n is the order of the filter. The following n^{th} order difference equation is obtained from equation 8 by applying the shift operator, $q^n f(k) = f(k+n)$:

$$y(k) = b_0 x(k) + b_1 x(k-1) + b_2 x(k-2) + \cdots + b_n x(k-(1-n)) \ . \quad (9)$$

Equation 9 is a robust non-recursive filter but will require almost 10 times as many terms as a recursive type to achieve the same high frequency attenuation. If the set of coefficients are chosen symmetrically (*i.e.* $b_k = b_{(n-k)}$), the filter will have linear phase. What this property means in physical terms, is that any signal that goes through the filter will be time delayed, but undistorted. The coefficients can be easily obtained by using the *FIR1* command in Matlab®.

Frequency Spectrum Analysis

As the engine throttle increases, one can hear an increasing frequency. Physically, this corresponds to harmonics of the diesel engine pistons as the speed changes. By recording various power levels and analyzing the frequency spectrum, the dominant frequency components can be determined. Figures 13 and 14 compare the unfiltered power spectral density for 4 power levels recorded on an Amtrack commuter train. The throttle lever for this train has a total of eight running speeds.

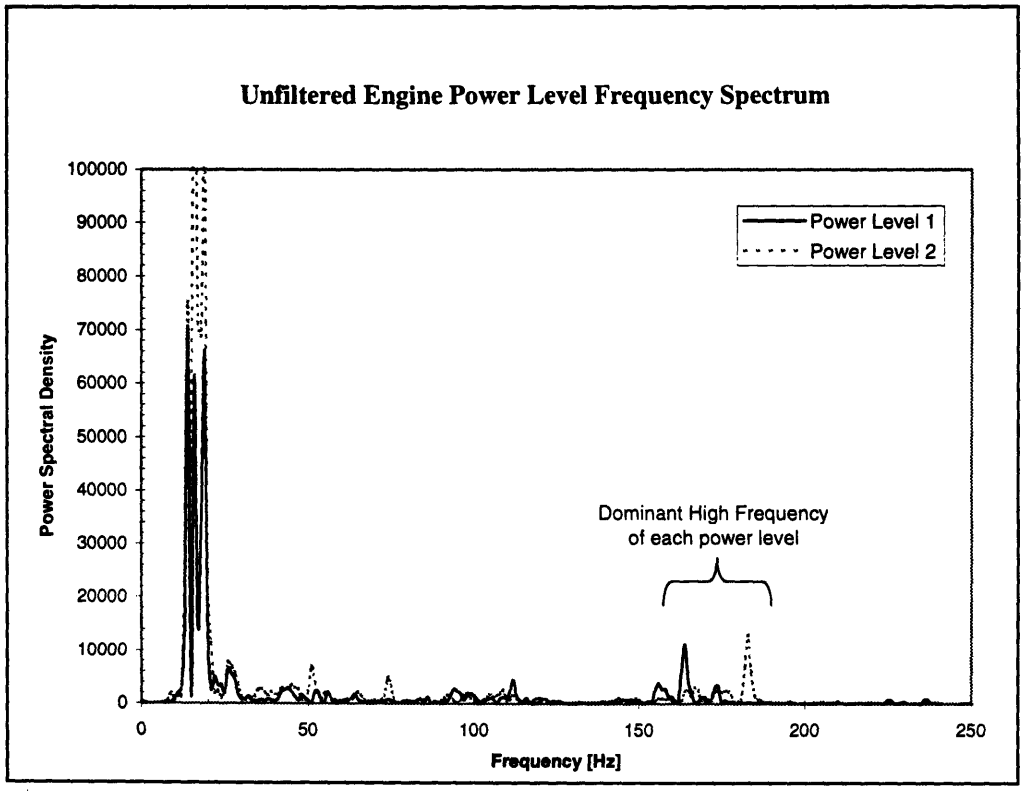


Figure 13. Power spectral density of engine power levels 1 and 2.

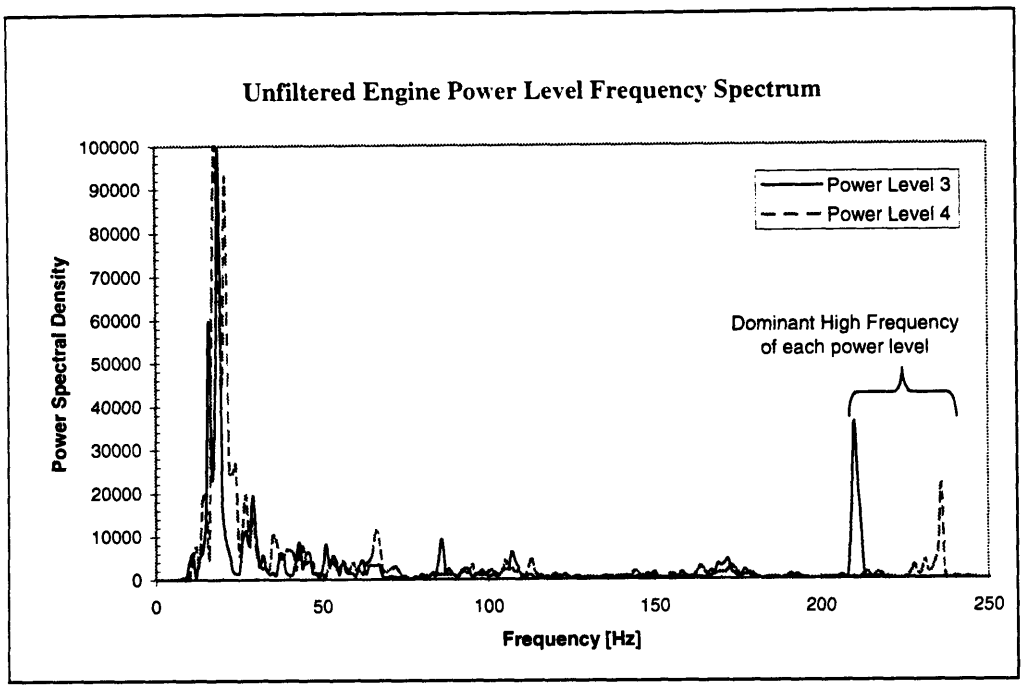


Figure 14. Power spectral density of train engine power levels 4 and 5.

Figures 13 and 14 show that all 4 power levels have a similar cluster of frequencies below 100 Hz and distinctive frequencies above 100 Hz. The low frequency cluster most likely corresponds to mechanical resonance.

The high-pass FIR filter show in Figure 15 is applied to each unfiltered power level recording to determine whether frequencies higher than 112 Hz correspond to the pitch changes heard as the power varies. As expected, each resulting spectrum shows a dominant frequency (*see Figure 16*). One can hear the pitch increase with increasing power level by listening to the filtered recording successively.

Thus, it is reasonable to extract the low frequency band that is common to all power levels using a low-pass FIR filter (*the inverse of Figure 15*) and play them continuously in the background.

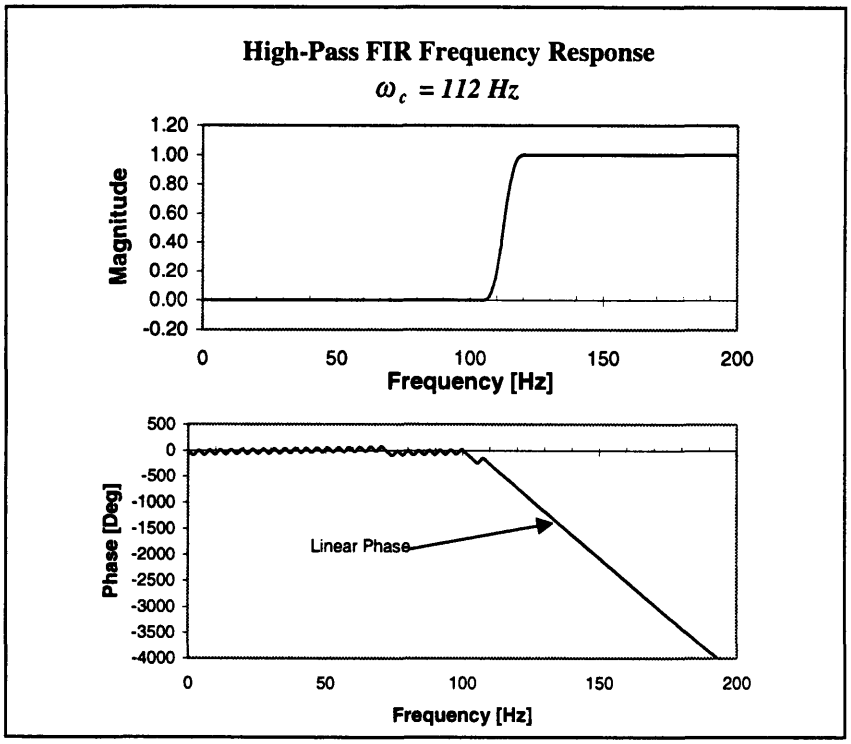


Figure 15. Digital FIR filter frequency response.

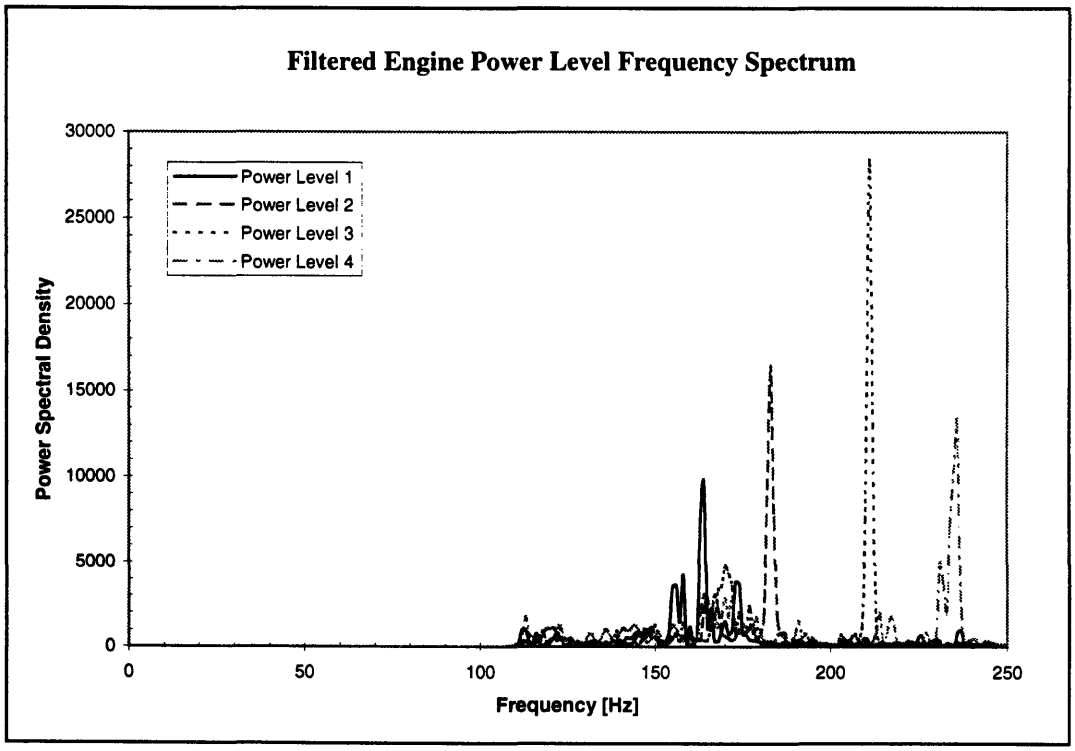


Figure 16. High-pass filtered power levels.

4.2.1 Engine Instrument Parameters

Having identified the frequency range at which the FM generated engine sound should play, an instrument file must be generated and written to the FM chip register set (*see Table 4*). The engine instrument parameters are defined as follows:

Table 6. Engine Noise Instrument Parameters.

OFFSET (HEX)	00	01	02	03	04	05	06	07	08	09	0A
ENGINE FM PARAMETERS (HEX)	61	70	68	00	F2	52	0B	0B	00	00	0A

A somewhat heuristic approach is taken in determining all the parameters, but some are easily determined by listening to the actual sound; like adding vibrato to simulate an undulating frequency.

4.3 Digital Playback of Track Sound

The main feedback of train speed is provided by the track noise created as the train wheels pass over connecting track rails. If these connection points occur at equally spaced intervals, then the rate of track noise instances (*a "click-clack" sound*) will be a function of the train speed. Therefore it is necessary to vary the digitally recorded track noise according to the speed calculated by the simulator.

One approach is to play one digitally recorded click-clack sound in a loop continuously. The sampling frequency can be modulated to give the sensation of changing speed. The disadvantage here is that other digital sounds played during the same time will be distorted because they are not playing at their recorded sampling frequency. For this simulator, other digitally recorded sounds, such as the train horn and air brakes, must be played throughout the simulation thus eliminating this option. An alternative approach using only one sampling frequency would be to discretize a single click-clack sound and then synchronize the playback. This approach is discussed below.

4.3.1 Decomposition

The decomposition of track noise is accomplished with the use of a sound wave editor. This type of program allows the user to play and edit desired section of a total sound recording. Using this program, one representative track click-clack sound is extracted from an actual digital recording (*see Figure 17*).

The time trace in Figure 17 is then broken up into 4 discrete components as shown in Figure 18. Each component has a time duration of Δt_i therefore,

$$t_{total} = \sum_{i=1}^{4} \Delta t_i$$

where t_{total} is the total time required to play all the components sequentially.

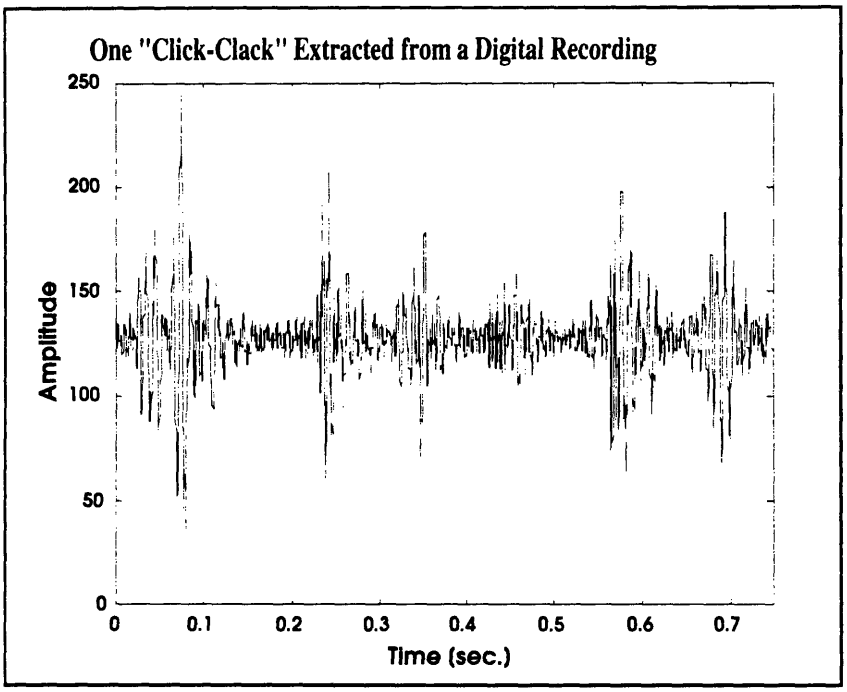


Figure 17. Track noise time trace.

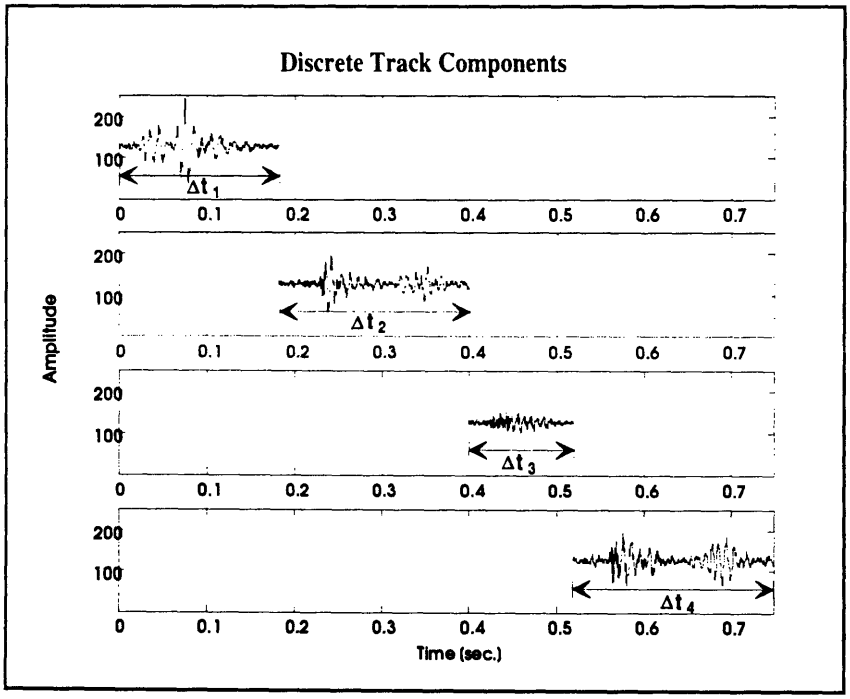


Figure 18. Full track noise broken into 4 discrete parts.

4.3.2 Synchronized Playback

The function *start_sound()* of the SMIX library is designed to play up to 8 digital sounds simultaneously in the background. As a result, a triggering scheme is required to playback the 4 track parts according to the train speed. This is done by adding a variable time delay between each track component and between each 4 component group. The following time line illustrates this scheme:

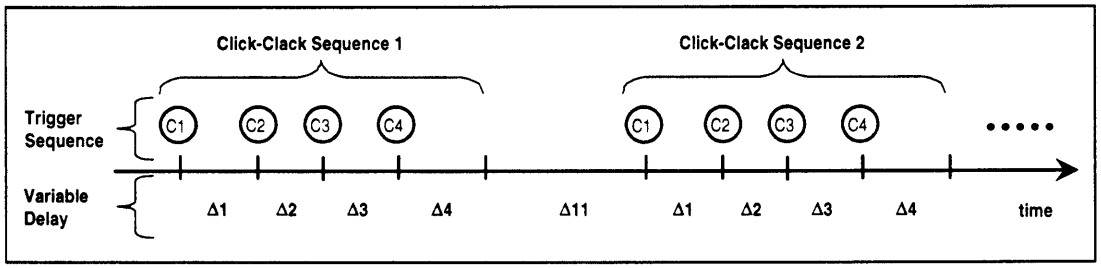


Figure 19. Track noise trigger sequence.

According to Figure 19, each component is followed by a variable time delay (Δ_i), which is calculated as a function of the current train speed. For simplicity, a linear interpolation is used as show in the following figure:

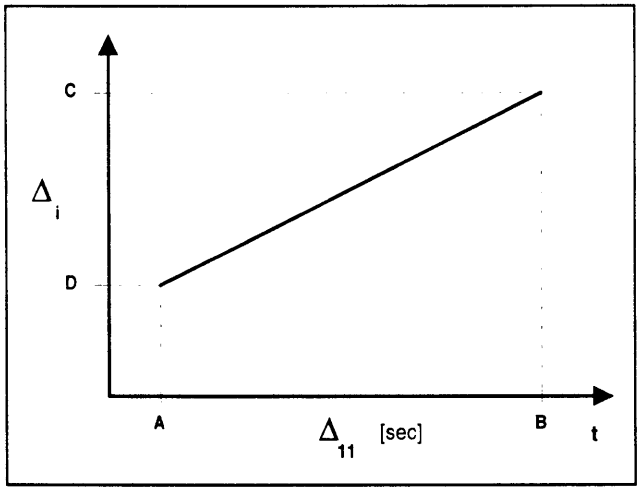


Figure 20. Linear interpolation for track component delays.

The following equation is used to calculate the delays:

$$\Delta_i = \frac{C-D}{B-A}(\Delta_{11} - A) + D \quad (10)$$

where $\Delta_{11} \propto \frac{1}{speed}$ and the values of A, B, and D are set in software. The value of C is equal to the total time required to playback the i^{th} component. Thus, as the speed increases, Δ_{11} decreases and the time delay between each component decreases.

4.4 Other Sounds

In addition to the engine and track noise, other sounds have been added to the simulation to increase realism. The following section will briefly discuss the importance of these other sounds.

4.4.1 Bell

The bell sound is FM synthesized using the following instrument parameters:

Table 7. Bell Instrument Parameters.

OFFSET (HEX)	00	01	02	03	04	05	06	07	08	09	0A
BELL FM PARAMETERS (HEX)	07	12	4F	00	F2	F2	60	72	00	00	08

The parameters were obtained from a predefined SoundBlaster instrument (SBI) file provided by Creative Labs. The bell is important because real train operators must ring the bell when arriving and departing from a station.

4.4.2 Dead-man Alerter

The dead-man alerter is a runaway train prevention system used in locomotives. The alerter goes off if the train operator does nothing for 45 seconds. The actual siren sound used on locomotives was not recorded and is not used with this simulator. Instead, a similar sound is played in a loop until the subject presses the dead-man switch.

4.4.3 Speeding Indicator

Another safety mechanism used is a speeding warning. Two cues are used to get the subject's attention: an audible warning and a flashing LED lamp. The digitally sampled warning is not the same sound used on real locomotives.

4.4.4 Brake Steam and Screeching

Braking sound effects proved to be an important audible cue of speed reduction and speed control. For this simulator, both power and braking are commanded through one joystick. The joystick position is determined according to the value sampled by an analog-to-digital converter and ranges between ±1.0.

Power increases from idle to full throttle if the joystick value is greater than zero. Braking occurs when a negative value is registered. Without an audible cue, it is difficult to know whether or not the joystick is near the zero point. Thus, the sound generation system plays a digital air brake sound every time the joystick makes a transition from power to braking. In addition, a metal-on-metal screeching sound, which has a programmable time duration, is also played. This added sound makes the subject aware of active braking.

4.4.5 Digital Train Horn

In addition to the breaking sound effects, a real train horn sound has been digitally sampled. Train operators must blow the horn when approaching grade crossings. The user can press one of the control box buttons to play the horn throughout the simulation.

4.4.6 How to Add New Sounds

The digital sounds described above have been added as outlined in §2.4. For sample code, refer to *smix.c* located on the attached disk.

4.5 PC Sound Generation Software

The sound effects are controlled by a program running on a Pentium 133Mhz with 32 megs of RAM and a network card. This code is a mixture of C and C^{++} and is compiled for DOS with Borland C/C^{++} version 4.5. The project file controls linking all the source code and generates the executable and object files. The user should make sure the source directories set under the *project options* menu are correct. The following table summarizes the relevant files of the project:

Table 8. PC Sound Generation Software Program Files.

SOURCE	INCLUDE	LIBRARY
DIGITAL PLAYBACK		
smix.c detect.c xms.c	smix.h detect.h xms.h	
USER INTERFACE		
display.cpp	display.h	
NETWORK SOFTWARE		
netclint.cpp	netclint.h	tllbsock.lib obsolete.lib
MAIN PROGRAM		
trainsnd.cpp		

The above files are also located on the companion diskette.

4.5.1 Program Flow

Figure 21 show the program flowchart for *trainsnd.cpp*. Upon execution, the network, digital sound, FM sound, and display graphics are initialized. The program time base is also started at this point. Once the idle engine sound is generated, the program enters the main control loop, which is performed until the program is terminated by pressing the "q" key. If the train velocity is below 10 m/s, the network will be checked between each track component. Next, any user inputs from the keyboard or network are checked. These can be commands to trigger the horn, bell, or warning sounds. If the power level has changed from the current level, the FM engine frequency is also increased or decreased accordingly. Before the i^{th} track component is played, the network is checked once more if velocity is greater than 10 m/s and $i = 5$. This essentially reduces the rate at which the client checks the network in order to match the slower host workstation (*see §3.0 for details on synchronization*). Before repeating the command loop, the simulation time is checked to see if the update time, set by the programmer, has been exceeded. If so, all the delays are recalculated according to the linear interpolation described in §4.3.2.

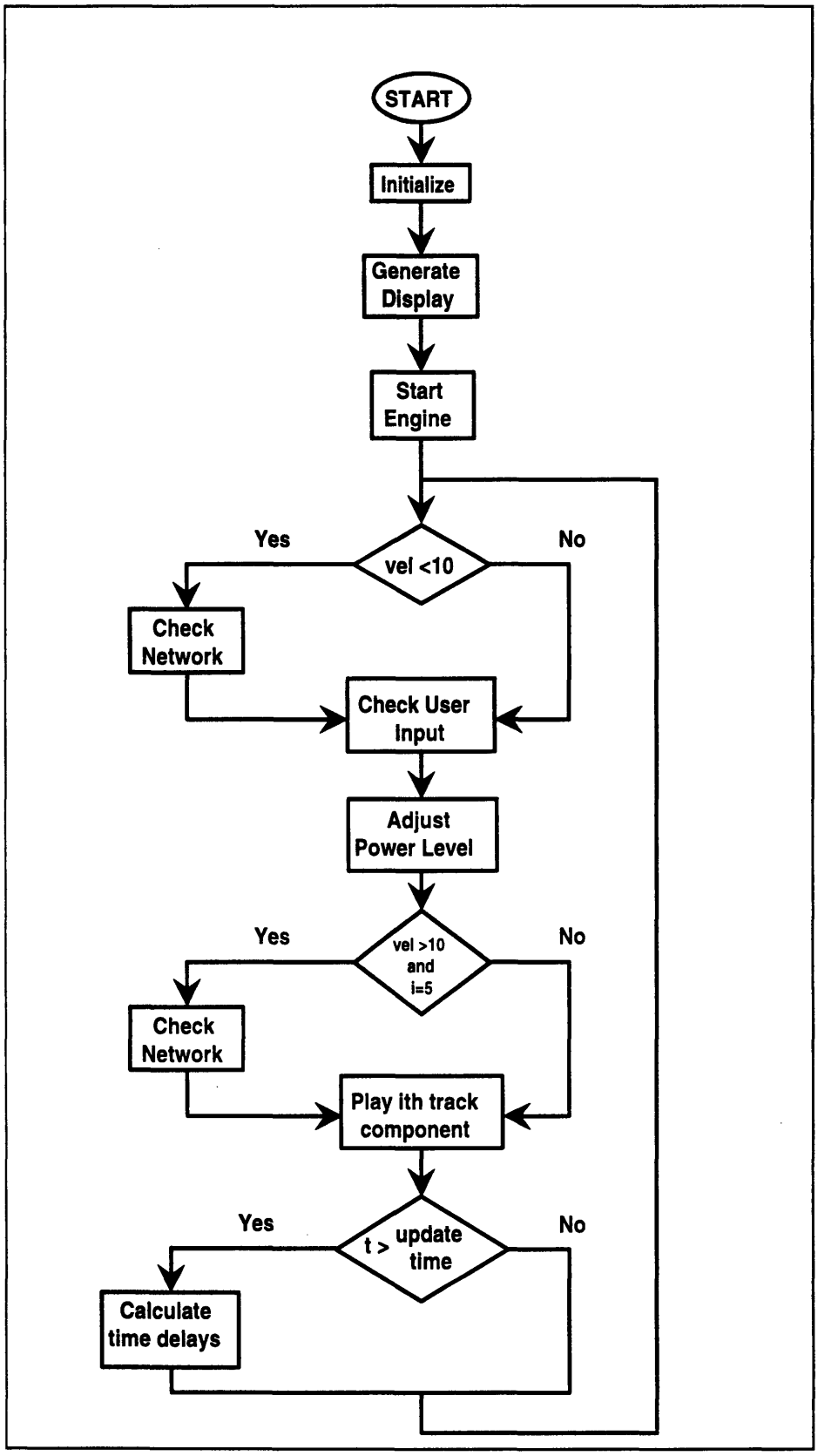


Figure 21. Main program flow chart.

4.5.2 Some FM Function Notes

The main program brings together the digital playback functions defined in *smix.c*, the networking functions defined in *netclint.cpp*, and user interface functions defined in *display.cpp*. A majority of functions defined within the main program itself pertain to FM. Even though the program is commented, this section is devoted to explaining some of the functions in *trainsnd.cpp* relating to FM.

Initialize_Sound_Timbre_2op_Mode(ch, inst_num, L_R_B)

Before an FM sound can be played by the SB16, all the FM parameters must be written to the registers (*see Table 3*). This function loads into the registers the row in the instrument table that corresponds to the index *inst_num*. An array of modulator register offsets for various channels is used to concisely write the parameters to the registers of **any** channel *ch*. The offset array is defined as follows:

```
/* Offsets for channels 1-9*/
int Offset [] =
   {
     0x00;
     0x01;
     0x02;
     0x08;
     0x09;
     0x0A;
     0x10;
     0x11;
     0x12};
```

The last argument determines the stereo mode of the sound. The argument will have one of three values: 10h (left), 20h (right), or 30h (both). These values have been defined using the #define preprocessing directive.

GenerateSound(ch, fn, block, inst_num, L_R_B)

Within the main program, *GenerateSound( )* is called to play instrument *inst_num* from the instrument table with a F-NUMBER *fn* and block number *block* on channel *ch*. Recall that F-NUMBER is **not** the frequency in hertz, but a 10-bit number loaded into registers A0h-A8h and B0h-B8h. This function uses the bit shift operator to separate the high and low bit values and write them to the proper register.

StopFMsound(ch, freqnum)

This function is used to turn off the FM sound playing on channel *ch*. Stopping a sound requires changing KEYON (*bit 5 on register B0h-B8h*) to zero. Changing only this bit in the register will produce a clicking noise, so the current value of F-NUMBER *freqnum* must be masked with KEYON=0.

4.6 Train Simulator Development Structure

As with most simulators, if some developmental structure is not used to organize the code, debugging can become overwhelming and practically impossible. The designers of the train simulator segmented the code into a series of libraries in order to isolate problems and help troubleshooting. The following figure shows the libraries that are relevant to the ethernet and serial port software:

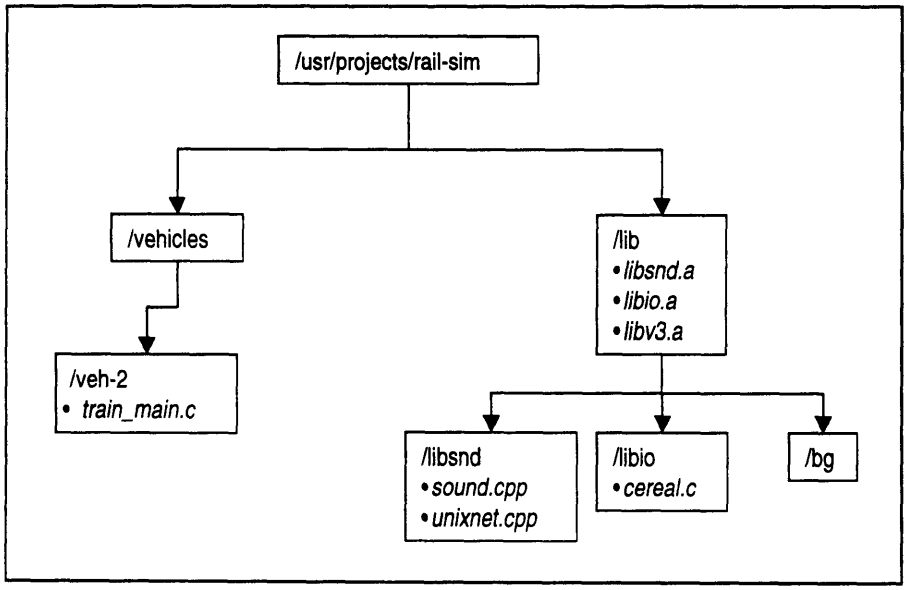


Figure 22. Directory structure and libraries used for networking and user input.

The directory */usr/projects/rail-sim* is termed the pivotal directory because it bridges the libraries and the simulation executable. Three libraries are used for networking and obtaining user input: *libsnd*, *libio*, and *bg*. After compilation, each library will place a library archive file, denoted with a "*.a*" extension, into the parent directory. These files are then used by the makefile in */vehicles/veh-2* to create the simulator executable.

Using the Makefiles

The makefile is analogous to the PC compiler project file. It is simply a convenient way to execute repeated command line compiler options. To compile the code in a library,

type "*make install*" within the library directory. This will compile and install a library archive into the parent directory.

4.6.1 Networking Library

The code used to collect and send data to the PC is located within the *libsnd* library. The program *unixnet.cpp* defines the host ethernet functions as discussed in §3.0 while *sound.cpp* defines the network initialization and data transfer functions. The latter two functions are called within the main program (*see train_main.c located on the attached disk*).

Adding A New Network Output Variable

As explained in §3.0, the programmer must ensure that the network data transfer function argument list for both the client and host agree. The following four steps should be followed when adding a new network output variable.

(1) Go to the */usr/projects/rail-sim/lib/libsnd* directory and open *sound.cpp*. At the beginning of the file, append the new data type to the end of the function declarations for *send_sound()* and *NetworkSendRecv()*.

(2) Now change both function **definitions**. In *NetworkSendRecv()*, the character format must also be added (*i.e. %d for integer or %f for float types*).

(3) Save the file and type "*make install*" at the command prompt. This will compile the new changes and install the library archive into the parent directory.

(4) Last, add the argument type to the *send_sound()* function in *train_main.c*. Make sure the variable type is in the same position within the argument list. Save the file and type "*make all*" at the command prompt.

The programmer must also change the client PC function *NetSendRecv()* defined in *trainsnd.cpp*.

4.6.2 Serial Interface Libraries

A computer can receive and send information to external devices via its parallel and serial port. The selection of each type depends on the needs of the system and how data is collected and sent. For instance, parallel communication is suitable for high bandwidth applications, but is susceptible to noise and therefore limited to short range transmission lines. A serial port, on the other hand, can send data as far as 100 feet, but with a much lower bandwidth. The choice for the train simulator is based more on ease of implementation rather than bandwidth or transmission length. A commercial serial port interface unit is used to collect user input.

The LV824-G CerealBox, made by BG Systems, serves as the data acquisition system for the SGI workstation. The CerealBox provides 24 channels of digital input/output, 8 analog inputs, and 3 analog outputs. The library located in the */usr/projects/rail-sim/lib/bg* directory contains the low level functions required to use the digital and analog features. Please refer to the BG Systems manual for details on the functions contained within this library. The *libio* library contains the functions used with the train simulator to initialize the CerealBox and transfer data.

Digital Channel Data Banks

The digital I/O channels provided by the CerealBox are divided into three banks, with 8 channels per bank. Bank 1 contains channels 1-8, bank 2 has channels 9-16, and bank 3 controls channels 17-24. Each bank can be used for input or output, but **not** both at the same time. The program *cereal.c* located in the *libio* directory defines in what mode the three available banks should operate. For the train simulator, bank 1 and 2 are set for digital input and bank 3 is used for digital output.

The data collected by the CerealBox is stored in the data structure *bgdata*. Two array members of the structure, *bgdata.din[3]* and *bgdata.dout[3]* hold the values from each bank depending on the set mode. For example, data being collected from channels 1-8 (*bank 1*) and channels 9-16 (*bank 2*) on the train simulator will be stored in *bgdata.din[0]*

and *bgdata.din[1]*, respectively. Data being sent to channels 17-27 are written to *bgdata.dout[3]*. Breaking the channels into banks of 8 is not arbitrary, but done so that each channel can represent one bit in an 8-bit computer word. Therefore, the best way to send and extract data from *bgdata.dout* or *bgdata.din* is to define a bit field. This gives the programmer a convenient method of setting individual bits values without having to calculate the equivalent hex number. The following bit field is used for this purpose:

```
/* Digital I/O Bit Field Type definition/
typdef union {
      int i;
      struct {
            unsigned      allpins:8;
      { byte;
      struct {
            unsigned      p8:1, p7:1, p6:1, p5:1,
                          p4:1, p3:1, p2:1, p1:1;
      } bits;
} digital_IO;
```

Variables declared as type *digital_IO* can access individual bits or the entire word as follows:

```
digital_IO  bank1in, bank2in, bank3out;
:
bank3out.bits.p1 = 1;   /* write a low value for channel 17 */
bank3out.bits.p5 = 0;   /* write a low value for channel 21 */

bgdata.dout[2] = bank3out.byte.allpins;   /* write all the bits to the output array */
bank1in.byte.allpins = bgdata.din[0];  /* write all the input data from ch1-8 to bank1 */
```

4.7 Cabin Enclosure

For the simulation to truly be realistic, the simulator environment had to evolve from the two monitors on a table shown in Figure 12. As a first attempt, a cabin enclose has been built to isolate the subject from the room (*see Figure 23*). A projection unit replaces the OTW monitor and the cabin structure houses the display, control boxes, and throttle (*see Figure 24*).

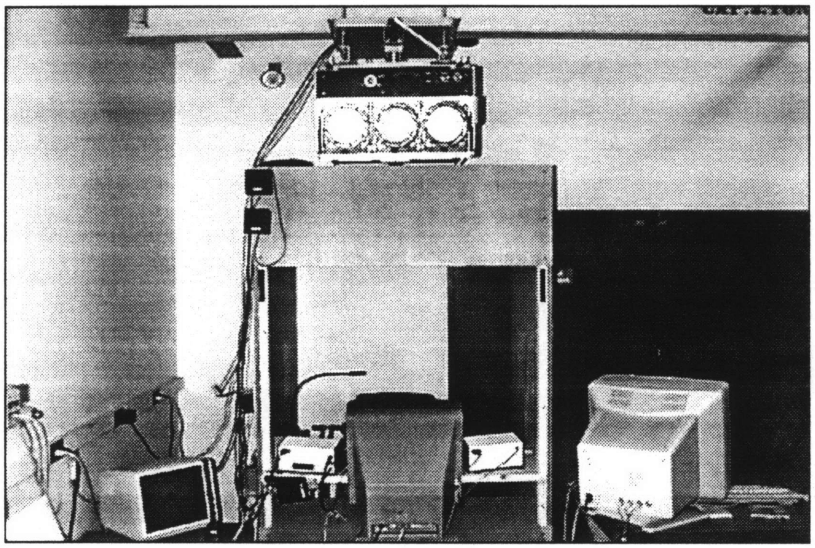


Figure 23. Front view of cabin enclosure.

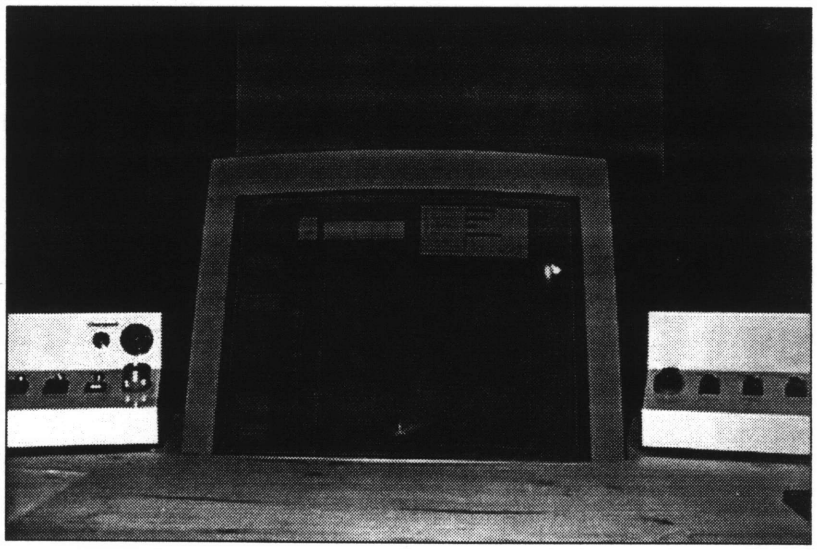


Figure 24. Subject's view from within the cabin enclosure.

4.7.1 Hardwiring Keyboard buttons

Until now, the keyboard has been used primarily to input train commands. For example, some function keys are used to open or close the doors, set cruise control, and apply emergency braking. To add realism, the CerealBox and two control boxes are used to control the train. Figure 25 show the external I/O interface boxes which connect to the CerealBox.

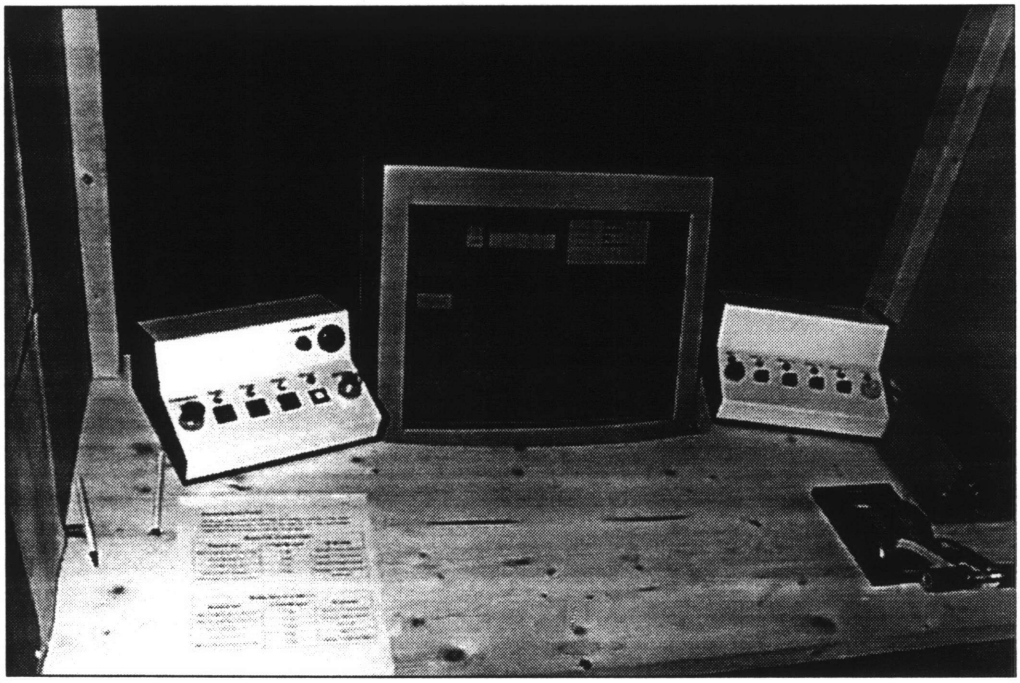


Figure 25. External user IO boxes.

All buttons have an LED that lights red or green to indicate the switch state. Some LEDs are hardwired to the switch and others, like the motor circuit breaker LEDs, illuminate according to the voltage at their terminals. This is done to signal failures with red LEDs.

Table 9 shows the channel-to-button mapping used within *train_main.c* to control the train.

Table 9. Channel-to-button Map.

	Bit	Channel	Keyboard	Function
Bank1in	P1	1		Play Bell Sound
	P2	2	F1	Motor 1 Circuit Braker
	P3	3	F2	Motor 2 Circuit Braker
	P4	4	F3	Motor 3 Circuit Braker
	P5	5	F4	Motor 4 Circuit Braker
	P6	6		Play Horn Sound
	P7	7		
	P8	8		
Bank2in	P1	9	F9	Emergency Brake Release
	P2	10	F10	Brake Pump
	P3	11	F8	Door Open/Close
	P4	12		Cruise Control On/Off
	P5	13		
	P6	14		
	P7	15	F12	Emergency Stop
	P8	16	ESC	Alerter Reset
Bank3out	P1	17		Cruise Control LED
	P2	18		Motor 1 LED
	P3	19		Motor 2 LED
	P4	20		Motor 3 LED
	P5	21		Motor 4 LED
	P6	22		Door LED
	P7	23		Speeding Flasher
	P8	24		Speeding Buzzer

Within *train_main.c*, values are written to *bank3out* according to inputs from *bank1in* and *bank2in*. The following procedure is used:

```
⋮
bgdata.dout[2] = bank3out.byte.allpins;   /* write all the bits to the output array */

send_outputs(&bgdata);  /*send a data packet the CerealBox */
check_outputs(&bgdata);  */ get the input data */

bank1in.byte.allpins = bgdata.din[0];  /* ch1-8 input data */
bank2in.byte.allpins = bgdata.din[1];  /* ch9-17 input data */
⋮
```

4.8 Seat Vibration

The last physical enhancement applied to the train simulator was to add vibrations to the subjects chair using the 15" speaker driver shown in Figure 26. Cost is the driving factor in choosing this method because other means like hydraulics systems are messy and expensive. Commercial products that use the same voice coil drive principles are also expensive and can not provide adequate bandwidth with sufficient power.

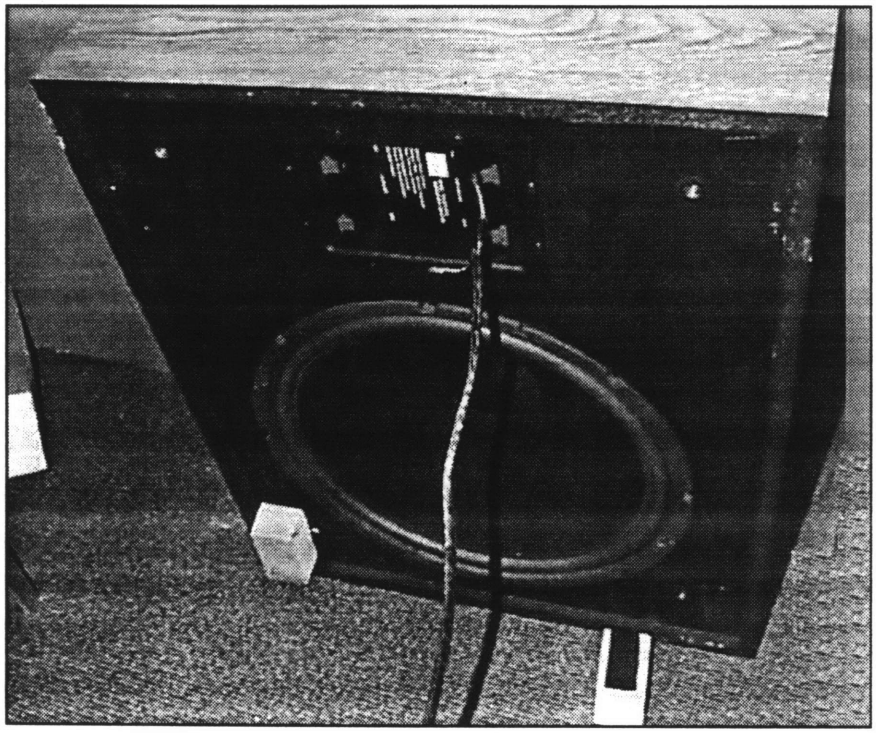


Figure 26. 15" speaker placed under the subject's chair.

This speaker has a built in low-pass filter, so playing the engine and track noise through this speaker creates a low frequency vibration on the seat above.

4.9 Enhancement Cost

Table 10 summarizes the approximate cost of all the components used to upgrade the train simulator. Considering that there are commercial train simulators that cost on the order of $100K, the upgrades done here should be considered cost effective if subsequent experiments provide more meaningful data.

Table 10. Enhancement Cost Summary.

ITEM	APPROXIMATE COST
Pentium PC	$2,000
SoundBlaster 16 Sound Card	$100
Micro Works Sound System	$300
Network Adapter	$200
LAN Workplace for DOS	$100
15" Speaker with crossover	$150
Cabin Enclose	$250
CerealBox serial data acquisition	$800
Control Boxes	$100
TOTAL	**$4,000**

As mentioned early on, there is a balance between added fidelity and added cost. A hydraulically controlled six degree-of-freedom platform would be a nice addition, but it would not be appropriate considering the type of human-machine studies conducted.

Chapter 5: CONCLUSION

The objective of this design project was to develop a framework for adding sound enhancements to graphical simulations. The tools used within this framework include programming the SoundBlaster 16 PC sound card to play both digitally sampled sounds and FM synthesized sounds, establish an ethernet link between a PC and Unix workstation, and interface external input devices. Each of these tools has been successfully applied to the Volpe high-speed train simulator.

The most difficult aspect of adding sound to the Volpe train simulator was synchronizing the track sound components. The solution of adding a variable delay at the end of each component is only one solution, however. A more complicated method can be constructed by taking advantage of the time-base running on the PC. Instead of using a hard delay that actually stops the program after each component, triggering can be based on time. This will be advantageous when the calculated delay between click-clack sequences is long (*i.e. low speeds*). Using the current method, the time delay time is limited to 3 seconds, so that the program does not lag behind. Because the train speeds up quickly, the delays are usually no greater than 500 msec and thus are not noticeable. Hence, if triggering is based on time, the main control loop is always running and can be checking for user inputs like bell and horn flags continuously.

In addition, it is possible to make the system more user-friendly. In particular, when a new sound or transfer variable needs to be included, several functions must be changed and recompiled on both the PC and SGI. A streamlined version could possibly have a user interface that controls all the variables being transferred and allows modifications without recompiling the code.

There are also other sounds effects that can be added to increase the overall realism without increasing the system cost. For example, wind turbulence, could be digitally sampled and played back during the simulation. Also, the train horn could be improved to play as long as a lever is pulled, as opposed to just once each time a button is pressed. This would require breaking the original digital recording into three sections: onset, sustain, and decay. When the lever is pulled, the onset is played once followed by the sustain phase. This lasts until the lever is released, at which point sustain then transitions into the decay phase. This scheme would allow the train operator to control the horn duration; as is the case when approaching a grade crossing.

Based on my experience, the following background allows one to more effectively add sound to simulations: signal processing , digital filtering, and programming hardware. In addition, knowing how to use a sound wave editor is essential for managing digital sound effects. A good ear also helps.

APPENDICES

Appendix A: FM Sample Code

FM sample code goes here.

```
/*
 * FM synthesizer low-level interface demo program.
 * Copyright (c) 1993 Creative Labs, Inc.
 */


/*
 * This program is not intended to explain all the aspects of FM sound
 * generation on Sound Blaster cards. The chips are too complicated for
 * that. This program is just to demonstrate how to generate a tone and
 * control the left and right channels. For more information on the FM
 * synthesizer chip, contact Yamaha.
 *
 * Here's a brief description of FM: Each sound is created by two operator
 * cells (called "slots" in the Yamaha documentation), a modulator and a
 * carrier. When FM synthesis was invented, the output value of the
 * modulator affected the frequency of the carrier. In the Yamaha chips, the
 * modulator output actually affects the phase of the carrier instead of
 * frequency, but this has a similar effect.
 *
 * Normally the modulator and carrier would probably be connected in series
 * for complex sounds. For this program, I wanted a pure sine wave, so I
 * connected them in parallel and turned the modulator output down all the
 * way and used just the carrier.
 *
 * Sound Blaster 1.0 - 2.0 cards have one OPL-2 FM synthesis chip at
 * addresses 2x8 and 2x9 (base + 8 and base + 9). Sound Blaster Pro version
 * 1 cards (CT-1330) achieve stereo FM with two OPL-2 chips, one for each
 * speaker. The left channel FM chip is at addresses 2x0 and 2x1. The right
 * is at 2x2 and 2x3. Addresses 2x8 and 2x9 address both chips
 * simultaneously, thus maintaining compatibility with the monaural Sound
 * Blaster cards. The OPL-2 contains 18 operator cells which make up the
 * nine 2-operator channels. Since the CT-1330 SB Pro has two OPL-2 chips,
 * it is therefore capable of generating 9 voices in each speaker.
 *
 * Sound Blaster Pro version 2 (CT-1600) and Sound Blaster 16 cards have one
 * OPL-3 stereo FM chip at addresses 2x0 - 2x3. The OPL-3 is separated into
 * two "banks." Ports 2x0 and 2x1 control bank 0, while 2x2 and 2x3 control
 * bank 1. Each bank can generate nine 2-operator voices. However, when the
 * OPL-3 is reset, it switches into OPL-2 mode. It must be put into OPL-3
 * mode to use the voices in bank 1 or the stereo features. For stereo
 * control, each channel may be sent to the left, the right, or both
 * speakers, controlled by two bits in registers C0H - C8H.
 *
 * The FM chips are controlled through a set of registers. The following
 * table shows how operator cells and channels are related, and the register
 * offsets to use.
 *
```

```
* FUNCTION MODULATOR- -CARRIER-- MODULATOR- -CARRIER-- MODULATOR- -
CARRIER--
* OP CELL  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
* CHANNEL  1  2  3  1  2  3  4  5  6  4  5  6  7  8  9  7  8  9
* OFFSET  00 01 02 03 04 05 08 09 0A 0B 0C 0D 10 11 12 13 14 15
*
* An example will make the use of this table clearer: suppose you want to
* set the attenuation of both of the operators of channel 4. The KSL/TOTAL LEVEL
* registers (which set the attenuation) are 40H - 55H. The modulator for
* channel 4 is op cell 7, and the carrier for channel 4 is op cell 10. The
* offsets for the modulator and carrier cells are 08H and 0BH, respectively.
* Therefore, to set the attenuation of the modulator, you would output a
* value to register 40H + 08H == 48H, and to set the carrier's attenuation,
* you would output to register 40H + 0BH == 4BH.
*
* In this program, I used just channel 1, so the registers I used were 20H,
* 40H, 60H, etc., and 23H, 43H, 63H, etc.
*
* The frequencies of each channel are controlled with a frequency number and
* a multiplier. The modulator and carrier of a channel both get the same
* frequency number, but they may be given different multipliers. Frequency
* numbers are programmed in registers A0H - A8H (low 8 bits) and B0H - B8H
* (high 2 bits). Those registers control entire channels (2 operators), not
* individual operator cells. To turn a note on, the key-on bit in the
* appropriate channel register is set. Since these registers deal with
* channels, you do not use the offsets listed in the table above. Instead,
* add (channel-1) to A0H or B0H. For example, to turn channel 1 on,
* program the frequency number in registers A0H and B0H, and set the key-on
* bit to 1 in register B0H. For channel 3, use registers A2H and B2H.
*
* Bits 2 - 4 in registers B0H - B8H are the block (octave) number for the
* channel.
*
* Multipliers for each operator cell are programmed through registers 20H -
* 35H. The table below shows what multiple number to program into the
* register to get the desired multiplier. The multiple number goes into
* bits 0 - 3 in the register. Note that it's a bit strange at the end.
*
*  multiple number   multiplier     multiple number   multiplier
*       0              1/2              8               8
*       1               1               9               9
*       2               2              10              10
*       3               3              11              10
*       4               4              12              12
*       5               5              13              12
*       6               6              14              15
*       7               7              15              15
*
* This equation shows how to calculate the required frequency number (to
* program into registers A0H - A8H and B0H - B8H) to get the desired
* frequency:
*           fn=(long)f * 1048576 / b / m /50000L
* where f is the frequency in Hz,
*     b is the block (octave) number between 0 and 7 inclusive, and
*     m is the multiple number between 0 and 15 inclusive.
*
```

```
 */


#define STEREO       // Define this for SBPro CT-1330 or later card.
#define OPL3         // Also define this for SBPro CT-1600 or later card.


#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <ctype.h>
#include <dos.h>

#define KEYON  0x20    // key-on bit in regs b0 - b8

 /* These are offsets from the base I/O address. */
#define FM      8      // SB (mono) ports (e.g. 228H and 229H)
#define PROFM1  0      // On CT-1330, this is left OPL-2.  On CT-1600 and
                       // later cards, it's OPL-3 bank 0.
#define PROFM2  2      // On CT-1330, this is right OPL-2.  On CT-1600 and
                       // later cards, it's OPL-3 bank 1.



#ifdef OPL3
 #define LEFT    0x10
 #define RIGHT                    0x20
#endif


unsigned IOport;       // Sound Blaster port address



void mydelay(unsigned long clocks)
/*
 * "clocks" is clock pulses (at 1.193180 MHz) to elapse, but remember that
 * normally the system timer runs in mode 3, in which it counts down by twos,
 * so delay3(1193180) will only delay half a second.
 *
 *   clocks = time * 2386360
 *
 *    time = clocks / 2386360
 */
{
  unsigned long elapsed=0;
  unsigned int last,next,ncopy,diff;

  /* Read the counter value. */
  outp(0x43,0);                    /* want to read timer 0 */
  last=inp(0x40);                  /* low byte */
  last=~((inp(0x40)<< 8) + last);        /* high byte */

  do {
    /* Read the counter value. */
    outp(0x43,0);                  /* want to read timer 0 */
```

```
    next=inp(0x40);                    /* low byte */
    ncopy=next=~((inp(0x40)<< 8) + next);  /* high byte */

    next-=last;     /* this is now number of elapsed clock pulses since last read */

    elapsed += next; /* add to total elapsed clock pulses */
    last=ncopy;
  } while (elapsed<clocks);
}


int base16(char **str, unsigned *val)
/* Takes a double pointer to a string, interprets the characters as a
 * base-16 number, and advances the pointer.
 * Returns 0 if successful, 1 if not.
 */
{
  char c;
  int digit;
  *val = 0;

  while ( **str != ' ') {
    c = toupper(**str);
    if (c >= '0' && c <= '9')
      digit = c - '0';
    else if (c >= 'A' && c <= 'F')
      digit = c - 'A'  + 10;
    else
      return 1;         // error in string

    *val = *val * 16 + digit;
    (*str)++;
  }
  return 0;
}


int base10(char **str, unsigned *val)
/* Takes a double pointer to a string, interprets the characters as a
 * base-10 number, and advances the pointer.
 * Returns 0 if successful, 1 if not.
 */
{
  char c;
  int digit;
  *val = 0;

  while ( **str != ' ') {
    c = toupper(**str);
    if (c >= '0' && c <= '9')
      digit = c - '0';
    else
      return 1;         // error in string
```

```
    *val = *val * 10 + digit;
    (*str)++;
  }
  return 0;
}


unsigned ReadBlasterEnv(unsigned *port, unsigned *irq, unsigned *dma8,
 unsigned *dma16)
/* Gets the Blaster environment statement and stores the values in the
 * variables whose addresses were passed to it.
 * Returns:
 *  0 if successful
 *  1 if there was an error reading the port address.
 *  2 if there was an error reading the IRQ number.
 *  3 if there was an error reading the 8-bit DMA channel.
 *  4 if there was an error reading the 16-bit DMA channel.
 */
{
  char    *env;
  unsigned val;
  int     digit;

  env = getenv("BLASTER");

  while (*env) {
    switch(toupper( *(env++) )) {
      case 'A':
        if (base16(&env, port))     // interpret port value as hex
          return 1;                 // error
        break;
      case 'I':
        if (base10(&env, irq))      // interpret IRQ as decimal
          return 2;                 // error
        break;
      case 'D':
        if (base10(&env, dma8))     // 8-bit DMA channel is decimal
          return 3;
        break;
      case 'H':
        if (base10(&env, dma16))    // 16-bit DMA channel is decimal
          return 4;
        break;
      default:
        break;
    }
  }

  return 0;
}


void FMoutput(unsigned port, int reg, int val)
/* This outputs a value to a specified FM register at a specified FM port. */
```

```
{
  outp(port, reg);
  mydelay(8);        /* need to wait 3.3 microsec */
  outp(port+1, val);
  mydelay(55);       /* need to wait 23 microsec */
}


void fm(int reg, int val)
/* This function outputs a value to a specified FM register at the Sound
 * Blaster (mono) port address.
 */
{
  FMoutput(IOport+FM, reg, val);
}


void Profm1(int reg, int val)
/* This function outputs a value to a specified FM register at the Sound
 * Blaster Pro left FM port address (or OPL-3 bank 0).
 */
{
  FMoutput(IOport+PROFM1, reg, val);
}


void Profm2(int reg, int val)
/* This function outputs a value to a specified FM register at the Sound
 * Blaster Pro right FM port address (or OPL-3 bank 1).
 */
{
  FMoutput(IOport+PROFM2, reg, val);
}



void main(void)
{
  int i,val1,val2;

  int block,b,m,f,fn;

  unsigned dummy;


  clrscr();

  ReadBlasterEnv(&IOport, &dummy, &dummy, &dummy);

#ifdef STEREO
 #ifdef OPL3
  printf("Program compiled for Sound Blaster Pro ver. 2 (CT-1600) and SB16 cards.\n");
 #else
  printf("Program compiled for Sound Blaster Pro ver. 1 (CT-1330) cards.\n");
```

```
 #endif
#else
  printf("Program compiled for Sound Blaster 1.0 - 2.0 cards (monaural).\n");
#endif


  fm(1,0);      /* must initialize this to zero */

#ifdef OPL3
  Profm2(5, 1); /* set to OPL3 mode, necessary for stereo */
  fm(0xC0,LEFT | RIGHT | 1);    /* set both channels, parallel connection */
#else
  fm(0xC0,            1);    /* parallel connection */
#endif

  /*****************************************
   * Set parameters for the carrier cell *
   *****************************************/

  fm(0x23,0x21); /* no amplitude modulation (D7=0), no vibrato (D6=0),
                  * sustained envelope type (D5=1), KSR=0 (D4=0),
                  * frequency multiplier=1 (D4-D0=1)
                  */

  fm(0x43,0x0);  /* no volume decrease with pitch (D7-D6=0),
                  * no attenuation (D5-D0=0)
                  */

  fm(0x63,0xff); /* fast attack (D7-D4=0xF) and decay (D3-D0=0xF) */
  fm(0x83,0x05); /* high sustain level (D7-D4=0), slow release rate (D3-D0=5) */


  /*****************************************
   * Set parameters for the modulator cell *
   *****************************************/

  fm(0x20,0x20); /* sustained envelope type, frequency multiplier=0   */
  fm(0x40,0x3f); /* maximum attenuation, no volume decrease with pitch */

  /* Since the modulator signal is attenuated as much as possible, these
   * next two values shouldn't have any effect.
   */
  fm(0x60,0x44); /* slow attack and decay */
  fm(0x80,0x05); /* high sustain level, slow release rate */


  /*************************************************
   * Generate tone from values looked up in table. *
   *************************************************/

  printf("440 Hz tone, values looked up in table.\n");
  fm(0xa0,0x41); /* 440 Hz */
  fm(0xb0,0x32); /* 440 Hz, block 0, key on */

  getche();
```

```
fm(0xb0,0x12);  /* key off */


/*****************************************
 * Generate tone from a calculated value. *
 *****************************************/

printf("440 Hz tone, values calculated.\n");
block=4;        /* choose block=4 and m=1 */
m=1;                    /* m is the frequency multiple number */
f=440;          /* want f=440 Hz */
b= 1 << block;

/* This is the equation to calculate frequency number from frequency. */

fn=(long)f * 1048576 / b / m /50000L;

fm(0x23,0x20 | (m & 0xF));  /* 0x20 sets sustained envelope, low nibble
                        * is multiple number
                        */
fm(0xA0,(fn & 0xFF));
fm(0xB0,((fn >> 8) & 0x3) + (block << 2) | KEYON);

getche();


/*********************************************************
 * Generate a range of octaves by changing block number. *
 *********************************************************/

printf("Range of frequencies created by changing block number.\n");
for (block=0; block<=7; block++) {
  printf("f=%5ld Hz (press Enter)\n",(long)440*(1 << block)/16);
  fm(0xB0,((fn >> 8) & 0x3) + (block << 2) | KEYON);
  getche();
}


/*****************************************************************
 * Generate a range of frequencies by changing frequency number. *
 *****************************************************************/

printf("Range of frequencies created by changing frequency number.\n");
block=4;
for (fn=0; fn<1024; fn++) {
  fm(0xA0,(fn & 0xFF));
  fm(0xB0,((fn >> 8) & 0x3) + (block << 2) | KEYON);
  delay(1);
}


/*****************************************************************
 * Single tone again.  Both channels, then if on stereo board,   *
 * play tone in just the left channel, then just the right channel. *
 *****************************************************************/
```

```
  printf("440 Hz again, both channels.\n");
  block=4;
  fn=577;              /* This number makes 440 Hz when block=4 and m=1 */
  fm(0xA0,(fn & 0xFF));
  fm(0xB0,((fn >> 8) & 0x3) + (block << 2) | KEYON);

#ifdef STEREO
 #ifdef OPL3
   /* This left and right channel stuff is the only part of this program
    * that uses OPL3 mode.  Everything else is available on the OPL2.
    */

   getche();
   printf("Left channel only\n");
   fm(0xC0,LEFT | 1);     /* set left channel only, parallel connection */

   getche();
   printf("Right channel only\n");
   fm(0xC0,RIGHT | 1);    /* set right channel only, parallel connection */
 #else
   getche();
   fm(0xB0,((fn >> 8) & 0x3) + (block << 2));      // key off

   printf("Left channel only\n");
   Profm1(0xB0,((fn >> 8) & 0x3) + (block << 2) | KEYON);

   getche();
   Profm1(0xB0,((fn >> 8) & 0x3) + (block << 2));  // key off

   printf("Right channel only\n");
   Profm2(0xB0,((fn >> 8) & 0x3) + (block << 2) | KEYON);

 #endif
#endif

  /*********************************
   * Attenuate the signal by 3 dB. *
   *********************************/

  getche();
  fm(0xB0,((fn >> 8) & 0x3) + (block << 2) | KEYON);
  printf("Attenuated by 3 dB.\n");
  fm(0x43,4);    /* attenuate by 3 dB */
  getche();

  fm(0xB0,((fn >> 8) & 0x3) + (block << 2));

#ifdef OPL3
  /* Set OPL-3 back to OPL-2 mode, because if the next program to run was
   * written for the OPL-2, then it won't set the LEFT and RIGHT bits to
   * one, so no sound will be heard.
   */
  Profm2(5, 0);  /* set back to OPL2 mode */
#endif
}
Mixtest goes here. code goes here.
```

Appendix B: Digital Playback Sample Code

```
/*     SMIXC is Copyright 1995 by Ethan Brodsky.  All rights reserved     */

/**** MIXTEST.C ************************************************************ */

#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "detect.h"
#include "smix.h"

#define ON  1
#define OFF 0

#define TRUE  1
#define FALSE 0

#define SHAREDEMB    /* Undefine this to store all sounds in separate EMBs */
#define NUMSOUNDS 6  /* Change this if you want to add more sounds         */

char *resource_file = "mixtest.snd";

char *sound_key[NUMSOUNDS] =
 {
   "JET",
   "GUN",
   "CRASH",
   "CANNON",
   "LASER",
   "GLASS"
 };

int baseio, irq, dma, dma16;

SOUND *sound[NUMSOUNDS];

int  stop;

long counter;

char inkey;
int  num;
int  temp;

void ourexitproc(void)
 {
   int i;

   for (i=0; i < NUMSOUNDS; ++i)
     if (sound[i] != NULL) free_sound(sound+i);
#ifdef SHAREDEMB
   shutdown_sharing();
#endif
```

```
 }

void init(void)
 {
  int i;

  randomize();

  printf("-------------------------------------------\n");
  printf("Sound Mixing Library v1.27 by Ethan Brodsky\n");
  if (!detect_settings(&baseio, &irq, &dma, &dma16))
   {
    printf("ERROR:  Invalid or non-existant BLASTER environment variable!\n");
    exit(EXIT_FAILURE);
   }
  else
   {
    if (!init_sb(baseio, irq, dma, dma16))
     {
      printf("ERROR:  Error initializing sound card!\n");
      exit(EXIT_FAILURE);
     }
   }

  printf("BaseIO=%Xh    IRQ%u    DMA8=%u    DMA16=%u\n", baseio, irq, dma, dma16);

  printf("DSP version %.2f:  ", dspversion);
  if (sixteenbit)
   printf("16-bit, ");
  else
   printf("8-bit, ");
  if (autoinit)
   printf("Auto-initialized\n");
  else
   printf("Single-cycle\n");

  if (!init_xms())
   {
    printf("ERROR:  Can not initialize extended memory\n");
    printf("HIMEM.SYS must be installed\n");
    exit(EXIT_FAILURE);
   }
  else
   {
    printf("Extended memory successfully initialized\n");
    printf("Free XMS memory: %uk    ", getfreexms());
    if (!getfreexms())
     {
      printf("ERROR:  Insufficient free XMS\n");
      exit(EXIT_FAILURE);
     }
    else
     {
      printf("Loading sounds\n");
#ifdef SHAREDEMB
      init_sharing();
```

```
#endif
        open_sound_resource_file(resource_file);

        for (i=0; i < NUMSOUNDS; i++)
          load_sound(&(sound[i]), sound_key[i]);
        atexit(ourexitproc);

        close_sound_resource_file();
       }
     }
   init_mixing();
   printf("\n");
 }

void shutdown(void)
 {
   int i;

   shutdown_mixing();
   shutdown_sb();

   for (i=0; i < NUMSOUNDS; i++)
     free_sound(sound+i);
#ifdef SHAREDEMB
   shutdown_sharing();
#endif
   printf("\n");
 }

int main(void)
 {
   init();

   start_sound(sound[0], 0, ON); /* Start up the jet engine */

   printf("Press:\n");
   printf("  1)  Machine Gun\n");
   printf("  2)  Crash\n");
   printf("  3)  Cannon\n");
   printf("  4)  Laser\n");
   printf("  5)  Breaking glass\n");
   printf("  Q)  Quit\n");

   stop = FALSE;

   counter = 0;

   do
    {
     /* Increment and display counters */
      counter++;
      cprintf("%8lu %8lu %4u", counter, intcount, voicecount);
      gotoxy(1, wherey());

     /* Maybe start a random sound */
      if (!random(10000))
```

```
      {
       num = (random(NUMSOUNDS-1))+1;
       start_sound(sound[num], num, OFF);
      }

    /* Start a sound if a key is pressed */
     if (kbhit())
      {
       inkey = getch();
       if ((inkey >= '0') && (inkey <= '9'))
        {
         num = inkey - '0'; /* Convert to integer */
         if (num < NUMSOUNDS)
          start_sound(sound[num], num, FALSE);
        }
       else
        stop = TRUE;
      }
   }
  while (!stop);

  printf("\n");
  stop_sound(0); /* Stop the jet engine */

  shutdown();

  return(EXIT_SUCCESS);
 }
```

Appendix C: Ethernet Code

Client Code:

```
/*****************************************************************************
 * Copyright (C) Human-Machines Systems Laboratory, MIT, 1994
 * File: NetClint.h
 *    Function: network client program
 *    Created in C by Shi-Ken Chen in 1993.
 *    Modified to C++ by Jie Ren in 1994.
 *    Added HostIP class by Steven G. Villareal in 1997
 *****************************************************************************/

extern "C" {
//#include <sys/socket.h>
#include "socket.h" //did this to keep all the relevant file in 1 dir.
}

#ifndef _netclint_h
#define _netclint_h


class NetClient {
    private:
        int socketOpen;
        fd_set readfds, writefds;
        char hostip[15],inchar;
    protected:
    public:

    NetClient(char *);
    ~NetClient();
    int  open( int );
    void close();
    int send( char *, int);
    int recv( char *, int);
    int isreadready();
    int iswriteready();
};

class HostIP {
    private:
        char hostip[15];
    public:

    HostIP(char *);
    ~HostIP();
    char *returnIP();
};

#endif /*_NetClient_h */
```

```
/*********************************************************************
 * Copyright (C) Human-Machines Systems Laboratory, MIT, 1994
 * File: NetClint.cpp
 *    Function: network client program
 *    Created in C by Shi-Ken Chen in 1993.
 *    Modified to C++ by Jie Ren in 1994.
 *        Added HostIP class by Steven G. Villareal in 1997.
 *********************************************************************/
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <process.h>       // Necessary to run in C++ environment
#include "netclint.h"

/*
 * host : ip address, such as "89.0.0.5"
 */
NetClient::NetClient(char *host )
{
    strcpy(hostip, host);
    if (!loaded()) {   // to see if the network driver is installed.
        printf("The TCP/IP protocol stack is not loaded\n");
        exit(1);
    }
    socketOpen = -1;
}
NetClient::~NetClient()
{
    if ( socketOpen >= 0 ) close();
}

//---------------------------------------------------------------------
//    Make the network connection
// port: 1024-4999, has to match the port number in server!
// return: 1 - open successfully
//         0 - can not open.
//---------------------------------------------------------------------
int NetClient::open( int port)
{
    struct sockaddr_in addr;
    int  i;

    // make a local socket
    if ((socketOpen = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        soperror("socket");
        return 0;
    }

    // connect the local socket to sever.
    unsigned long remote_ip;
    char *host = hostip;
    if ((remote_ip = rhost(&host)) == -1) {
        printf("Unknown or illegal hostname = %s", hostip);
        return 0;
    }
```

```
    printf( "\n\n Opened netclient (%s). Socket #%d ", hostip, socketOpen );

    bzero ((char *)&addr, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    addr.sin_addr.s_addr = remote_ip;
    if (connect(socketOpen, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
        soperror("connect");
        return 0;
    }

    return 1;
}

//---------------------------------------------------------------------
//   Close the connection and the socket
//---------------------------------------------------------------------
void NetClient::close()
{
    soclose(socketOpen);
    socketOpen = -1;
}

//---------------------------------------------------------------------
// To see if the socket (network) is connected.
// return :  1 - yes
//           0 - no
//---------------------------------------------------------------------
int NetClient::isreadready()
{
        struct timeval seltime;

    FD_ZERO(&readfds);
    FD_SET(socketOpen, &readfds);

    seltime.tv_sec = 1; seltime.tv_usec = 0;
    int se= select( socketOpen+1, &readfds, (fd_set *) 0, (fd_set *) 0, &seltime );
    if (se <= 0 )  return 0;
    return 1;
}

int NetClient::iswriteready()
{
    struct timeval seltime;

    FD_ZERO(&writefds);
    FD_SET(socketOpen, &writefds);
    seltime.tv_sec = 1; seltime.tv_usec = 0;
    int se= select( socketOpen+1, (fd_set *) 0, &writefds, (fd_set *) 0, &seltime );
    if (se <= 0 )  return 0;

    return 1;
}

// send buffer_s (length of slen) to server
// return: the number of bytes sent.
```

```
//      < -1    fail.
int NetClient::send( char *buffer_s, int slen)
{
    int sn;
        if ( FD_ISSET(socketOpen, &writefds) ) {
            sn = sowrite( socketOpen, buffer_s, slen);  //BUFFER_SIZE );
            if (sn < 0)  {
                //soclose( socketOpen );
fprintf(stderr, "\07 send error! \n");
                return -1;
            }
        }
    return sn;
}

// receive buffer_r (length of rlen) from server
// return : the number of bytes received
//      : -1 - failure.
int NetClient::recv( char *buffer_r, int rlen)
{
    int len;
        if (FD_ISSET(socketOpen, &readfds)) {
// printf("R ");
            len = soread( socketOpen, buffer_r, rlen);
// printf("%d, %s ",len,buffer_r);
        } else {
fprintf(stderr,"\07 receive failure!\n");
            return -1;
        }
        return len;
}

/* This function is used to prompt the user for the host IP
   address if it has not been enterned entered on the command line. */
HostIP::HostIP(char *commandline)
{
    strcpy(hostip, (commandline+1));  //copy command line arguments
    cout << "command line argument entered is "<<hostip<<endl;
    if(commandline[0]!='-') {   //if not command arguments ask for them
        cout<<"Enter the Host IP Numerical Address: " ;
        cin >> hostip;
    }
    else {
        cout << "Command line arguemt used!"<<endl;
    }
}

HostIP::~HostIP()
{

}

char *HostIP::returnIP(void)
{
    return hostip;
}
```

```
/*****************************************************************************
 Copyright (C) Human-Machines Systems Laboratory, MIT, 1997
 File: unihost.cpp
    The following program is a modification to the original networking
    code written by Dr. Chin and Dr. Ren and prompts the user to enter
    the ip address of the host machine.  This provides a better user
    environment.  On the command line, type "-ipadress" after the file
    name otherwise you will be prompted to enter the ip address.
*****************************************************************************/
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <dos.h>
#include <ctype.h>
#include <string.h>
#include <sys\timeb.h>
#include <math.h>
#include <stddef.h>
#include "netclint.h"
#define ON  1
#define OFF 0
#define TRUE  1
#define FALSE 0

/*Networking Definitions*************************************************/
#define ESC 27
#define ECHO_PORT 1109
#define BUFFER_SIZE 80

/*Function Prototypes****************************************************/
void InitializeNetwork(NetClient &enet);
void NetSendRecv(NetClient &enet, float dataout, float *data1in, float *data2in,
                 int *data3in, int  *data4in);

      /*Networking Global Variables*/
int host, len;
char buffer_recv[BUFFER_SIZE], buffer_send[BUFFER_SIZE];


void main(int argc,char *argv[])
{
    int  transfercount =0, data3in, data4in;
    float  dataout,datain1,datain2;
    dataout = 579;

    HostIP ipaddress(argv[1]);
    NetClient enet(ipaddress.returnIP());
    InitializeNetwork(enet);

    do{
       NetSendRecv(enet, transfercount,  &datain1,  &datain2, &data3in, &data4in);
       transfercount++;
       cout <<"Transfer #"<<transfercount
            <<" buffer_recv = " << buffer_recv
```

```
                << " buffer_send =  "<< buffer_send
                <<endl;

    } while (!kbhit());

}


/*Network Functions */

void InitializeNetwork(NetClient &enet)
{
    if (!enet.open(ECHO_PORT)) {
        printf("couldn't open network connection\n");
        exit(1);
    }
}

void NetSendRecv( NetClient &enet, float dataout, float *data1in, float *data2in,
                        int *data3in,  int  *data4in)
{

    if(enet.iswriteready()) {
        sprintf(buffer_send,"%3.2f",dataout); //send argument variable
        enet.send((char *)buffer_send, BUFFER_SIZE); //Data sent out of the pc
        printf("write ready \n\n");
    }
    else printf("write not ready \n");


    if(enet.isreadready()) {
        len=enet.recv((char *)buffer_recv,BUFFER_SIZE);
        printf("reading data from the buffer \n\n");
        sscanf(buffer_recv,"%f %f %d %d",data1in,data2in,data3in,data4in); //Data received from
host
        if(len<=0) {
              printf("\n zero receiving length");
              exit(1);
        }
    }
    else {
        printf("\n not connected: read not ready \n");
    }

}
```

```
/*****************************************************************************
Copyright (C) Human-Machines Systems Laboratory, MIT, 1997
Authors(s): Jianjuan Hu, Steven G. Villareal

File: multinet.cpp
    The following program  uses the  networking code written by Dr. Chin
    and Dr. Ren to link a PC (client) to 2 SGI workstations (hosts). The IP
    addresses of the 2 host machines are defined here.
*****************************************************************************/

#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <dos.h>
#include <ctype.h>
#include <string.h>
#include <sys\timeb.h>
#include <math.h>
#include <stddef.h>
#include "netclint.h"
#define ON  1
#define OFF 0
#define TRUE  1
#define FALSE 0

/*Networking Definitions*************************************************/
#define ESC 27
#define ECHO_PORT_1 1109   //Must have SAME definition in host 1 code
#define ECHO_PORT_2 1100//Must have SAME definition in host 2 code
#define BUFFER_SIZE 80     //Must have SAME definitionon both host codes

/*Function Prototypes*****************************************************/
void InitializeNetwork(NetClient &enet, int &echo_port);
void NetSendRecv_1(NetClient &enet, float dataout, float *data1in, float *data2in,
                   int *data3in, int  *data4in);
void NetSendRecv_2(NetClient &enet, float dataout, float *data1in, float *data2in,
                   int *data3in, int  *data4in);

/*Networking Global Variables*/
int  len, echo_port;
char buffer_recv_1[BUFFER_SIZE], buffer_send_1[BUFFER_SIZE];
char buffer_recv_2[BUFFER_SIZE], buffer_send_2[BUFFER_SIZE];

void main(int argc,char *argv[])
{
    int  transfercount =0, data3in, data4in;
    float  dataout_1, dataout_2,datain1,datain2;
    dataout_1 = 111;
    dataout_2 = 222;

    /* Initialization for Host 1 object */
    NetClient Host1("89.0.0.7"); //host1 IP is set within code here
    InitializeNetwork(Host1, ECHO_PORT_1); //second argument not needed when
                                           //using only 1 host.
```

```
    /* Initialization for Host 2 object */
    NetClient Host2("89.0.0.5"); //host2 IP is set within code
    InitializeNetwork(Host2, ECHO_PORT_2);

    do{
       NetSendRecv_1(Host1, dataout_1,  &datain1,  &datain2, &data3in, &data4in);
       NetSendRecv_2(Host2, dataout_2,  &datain1,  &datain2, &data3in, &data4in);
       transfercount++;
       cout <<"Transfer #"<<transfercount
            <<" buffer_recv_1 = " << buffer_recv_1
            << " buffer_send_1 =  "<< buffer_send_1
            <<endl;

       cout <<"Transfer #"<<transfercount
            <<" buffer_recv_2 = " << buffer_recv_2
            << " buffer_send_2 =  "<< buffer_send_2
            <<endl;

    } while (!kbhit());

}



/*Network Functions */

void InitializeNetwork(NetClient &enet, int &echo_port)
/* Added the second argument so that only one function is needed to initialize
    multiple hosts*/
{
    if (!enet.open(echo_port)) {
       printf("couldn't open network connection\n");
       exit(1);
    }
}

void NetSendRecv_1( NetClient &enet, float dataout, float *data1in, float *data2in,
                    int *data3in,  int  *data4in)
/* The number and type of arguments should agree with host sending function*/
{

    if(enet.iswriteready()) {
       sprintf(buffer_send_1,"%3.2f",dataout); //send argument variable
       enet.send((char *)buffer_send_1, BUFFER_SIZE); //Data sent out of the pc
       printf("write ready \n\n");
    }
    else printf("write not ready \n");


    if(enet.isreadready()) {
       len=enet.recv((char *)buffer_recv_1,BUFFER_SIZE);
       printf("reading data from the buffer \n\n");
       sscanf(buffer_recv_1,"%f  %f  %d  %d",data1in,data2in,data3in,data4in); //Data  received
from host
       if(len<=0) {
            printf("\n zero receiving length");
```

```
            exit(1);
        }
    }
    else {
        printf("\n not connected: read not ready \n");
    }

}


void NetSendRecv_2( NetClient &enet, float dataout, float *data1in, float *data2in,
                    int *data3in,  int  *data4in)
{

    if(enet.iswriteready()) {
        sprintf(buffer_send_2,"%3.2f",dataout); //send argument variable
        enet.send((char *)buffer_send_2, BUFFER_SIZE); //Data sent out of the pc
        printf("write ready \n\n");
    }
    else printf("write not ready \n");


    if(enet.isreadready()) {
        len=enet.recv((char *)buffer_recv_2,BUFFER_SIZE);
        printf("reading data from the buffer \n\n");
        sscanf(buffer_recv_2,"%f  %f  %d  %d",data1in,data2in,data3in,data4in);  //Data  received
from host
        if(len<=0) {
            printf("\n zero receiving length");
            exit(1);
        }
    }
    else {
        printf("\n not connected: read not ready \n");
    }

}
```

Host Programs:

```
/*************************************************************
  Copyright (C) Human-Machine Systems Laboratory,
          Massachusetts Institute of Technology
  File: unixnet.h
  Function: Ether-network connection between IRIS and PC
     Created by Shi-Ken Chen in C, 1993
     Modified by Jie Ren in C++,  Aug. 1994.
*************************************************************/

#define NET_UNIX        // comment out when used in PC!!!!

#ifdef  NET_UNIX
#include <termio.h>
#include <sys/socket.h>
#include <sys/endian.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <bstring.h>
#include <sys/time.h>
#define soread  read
#define sowrite write
#define soclose ::close
int kbhit();

#else   /*PC*/
   #include <time.h>
   #include <stdlib.h>
   #include <conio.h>
   extern "C" {
    #include <sys/socket.h>
   }
#endif

#ifndef _unixnet_h
#define _unixnet_h

/* For iris, the socket number starts with ~3
        the accept will creat new socket from ~7, 8
*/
#define MAX_SOCKETS   16

/*@Introduction
  Class `NetServer` is a network server object. It opens and
  close a connection socket. The port number used is from
  1024 to 4999 for a unix system (pc users may use 7-?).
  A server and a client has to be synchronized (use the same
  transfer rate). Otherwise the data will be piled up in the
  receiver buffer, which will result transfer delay.
*/

class NetServer {
  private:
    int in_use[MAX_SOCKETS];
```

```
    int socketMin;
    int socketMax;
    int firstcreat;
    int port;
    int socketOpen, wksocket;
  protected:
    void read_and_echo(int, char*, int);
    void make_new_echo(int);

  public:

      /*@Section Constractors*/
    NetServer(int);
    ~NetServer();

      /*@Class_Methods */
    int  isconnected();
    int send( char *, int);
    int recv( char *, int);
    int open();
    void close();

};


#endif /*_unixnet_h*/
```

```
/*************************************************************
  Copyright (C) Human-Machine Systems Laboratory,
          Massachusetts Institute of Technology
  File: unixnet.cpp
  Function: Ether-network connection between IRIS and PC
     Created by Shi-Ken Chen in C, 1993
     Modified by Jie Ren in C++,  Aug. 1994.
*************************************************************/

#include <stdio.h>
#include <sys/types.h>

#include "unixnet.h"


/*@Method
  Constractor:  port_num: 1024-4999 for assignment to clients.
              The clinet port number must match this
            server port number.
*/
NetServer::NetServer( int port_num)
{
   port = port_num;
   socketOpen = -1;

#ifndef NET_UNIX    // for PC only.
   if (!loaded() ) {
     fprintf(stderr," The TCP/IP protocol stack is not loaded\n");
     exit(1);
   }
#endif /*NET_UNIX*/

}

/*@Method
  if the socket is open, close it.
*/
NetServer::~NetServer()
{
   if ( socketOpen > 0 ) close();
}


/*@Method
  open a socket and bind it to clinet socket (with 'port' number).
  return: 1   - successful
        0   - failure
*/
int NetServer::open()
{
   int    x;
   struct sockaddr_in addr;

   for (x = 0; x <= MAX_SOCKETS; x++) {
     in_use[x] = 0;          // every port is free.
   }
```

```
    /* open a local socket */
    if ((socketOpen = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
      perror("socket");
      return 0;
    }

    int on=1;    /* override the socket addr if it is in use */
                 /* in PC, it is char on = 1 */
                 /* but 'char on=1' does not work with iris */
    setsockopt(socketOpen, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));

    /* binding the local socket to clinent */
    bzero ((char *)&addr, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    addr.sin_addr.s_addr = 0;

    // printf("SocketOpen=%d\t Port=%d\n",socketOpen,port);

    if (bind(socketOpen, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
      (void) close();
      perror("bind");
      return 0;
    } //else printf(" so=%d ", socketOpen);

    // To indicate that a socket is ready to listen for imcoming
    // connection requests, 5 is is currently limited by the system.
    if (listen(socketOpen, 5) < 0) {
      (void) close();
      perror("listen");
      return 0;
    }

    firstcreat = 1;  // synchronize with accept
    socketMin = socketOpen;
    socketMax = socketOpen;

    return 1;
}


/*@Method close network socket
*/
void NetServer::close()
{
  soclose(socketOpen);
  socketOpen = -1;  // indicates that the socket has been closed.
}

/*@Method
 * Test to see if the the network socket is connected
 * return  0 : wait for connection
 *        >0 : ready to receive/send socket
 */
int NetServer::isconnected()
```

```
{
   static int x,i;
   static struct timeval  seltime;
   static fd_set          readfds;

   FD_ZERO(&readfds);
   seltime.tv_sec = 0; seltime.tv_usec = 0;
   FD_SET(socketOpen, &readfds);

     // set all sockets in use for reading.
   for (x = socketMin; x <= socketMax; x++) {
     if (in_use[x]) {
         //printf(" x=%d ", x);
        FD_SET(x, &readfds);
     }
   }
   i = select(MAX_SOCKETS,&readfds, (fd_set *) 0, (fd_set *) 0,&seltime);
   /**
   if ( i <= 0 ) {    // wait for clinet's response
     //fprintf(stderr, " Select Error!");
     return 0;
   }
   **/

   wksocket = socketOpen;
         //for (x = 0; x < MAX_SOCKETS; x++) {
   for (x = socketMin; x <= socketMax; x++) {
     if (FD_ISSET(x, &readfds)) {
        if (x != socketOpen) {
         //read_and_echo(x, recvbuf, recvlen);
         wksocket = x;
         return 1;
        } else {
         make_new_echo(x);
         return 0;
        }
     }
   }
}


/*@Method
 * Read a socket into recvbuf (buffer), with
 * a length of recvlen.
 * The user provides rooms for the recvbuf!
 * return   : number of bytes received.
 *       -1: failure.
 */
int NetServer::recv( char *recvbuf, int recvlen)
{
   static int len;

   if ((len = soread(wksocket, recvbuf, recvlen)) <= 0) {
     fprintf(stderr,"\nFail to receive from socket #%d. Len=%d\n",
          wksocket, len);
     in_use[wksocket] = 0;
```

```
    return -1;
  }
  return len;
}

/*@Method
 * Send the content in sendbuf (buffer), with
 * a length of sendlen, to the client.
 * return   : the number of bytes sent.
 *       -1: failure.
 */
int NetServer::send( char *sendbuf, int sendlen)
{
  static int rc;

  rc = sowrite(wksocket, sendbuf, sendlen);
  if (rc < 0) {
      fprintf(stderr,"Fail to send by socket #%d\n", wksocket);
      in_use[wksocket] = 0;
      return -1;
  }
  return rc;

}


/*@Method
 * Make a new socket using accept()
 */
void NetServer::make_new_echo(int s)
{
  int          ns;
  struct sockaddr_in peer;
  int          peersize = sizeof(peer);

  if ((ns = accept(s, (struct sockaddr *)&peer, &peersize)) < 0) {
    fprintf(stderr,"Could not accept new connection\n");
    return;
  }

  if ( ns >= MAX_SOCKETS) {
    fprintf(stderr, "\07 Socket index exceeds maximum!\n");
  }

  if (firstcreat) {
    //socketMin = ns;
    socketMax = ns;
    firstcreat = 0;
  } else socketMax = ns;
  in_use[ns] = 1;

  fprintf(stderr,"\n Created socket #%d from %s:%d",
      ns, inet_ntoa(peer.sin_addr), htons(peer.sin_port));
}

#ifdef NET_UNIX
```

```
/*-------------------------------------------------------*/
int kbhit()   /* test if keyboard has been hit   */
/*-------------------------------------------------------*/
{
   int n;
   extern int errno;
   if ( ioctl( fileno(stdin), FIONREAD, &n ) )
     fprintf( stderr, "ioctl() returns error #%d\n", errno );
   return( n>0? n: 0 );
}
#endif /*NET_UNIX*/
```

```
/*****************************************************************
 * Copyright (C) Human-Machines Systems Laboratory, MIT, 1997
 * File: netlink.cpp (HOST program)
 *   This program creates an ethernet link between an SGI and a PC
 *   running NetClint.cpp software developed by the HMSL. This
 *   program must be started before the client!
 *****************************************************************/
*/
#include <gl/gl.h>
#include <gl/device.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <curses.h>
#include <time.h>
#include "unixnet.h"

/*================== NETWORK ========================================*/
#define ECHO_PORT    1109  /* the clinet has to use the same port#*/
#define BUFFER_SIZE  80    /* the clinet should have the same size!! */
char  buf_send[BUFFER_SIZE];
char  buf_recv[BUFFER_SIZE];
char inkey;
short val;
Device dev;

int networkSendRecv(float data1out,float data2out,
            int data3out, int data4out, float *datain);
NetServer netserv(ECHO_PORT);

int main(void)
{

/*-----------------------*/
/*  initialize variables */
/*-----------------------*/
 float k=0;
 float t=1.0;;

/*---------------------------------------*/
/* open network connection with Phantom PC */
/*---------------------------------------*/

  if ( !netserv.open()) {
    fprintf(stderr, "\07 Network is not ready!\n");
    exit(1);
  }
  else {
    fprintf(stderr, "\n\n The TCP server (%d) is ready.\n",
         ECHO_PORT);
  }

/*-----------------------------------*/
/* configure and open display window */
/*-----------------------------------*/
```

```
 prefposition(0,640,540,1023);
 winopen("input");

 color(BLACK);
 gconfig();

/*----------------*/
/* set up queuing */
/*----------------*/
 qdevice(ESCKEY);
 qdevice(KEYBD);
/*-------------------*/
/* main control loop */
/*-------------------*/
 while(1) {
  if(qtest()) {
    dev = qread(&val);
    if (dev == ESCKEY) exit(0);
    switch (val) {
        case '1':
              k--;
              break;
        case '2':
              k++;
               break;
    }
   }


/* Receive/Transfer Data and Calculate the velocity based on the time transmitted from PC.
*/
   networkSendRecv(k,1,1,1,&t);
   if(*buf_recv!='N') {
    printf("Data In from PC ==> %s    Data Out from sgi ==> %s  \n",buf_recv,buf_send);
   }

 }


return 0;
}


/* receive data from the network.
   This server sends out a string of data ONLY after receiving
   and the client operates in the same manner.
   In this way, the server and client are synchronized.

   return   -1 : when waiting for connection
         >=0 : number of bytes received.

   Note: The client is responsible to send the first string
        to kick off data transfer!
*/
int networkSendRecv(float data1out,float data2out,int data3out,
```

```
            int data4out, float *datain)
{
   int rc=-1;

   sprintf(buf_send,"%3.2f %3.2f",data1out, data2out,data3out,data4out);
   strcpy(buf_recv, "N");
   if ( netserv.isconnected() ) {
      rc = netserv.recv(buf_recv,BUFFER_SIZE);
     if ( rc > 0 ) {
        sscanf(buf_recv,"%f",datain);  /*Scan in only if data is transfered!*/
        netserv.send(buf_send,BUFFER_SIZE);
     }
     else sprintf(buf_recv," Rc = %d \n", rc);
   }
   return rc;
}
```

```
# Makefile for Client PC and SGI Host ethernet connection
SHELL = /bin/sh
FILES = Makefile netlink.cpp unixnet.cpp
OBJS = netlink.o unixnet.o
CFLAGS = -O -I/usr/include/bsd
OPTIONS =
LIBES = -L -L/usr/lib -lm -lfm_s -lc_s -lbsd -lgl_s
## LIBES = -lm -lfm_s -lgl_s -lc_s -lbsd
CC = CC
ether: $(OBJS)
    $(CC) $(OPTIONS) $(OBJS) $(LIBES) -o netlink

update:
    rm *.BAK *.CKP

cleanup:
    -rm *.o

.c.o:
    CC  $(CFLAG) -c $< -o $*.o
```

Appendix D: PC Train Sound Software

```
/*****************************************************************************/
/* Human-Machine Systems Lab High-Speed Train simulation sound enhancement
/* program. Copyright (C) Human-Machines Systems Laboratory, MIT, 1997
/* Written by Steven G. Villareal
/* Revision:
/* (1) created time base for the program          4/19/96
/* (2) added conditional statement for track component delays 4/21/96
/* (3) added mouse funcitons
/* (4) added synctrainsnd() function to replace switch command
/* (5) modified main to input host IP address from the command line  2/25/97
/* (5) modified network to get command line arguments or prompt user for host IP
/* (6) created display functions that create screen graphics 3/5/97
/* (7) added speeding/alerter waring sound 3/11/97
/*****************************************************************************/
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <dos.h>
#include <ctype.h>
#include <string.h>
#include <sys\timeb.h>
#include <math.h>
#include <stddef.h>
#include <graphics.h>
#include <netclint.h>
#include <display.h>
extern "C" {
        #include <detect.h>
        #include <smix.h>
}
#define ON  1
#define OFF 0
#define TRUE  1
#define FALSE 0

/*****************************************************************************/
#define NUMSOUNDS 10  /* Change this if you want to add more sounds        */
/*****************************************************************************/


#define KEYON_BIT               0x20
#define FM_MONO                         8
#define BANK0                   0
#define BANK1                   2
#define MIXER                           4
#define MASTERVOL_REG                                           0x22
#define FMVOL_REG               0x26
#define VOICEVOL_REG                    0x04
#define BLOCK4                  4
#define OP2                     2
#define OP4                     4
#define LEFT                    0x10
```

```
#define RIGHT                   0x20
#define BOTH                    0x30
#define CHANNEL1                  1
#define CHANNEL2                  2
#define UPDATE_TIME
        1000//[msec]
#define BRAKE_TIME
              10000//[msec]
#define L_TRACK
      3          //[m]
#define IDLE
              100
#define TRACKSOUND(indexnum,T_F) start_sound(sounds[indexnum],indexnum,T_F)
#define ENGINE_CH
                    1
#define ENGINE_INST
        0
/*Networking Definitions ************************************************/
#define ESC 27
#define ECHO_PORT 1109  //must be the same in host program
#define BUFFER_SIZE  80  //must be the same in host program


/*Function Prototypes ***************************************************/
void Ourexitproc(void);
void InitializeDigitalSound(void);
void ShutdownDigitalSound(void);
void timedelay(unsigned long clocks);
void FM_Write_Output(unsigned port, int reg, int val);
void FMout_Mono(int reg, int val);
void WriteRegValBank1(int reg, int val);
void WriteRegValBank0(int reg, int val);
void Mixer_Control(int reg, int val);
void InitializeFMSound(void);
void Initialize_Sound_Timbre_2op_Mode(int ch, int inst_num, int L_R_B);
int  GenerateSound(int ch,int fn, int block, int inst_num, int L_R_B);
void StopFMSound(int ch, int freqnum);
void DeinitializeFMSound(void);
void GenerateDisplay(void);

void SyncTrackSound(int &i);

void DisplayFrequency(int fn);
unsigned long GetRealTime(void);
unsigned long SetProgTime (unsigned long current_prog_clock);
unsigned long GetProgTime(void);
void TimeDelay (unsigned int num);
void InitializeNetwork(NetClient &enet);
void NetSendRecv(NetClient &enet, float dataout, float *data1in, float *data2in,
              int *data3in, int *data4in, int *data5in, int *data6in);


/*Global Variables*******************************************************/

/*FM variables*/
unsigned int bell_freq = 100, block = 4, inst_num;
```

```
int engpower = IDLE;
int Offset[]= // Array of modulator register
        {       // offsets for various channels
        0x00,  // on the FM chip.  Carrier
        0x01,  // registers are always the
        0x02,  // modulator register plus three
        0x08,  // Exs.
        0x09,  // Channel Modulator Carrier
        0x0A,  //  1      20h     23h
        0x10,  //  2      21h     24h
        0x11,  //  3      22h     25h
        0x12   //  4      28h         2Bh
        };     //  5      29h         2Ch

/* instrument table definition */
int inst[128][23] =
{
        /********************** Engine ***************************/
        { 0x61, 0x70, 0x68, 0x00, 0xF2, 0x52, 0x0B, 0x0B,
          0x00, 0x00, 0x0A, 0x00, 0x00, 0x00, 0x00, 0x00 },


         /********************** Bell Sound **********************/
        { 0x07, 0x12, 0x4f, 0x00, 0xF2, 0xF2, 0x60, 0x72,
          0x00, 0x00, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00 },
};


/*Digital Sound Variables*/
char *resource_file = "train.snd"; //this is the sound library file name!
int baseio, irq, dma, dma16;
long counter;
char inkey;
int  num;
int  temp;
double trackdelay[5] = {  2, 181, 214, 120, 228 }, del[5];
double del1, del2, del3, del4, del11, t,t1,t0,timer0,brake_end_time;
float velocity, jcmd;
char *sound_key[NUMSOUNDS] =
        {
         "trackc1",
         "trackc2",
         "trackc3",
         "trackc4",
         "horn",
         "idle",
         "steam",
         "alarm_loop",
         "speeding",
         "braking"
        };

SOUNDS *sounds[NUMSOUNDS];

/*Networking Global Variables*/
int host, len;
```

```
char buffer_recv[BUFFER_SIZE], buffer_send[BUFFER_SIZE];


int main(int argc, char *argv[])
{
 /* Flags and Counters */
 int speedflag = FALSE, alerterflag = FALSE, hornflag = FALSE, bellflag = FALSE;
 int trackflag = FALSE, brakeflag = FALSE, stopbrakeflag = FALSE;
 int i = 1, k=0, stop = FALSE;
 int bell_inst = 1, bell_ch = 2 ;
 HostIP        ipaddress(argv[1]);          /* Establish Host ID */
 NetClient enet(ipaddress.returnIP());  /* Creat NetClint Object */

/* Sound Card & Network Initialization and Screen Display Generation **********/
 InitializeNetwork(enet);
 InitializeDigitalSound();
 InitializeFMSound();  InitializeGraphics();
 GenerateDisplay();
 /*****************************************************************************/

 start_sound(sounds[5],5,TRUE);
 GenerateSound(ENGINE_CH,engpower,block,ENGINE_INST,BOTH);
 SetProgTime(0);          //Intialize the program clock to zero.
 t0 = GetProgTime();
 timer0 = t0;
 /* Main Control Loop*/
 do {
                if (velocity <= 10.0) {  //read net faster when train is going slow
                    NetSendRecv(enet,1e-3*GetProgTime(), &velocity, &jcmd,
                          &speedflag, &alerterflag, &hornflag, &bellflag);
                }

                gotoxy(5,7); clreol();
                gotoxy(5,7);
                cout << "Buffersend:  "<<buffer_send
                     << " Bufferrecv:  "<<buffer_recv
                     << endl;

        if ( hornflag ) { start_sound(sounds[4],4,FALSE);}

        if ( bellflag ) {
                 StopFMSound(bell_ch,bell_freq);
                 GenerateSound(bell_ch,bell_freq,4,bell_inst,BOTH);
                 }


 /* Speeding and Alerter Flags go here*/
        if( speedflag ) {
                if(!sound_playing(8)){
                    start_sound(sounds[8],8,TRUE);
                }
        }
        else {stop_sound(8);}

        if( alerterflag ) {
```

```
            if(!sound_playing(7)){  //if alerter snd is NOT playing do this.
                start_sound(sounds[7],7,TRUE);  //Play alerter sound
            }
        }
        else {stop_sound(7);}


        if (kbhit()) {        //If a key is hit, get that key and check the flags.
            inkey = getch();
                if ((inkey == 'b')) {
StopFMSound(bell_ch,bell_freq);GenerateSound(bell_ch,bell_freq,4,bell_inst,BOTH); }
                if ((inkey == '1')) start_sound(sounds[4],4,FALSE);  //Play the horn sound.
                if ((inkey == '2')) start_sound(sounds[8],8,FALSE);  //Play alerter sound
                if ((inkey == 'q')) stop = TRUE;    //exit the demo
        }


/* Adjust Engine Power accoding to Joystick Position */
            if (jcmd  > 0 ) {stop_sound(9); brakeflag = TRUE;} //

            if( ((jcmd) < 0) && (velocity > 0.5) ) { //play the air brake sound if jcmd is negative
                if (brakeflag) {
                  //and the train is moving.
                        start_sound(sounds[6],6,FALSE);
                        start_sound(sounds[9],9,TRUE);
                        brake_end_time = GetProgTime() + BRAKE_TIME;
                        brakeflag = FALSE;
                        stopbrakeflag = TRUE;
                }
            }

            if ( stopbrakeflag ) { //Controls screeching brake sound time length
                if( brake_end_time <= GetProgTime()  || velocity ==0) {
                        gotoxy(23,3);clreol();cout<<"TIME FINISHED "<<endl;
                        stop_sound(9); stopbrakeflag = FALSE;
                }
                else { gotoxy(23,3);clreol();cout<<"TIME still LEFT!"<<endl; }
            }

            if( (10*jcmd+IDLE) > engpower){ //increment engpower if powerin is larger
                if(engpower >= IDLE) {  //don't increment engpower ABOVE IDLE
                        engpower+=2;
                        GenerateSound(ENGINE_CH,engpower,block,ENGINE_INST,BOTH);
                        DisplayFrequency(engpower);
                }
                else engpower = IDLE;
                if (!trackflag){TimeDelay(50);} //avoid fast freq increase when sound is not
playing.
            }

            if( (10*jcmd+IDLE) < engpower) { //decrement engpower if powerin is smaller
                if(engpower >= IDLE) {   //don't decrement engpower BELOW IDLE
                        engpower-=2;
                        GenerateSound(ENGINE_CH,engpower,block,ENGINE_INST,BOTH);
                        DisplayFrequency(engpower);
                        }
```

```
                else engpower = IDLE;
                if (!trackflag){TimeDelay(40);} //avoid fast freq increase when sound not playing.
            }


/* Play the Track sound component */
            if (trackflag) {
                gotoxy(23,2);clreol();cout<<i<<endl;
                if (i == 5 && velocity > 10){
                        NetSendRecv(enet,1e-3*GetProgTime(), &velocity, &jcmd,
                                &speedflag, &alerterflag, &hornflag, &bellflag);
                }
                if (i == 3 && (velocity > 10.0 && velocity <= 70.0  ) ) {
                        NetSendRecv(enet,1e-3*GetProgTime(), &velocity, &jcmd,
                                &speedflag, &alerterflag, &hornflag, &bellflag);
                }

                SyncTrackSound(i);
            }
            else {
                NetSendRecv(enet,1e-3*GetProgTime(), &velocity, &jcmd,
                        &speedflag, &alerterflag, &hornflag, &bellflag); /*Get network info */
            }



    t1 = GetProgTime() - t0;    //t1 is the time that has elapsed since the beginning of the loop

    /* UPDATE INTERRUPT ************************************************************/
    if(t1 > UPDATE_TIME){
            t = 1e-3*GetProgTime();
            if (velocity < 0.125 ) velocity = 0.125;
            if ((del11 = L_TRACK*1000.0/velocity) > 2000) {
                trackflag = FALSE; }//Want to avoid calc. an infinit delay time.
            else {/* Calculate track component sound delays */
                trackflag = TRUE;
                if (velocity < 5) {
                        for (k = 1; k<5;k++)        del[k] = 0.75*trackdelay[k];
                }
                else {
                        for (k = 1;k<5;k++)
                        del[k] = 0.75*trackdelay[k]/1000.0*(del11-39.0)+0.45*trackdelay[k];
                        }
            }

            t0= GetProgTime();                  //Initialize the loop time before exiting.
    /* END INTERRUPT ***************************************************************/
    }


 } while (!stop);


StopFMSound(ENGINE_CH,engpower);
StopFMSound(bell_ch,bell_freq);
stop_sound(6);
```

```
 ShutdownDigitalSound();
 DeinitializeFMSound();
 closegraph();              /* Return the system to text mode */
 return(EXIT_SUCCESS);
}

/*Frequency Modulation Functions*/
/*****************************************************************************/
void timedelay(unsigned long clocks)
{
        unsigned long elapsed=0;
        unsigned int last, next, ncopy;

        outp(0x43, 0);
        last=inp(0x40);
        last=~((inp(0x40)<<8) + last);
        do
                {
                outp(0x43, 0);
                next=inp(0x40);
                ncopy=next=~((inp(0x40)<<8) + next);
                next-=last;
                elapsed+=next;
                last=ncopy;
                }
        while (elapsed<clocks);
}

void FM_Write_Output(unsigned port, int reg, int val)
{
        outp(port, reg);
        timedelay(8);          // delay about 3.3 microseconds
        outp(port+1, val);
        timedelay(55);         // delay about 23 microseconds
}

void FMout_Mono(int reg, int val)
{
        FM_Write_Output(baseio+FM_MONO, reg, val);
}

void WriteRegValBank0(int reg, int val)
{
        FM_Write_Output(baseio+BANK0, reg, val);
}

void WriteRegValBank1(int reg, int val)
{
        FM_Write_Output(baseio+BANK1, reg, val);
}

void Mixer_Control(int reg, int val)
{
        FM_Write_Output(baseio+MIXER, reg, val);
}
```

```
void InitializeFMSound(void)
{
        /*Card Initialization*/
        WriteRegValBank0(1,0);              //Initialize the card
        WriteRegValBank1(5,1);              //Get into OPL-3 Mode
        WriteRegValBank0(8,0);              //Setup FM mode
        WriteRegValBank0(0xBD, 0);          //Setup FM mode

        /* Mixer Control Initialization */
        Mixer_Control(MASTERVOL_REG, 0xEE);  //Set master volume to 14
        Mixer_Control(FMVOL_REG, 0xBB);      //Set FM volume to 9
        Mixer_Control(VOICEVOL_REG, 0xEE);   //Set FM volume to 13
}

void StopFMSound(int ch, int freqnum)
/* this function only changes bit 6; the keyon bit. This eliminates a
* clicking noise caused by changing the other bits in that register.
* The function arguments are the current channel flag and frequency number.
*/
{
        WriteRegValBank0(0xB0+(ch-1), (( ( (freqnum>>8) & 0x3) + (block << 2)) & 0xDF ));
}



void DeinitializeFMSound(void)
{
        int i;
        WriteRegValBank1(5,0);              //Reset chip to OPL-2 mode
        for(i=1;i<=9;i++){
                StopFMSound(i,engpower);
        }
        Mixer_Control(MASTERVOL_REG, 0xAA);     //Lower the master volume
//      _setcursortype(_NORMALCURSOR);
}



void Initialize_Sound_Timbre_2op_Mode(int ch, int inst_num, int L_R_B)
{
        WriteRegValBank0(0x20+Offset[ch - 1], inst[inst_num][0]);
        WriteRegValBank0(0x23+Offset[ch - 1], inst[inst_num][1]);
        WriteRegValBank0(0x40+Offset[ch - 1], inst[inst_num][2]);
        WriteRegValBank0(0x43+Offset[ch - 1], inst[inst_num][3]);
        WriteRegValBank0(0x60+Offset[ch - 1], inst[inst_num][4]);
        WriteRegValBank0(0x63+Offset[ch - 1], inst[inst_num][5]);
        WriteRegValBank0(0x80+Offset[ch - 1], inst[inst_num][6]);
        WriteRegValBank0(0x83+Offset[ch - 1], inst[inst_num][7]);
        WriteRegValBank0(0xE0+Offset[ch - 1], inst[inst_num][8]);
        WriteRegValBank0(0xE3+Offset[ch - 1], inst[inst_num][9]);
        WriteRegValBank0(0xC1, L_R_B | (inst[inst_num][10] & 0xF));
}

int GenerateSound(int ch, int fn, int block, int inst_num, int L_R_B)
/* Start an new FM instrument with frequency fn and block on the L_R_B speaker
```

```
/* side. */
{
        Initialize_Sound_Timbre_2op_Mode(ch, inst_num, L_R_B);
        WriteRegValBank0(0xA0+((ch)-1), (fn & 0xFF));
        WriteRegValBank0(0xB0+((ch)-1), ((fn>>8) & 0x3) + ((block << 2) | KEYON_BIT));
        return 0;
}

void SyncTrackSound(int &i)
/* Coordinates the track sound components according to the value of i. */
        {
                if (i == 5){
                    TimeDelay(del11);   //play the delay between total events
                    i = 1;              //reset back to 1st component
                }
                else {
                    TRACKSOUND(i-1,FALSE);
                    TimeDelay(del[i]);
         //         TimeDelay(100); //checking the network bandwidth...
                    i++;                //increment to next component
                }
        }

/*Digital Sound Playback Functions*/
/*****************************************************************************/
/*      SMIXC is Copyright 1995 by Ethan Brodsky.  All rights reserved       */
/*      Modified 3/20/96 by Steven G. Villareal, Human-Machine Systems Lab   */
/* ***************************************************************************/
void ourexitproc(void)
{
        int i;

        for (i=0; i < NUMSOUNDS; ++i)
                if (sounds[i] != NULL) free_sound(sounds+i);
}

void InitializeDigitalSound(void)
{
        int i;
        if (!detect_settings(&baseio, &irq, &dma, &dma16)) //Get Blaster Environment
                {
                 printf("ERROR:  Invalid or non-existant BLASTER environment variable!\n");
                 exit(EXIT_FAILURE);
                }
        else
                {
                 if (!init_sb(baseio, irq, dma, dma16))
                    {
                        printf("ERROR:  Error initializing sound card!\n");
                        exit(EXIT_FAILURE);
                    }
                }
        printf("DSP version %.2f:  ", dspversion);
        if (sixteenbit)
                printf("16-bit, ");
        else
```

```
            printf("8-bit, ");
        if (autoinit)
            printf("Auto-initialized\n");
        else
            printf("Single-cycle\n");
        if (!init_xms())
            {
                printf("ERROR:  Can not initialize extended memory\n");
              printf("HIMEM.SYS must be installed\n");
              exit(EXIT_FAILURE);
            }
        else
            {
                printf("Extended memory successfully initialized\n");
                printf("Free XMS memory: %uk    ", getfreexms());
                if (!getfreexms())
                 {
                        printf("ERROR:  Insufficient free XMS\n");
                        exit(EXIT_FAILURE);
                 }
                else
                 {
                        printf("Loading sounds\n");
                        open_sound_resource_file(resource_file);

                        for (i=0; i < NUMSOUNDS; i++)
                                load_sound(&(sounds[i]), sound_key[i]);
                        atexit(ourexitproc);

                        close_sound_resource_file();
                 }
            }
        init_mixing();
        printf("\n");
}


void ShutdownDigitalSound(void)
{
        int i;

        shutdown_mixing();
        shutdown_sb();

        for (i=0; i < NUMSOUNDS; i++)
            free_sound(sounds+i);
        printf("\n");
}




/* Program Time Base Functions ************************************************/
/* The following is modified code originally written by Ed Lanzilotta       */
```

```
/* 4/18/96
/*****************************************************************************/

static unsigned long prog_offset = 0;
static int prog_time_set = FALSE;

unsigned long GetRealTime(void)
/* Get the current real time from the CPU time base. */
{
        static unsigned long rt;        //This value will represent milliseconds
        struct timeb current_real_time;
        ftime(&current_real_time);    //Get the current real time
        rt = (current_real_time.time*1000) + current_real_time.millitm;

        return(rt);  //Return the real time clock value.
}

unsigned long SetProgTime (unsigned long current_prog_clock)
/* Initializes the program clock to the value of "current_prog_clock".
/* Returns the program time. */

{
        unsigned long prog_time;
        //      unsigned long current_prog_clock;

        prog_offset = GetRealTime() - current_prog_clock;
        prog_time = GetRealTime() - prog_offset;
        prog_time_set = TRUE;

        return(prog_time);
}

unsigned long GetProgTime(void)
/* This returns the value of the current program time. */
{
        unsigned long prog_time;
        if (prog_time_set)
                prog_time = GetRealTime() - prog_offset;
        else
          prog_time = SetProgTime(0);//Initialize the time set_prog_time has not been called yet
        return(prog_time);
}

void TimeDelay (unsigned int num)
/* Delays the program execution for the 'num' milliseconds. */
//unsigned int num;
{
        unsigned int end_time;
        int time_left;

        /* Calculate the end time */
        end_time = GetRealTime() + num;
        time_left = num;
        while (time_left > 0 )
                time_left = end_time - GetRealTime();
}
```

```
/*Network Functions */

void InitializeNetwork(NetClient &enet)
{
        if (!enet.open(ECHO_PORT)) {
                printf("couldn't open network connection\n");
                exit(1);
        }
}

void NetSendRecv(NetClient &enet, float dataout, float *data1in,
                                        float *data2in, int *data3in, int *data4in,
                                        int *data5in, int *data6in)
{
        sprintf(buffer_send,"%3.2f",dataout); //send argument variable

        if(enet.iswriteready()) {
                enet.send((char *)buffer_send, BUFFER_SIZE); //Data sent out of the pc
        }

        if(enet.isreadready()) {
                len=enet.recv((char *)buffer_recv,BUFFER_SIZE);
                sscanf(buffer_recv,"%f %f %d %d %d %d",data1in,data2in,data3in,data4in,

                                                                data5in,data6in); //Data received
from sgi
                if(len<=0) {
                    printf("\n zero receiving length");
                    ShutdownDigitalSound();
                    DeinitializeFMSound();
                    closegraph();           /* Return the system to text mode      */
                    exit(1);
                }
        }

}
```

REFERENCES

[1] Bowsher, J. M. The Physics of Music. New York: John Wiley & Sons, Inc., 1975.

[2] Creative Labs, Inc. The Developer Kit for Sound Blaster Series: Development Tools for Adding Sound to DOS Applications. Creative Labs, Inc., 1991.

[3] Hund, August. Frequency Modulation. 1st ed., New York and London: McGraw Hill Book Company, Inc., 1942.

[4] Jeans, Sir James. Science and Music. New York: The Macmillan Company, 1937.

[5] Jones, Edward R., et al. Human Factors Aspects of Simulation. Washington D.C.: National Academy Press, 1985.

[6] Josephs, Jess J. The Physics of Musical Sound. NY: Van Nostrand Reinhold Company, 1967.

[7] Messmer, Has-Peter. The Indispensable PC Hardware Handbook. New York: Addison-Wesley Publishing Company, 1995.

[8] Novell®. LAN WorkPlace® for DOS Administrator's Guide. ,1992.

[9] Volkmann, J; and Stevens, S. Acoustic Society Am.. 8, pg. 185-190 ,1937.

[10] Winckel, Fritz. Music, Sound, and Sensation: New York: Dover Publications, Inc. ,1967.

[11] Yamaha®. YMF262 Application Manual. Cat No.: LSI-6MF262A4, 1994.