Touch Screen Sound Controller

Senior Project

Electrical Engineering 2010

California Polytechnic State University

San Luis Obispo

By: James Fenley and Jonathan Law

Advisor: Dr. Fred DePiero

Date: June 11, 2010

This page is intentionally left blank

Table of Contents

Section ***Page***

List of Tables and Figures…… II

Acknowledgements…… III

Abstract…….. VI

1. Introduction…….. 1
2. Requirements…….. 2
3. Design……. 3
4. Construction & Testing…… 6
5. Hardware Implementation…… 11
6. Software Implementation…… 17
7. Dictionary of Terms…….. 20
8. Bibliography…….. 22
9. Conclusions & Future Modifications…… 23

Appendices

A. Parts List & Prices…… 24
B. Code for decoding .mp3 to .wav file…… 25
C. Code for writing a .wav file via server/client connection…….. 29
D. PCMout Program for outputting PCM data to a speaker…… 41
E. Final Code for Server/Client Programs and Touch Screen..... 46

List of Tables and Figures

Figures ***Page***

1. Figure 1: Software Control Flow…….. 5
2. Figure 2: Software Control Flow without Serial Input Device…….. 8
3. Figure 3: Block Diagram of Digital Echo Filter…….. 10
4. Figure 4: Final Software Control Flow in terms of Finished Program Names… 10
5. Figure 5: Olimex Development Board…….. 11
6. Figure 6: RealTerm Session, Example Downloading and Running a Program 12
7. Figure 7: Touch Screen Serial Connection Diagram…….. 13
8. Figure 8: SLCD43 Board…….. 14
9. Figure 9: POWERCOM4 Board…….. 14
10. Figure 10: Example Session of BMPLoad…….. 16

Acknowledgements

This project proved the most challenging in our academic career. We would like to take this opportunity to thank:

- Dr. Fred DePiero for his support in this project and for his genuine interest in our careers.
- John Oliver for his helpful suggestions after we hit a dead end in terms of our choice of microcontrollers for this project.
- The entire Electrical Engineering faculty; we are forever grateful for the steady patience and demand for engineering excellence that no doubt provided us with the invaluable skills needed to thrive in this industry.
- Reach Technology, Inc. for providing us with a free touch screen for the sake of bettering education.

And finally we would like to thank our parents and friends back home who motivated us to pursue our goals with persistence and determination.

Abstract

The Touch Screen Sound Controller sets out to explore the use of a touch screen as a sound control interface for the disk jockey profession. In addition, the project aims to provide an alternate means of transferring audio data by way of TCP/IP communications as opposed to MIDI. By applying our method, a user may stream pulse-code modulated data from a server onto a client's RAM via an Ethernet connection. A 32bit, 200MHz ARM9 microprocessor addresses data from the RAM and proceeds with executing DSP instructions from the user. The connection between the touch screen and the central microprocessor is a standard RS-232 serial cable following a UART communications protocol. Applying touch commands steers the audio to the user's needs.

1. Introduction

The demand for touch screen interfaces in electronic devices experienced a dramatic increase over the last decade. From smart phones like the iPhone(TM) to even ATMs and movie rental kiosks, it is hard to go a day without interacting with some sort of touch screen device.

Current technological trends predict the use of touch screens in the electronic music performer's tool kit. This fact and the continued demand for faster and easier communications and control interfaces for the performers on stage inspired the creation of this project. By using an embedded LCD touch screen device with programmable Flash memory, the user may create a custom interface of their choosing. This adds flexibility to how the DJ wishes to set up the performance. In addition, this system seeks to exploit the speed of an Ethernet connection for establishing data links for the performer to access audio data.

2. Requirements

The touch screen sound controller should provide the user with a basic menu to apply simple sound control such as play/pause and volume control. In addition the user should also have the ability to apply audio filtering effects.

The specific project requirements for the touch screen sound controller are:

1. Develop an interface for the user to interact with on the touch screen.
2. Manage the RS-232 UART serial data transmission between the touch screen LCD and the Olimex development board for sending touch commands.
3. Decode .mp3 files to Pulse Code Modulated data for digital signal processing.
4. Develop a web client / server for downloading decoded .mp3 file data to the Olimex flash memory via a TCP/IP connection.

3. Design

The system for the Touch Screen Sound Controller requires fast immediate communication between multiple arbitrary sources of PCM data to the controller and a touch screen method of input local or close to the controller. This paradigm allowed us to select two separate communications systems for interfacing; Ethernet and related Internet Protocols for distant multiplexed communication from audio sources and serial from the touch device used as control inputs to the controller itself. In order to quickly and inexpensively implement these systems we chose the Linux environment with open source APIs for programming all hardware devices. We applied Linux through a virtual machine called Virtual Box as a guest operating system to Windows 7. This eased testing by allowing us to utilize the software already natively available on our laptop through Windows 7 such as Windows Media Player while still providing the programmer friendly Linux environment.

First we illustrate why we chose Ethernet and the Internet as an interface. Beginning in 1982 MIDI (Musical Instrument Digital Interface) became industry standard for audio control. MIDI operates by sending character data through a MIDI connector to different audio devices for interpretation. This revolutionized digital audio production at the time because small amounts of data could be sent over MIDI connections to MIDI devices and the MIDI compatible devices could interpret the data and perform audio synthesis. This requires less data overhead (during a time of limited hardware memory) since rather than sending fully fledged audio the MIDI interfaces only needs to communicate through minimum code sizes to specify musical elements such as timbre and pitch. However this comes with its drawbacks, MIDI controllers can only send and interpret MIDI data, not audio data, and every endpoint before an analog to digital conversion requires its own synthesizer for interpretation of the MIDI data and to output. The time it takes to construct raw audio data from MIDI data causes lag and hardware overhead since every endpoint in the system requires a synthesizer. Additionally, all equipment communicating via MIDI requires a MIDI connector; a connector unfamiliar to any other type of electronic device **(Hass)**.

After researching the latest technology in audio manipulation, the team discovered a push from MIDI to Internet Protocol to transfer sound information. Considering the incredibly flexible nature of the Internet and its extremely high speeds this seemed like the obvious choice. Using Ethernet and an Internet Protocol would allow the hardware to exchange raw PCM data allowing audio information to be derived from any source without additional synthesizers. Additionally, the controllers could perform true digital signal processing rather than manipulate a set of code criteria. The combination of these two facts would allow such a controller to manipulate audio in a more natural and flexible manner and reduce lag as well as hardware overhead by making the most use of all hardware devices.

After the decision to utilize an Ethernet connection with an IP protocol for audio transmission we agreed upon using TCP/IP as our mode of software transport. TCP/IP has three main benefits; first it guarantees error free transport of all data, second it performs automatic flow control over the connection, and lastly most modern software applications utilize TCP/IP for any Internet related data transfer opening the system up to a large realm of creative opportunities. **(Larry and Bruce 474-525)**

Next we will discuss why we chose a serial connection for the control input devices interface to the main microcontroller. Serial UART interfaces may transfer character data asynchronously between devices. Serial is cheap and easy to program. Given the short distance the touch screen controller input data would need to travel to reach the main controller interface Serial seemed like the right choice. Additionally, most computing devices contain some sort of serial connection and almost all input devices use some form of serial UART interface **(Hankins and Layer)**. By using serial to route the control inputs the device is open to a wide range of possibilities for input device type, not limited to our touch screen. Even MIDI connectors are based off of a serial connection. With minimal configuration the host controller could receive input from other types of devices in order to control audio data.

Extending the discussion of Internet Protocol for PCM data transmission we will now illustrate how to implement communication and congestion control between the audio source and the controller. Since TCP/IP employs a server/client model for communications we followed suit and chose the audio source as an audio server and the controller as an audio client. The audio server must perform two functions, first it must create a service which streams pre-decoded audio in PCM form to a connected client and then must receive the manipulated audio from the client and output to a speaker system. We chose this design to allow for greater versatility when choosing both the source of audio and the controller. By allowing the source of audio to be a server, this source could potentially connect to as many controller devices as possible, limited only by connection speed. Likewise, making the controller a client would allow any device following the TCP/IP protocol to act as a controller, not limiting the system to the microcontroller and input device we specifically chose. *Beej's Guide to Network Programming* is a good starting point for anyone willing to program a TCP/IP client or server in the Linux environment **(Hall)**.

For our final hardware design choices we picked a Gateway laptop with Linux running as an operating system for the server. This connects to an Olimex Development Board for the audio client/controller via Ethernet/TCP/IP. Finally we selected a donated Reach Touch Screen which sends control inputs to the main controller via Serial UART.

In order to test our design we needed both a method of inputting PCM data to the server and outputting PCM data from the server to a speaker. For input we chose to decode .mp3

files given their substantial market penetration and because we have a large collection of .mp3 files already stored on our laptop. To write our decoder we used the open source mpg123 library. The mpg123 library benefits as being the first open source mpeg decoding library ever built for Linux systems making it very stable and the library has substantial documentation **(Hipp & Orgis)**. For output we chose to employ our laptop's speakers. In order to properly convert the audio we used the open source ALSA library. The ALSA library has similar benefits to the mpg123 library. It is open source, though earlier audio output libraries did exist it remains as one of the first Linux audio output libraries, it may be implemented to perform ADC on just about any sound card in the market making it flexible and guaranteed to work with our laptop, and the creators provide substantial documentation **(Nagorni)**. In essence, our laptop would act as both the decoder for our .mp3 files, the server, the ADC, and speaker system. *Figure 1* illustrates the final design flow.

For further explanations of the hardware, software, and libraries chosen consult the appendix section titled….

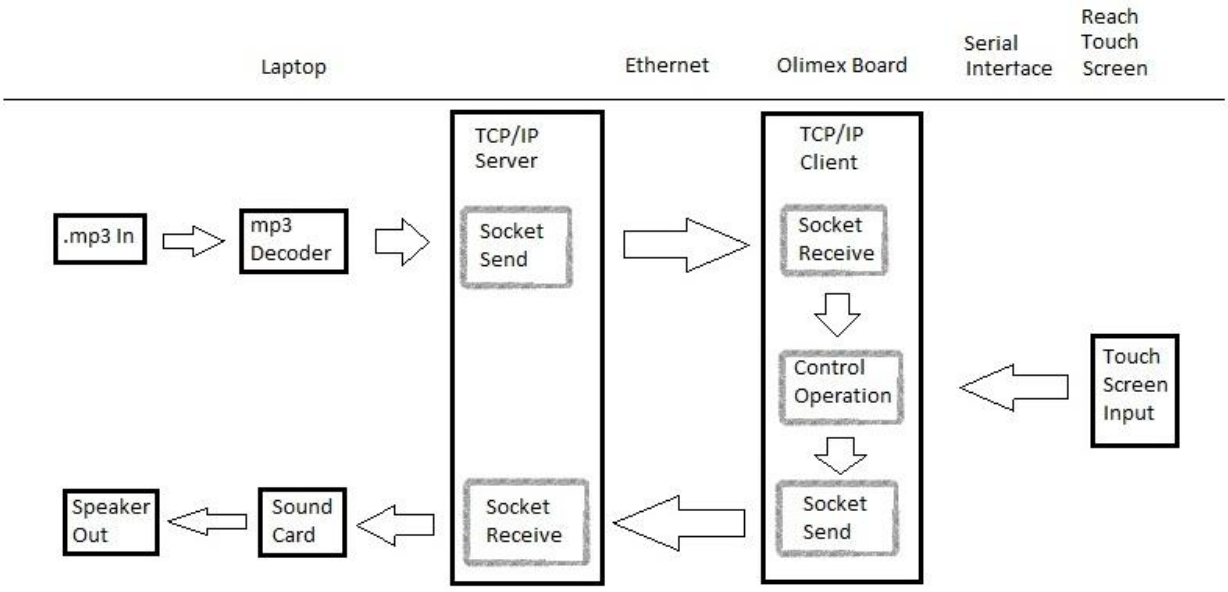


Figure 1: *Software control flow. The laptop provides the decoder, server, and audio output. Ethernet bridges the laptop to the Olimex Board. The Olimex Board provides the client and control operations. Serial interfaces the Reach Touch Screen and Olimex board. The Reach Touch Screen provides input commands to the controller.*

4. Construction & Testing

This project experiments with interfacing, congestion control, and signal processing between multiple devices; together termed a communications system. Since a communication system suggests moving information in one form from one point in a linear manner to another point, the test plan includes a modular methodology of building one block of the design, starting from the first block in the design where the input is injected, testing this block for an adequate output and then moving on to configure the next block and so forth until completing the final block and final output is achieved. We advise putting all compiled programs in the same directory.

Begin by installing Linux. Linux may be used as the main operating system or as a guest OS in a virtual machine such as Virtual Box. We used Virtual Box, created a virtual machine with 384 Mb of base physical memory, 100 Mb of video memory, AC'97 sound, a bridged (through Windows 7) Ethernet connection, and Ubuntu 10.4 as our OS. After properly configuring the system, download and install the sndfile library, mpg123 library, ALSA library, arm-linux-gcc-4.1.1 compiler, apache2 server, and minicom. These programs and libraries can all easily be found through a search engine. Installing the libraries/programs usually involves unzipping the file downloaded, opening a terminal, within the terminal changing to the directory where the files were unzipped to, and typing "./configure", "sudo make", and "sudo make install" to install. Always consult the README or INSTALL file first before trying to install these files. This should set up the basic infrastructure for construction and testing.

Next, start by building and testing the mp3 decoder. Utilizing the functionality of the mpg123 library consult the documentation and observe the example source code provided under LGPL by the creators of mpg123 to develop code. Compile using the normal gcc(1) compiler and name the program "./mp3decode". First experiment by simply converting an .mp3 file into raw PCM data, stereo, interleaved, using signed 16-bit integers at 44.1 kHz. In order to test correct conversion of the data one must output the PCM data to the .wav format. A .wav file consists of a short header describing the necessary information for playing the data which includes audio format (1 for PCM), number of channels, number of bytes per sample including all channels, number of bits per sample, and the audio data length followed by the actual PCM data. The sndfile library has built in functionality for writing .wav headers onto files. We used this functionality, however, the header is easy to create and the engineer may make one and append it the beginning of the PCM raw data file without using this library. One may then employ Windows Media Player or a similar player to play back the data output and then use the same player to play the .mp3 file to compare the sonic quality of the decoder. The code should look something similar to the code listed in **Appendix B**.

After confirming the mp3 decoder's ability to properly output decompressed PCM data, begin working on the server and client. Follow *Beej's Guide to Network Programming* to develop a simple server and client **(Hall)**. The server should simply wait for a client to connect and then immediately send "Hello World" which the client then outputs to the terminal. After confirming this programs ability to communicate with the client, combine the decoder and server into a single program where the server reads .wav data outputted from the decoder and sends it to the client. The client then writes the data to a .wav file. Name the server program "./server" and the client program "./client" when compiling. Playing the .wav file via Windows Media Player confirms that the file transferred correctly. We performed all communications locally on our laptop via the loopback IP address 127.0.0.1 to ease debugging. After completion test the results using the same process as before, by simply playing the data via Windows Media Player. The network should transfer the data with no loss. Applying Linux's diff(2) program can check for any differences between the .wav file written in the previous section to this .wav file to confirm that no data loss occurs. Refer to **Appendix C** for example code.

The next step requires creating and testing a system of playing raw PCM data through the speakers. Consult the documentation for the ALSA library and develop code which can configure the laptops sound card for playback at a specified sampling rate, number of channels, and data format. Name this program "./PCMout" when compiling. This also requires reconfiguring the original .mp3 decoder to output to PCM rather than .wav format. The resulting playback program should read from a file generated by the decoder and play the information through the speakers. The playback should sound comparable to that of Windows Media Player. Refer to **Appendix D** for example code.

With the playback mechanism correctly instantiated, pursue unifying all the systems generated into a flow control similar to that of *Figure 1* excluding the input device. Modify the server and client so the client re-routes the PCM data sent from the server back to the server. Set up the server to subsequently route the PCM data to the PCMout program as shown in *Figure 2***.** This requires modifying the server to spawn an additional process per connection that calls a Linux execution functions such as execlp(3) pipes the received PCM data from the TCP/IP socket to the program ./PCMout. Additionally, compile the client using the arm-linux-gcc-4.1.1 compiler and port it to the Olimex Board. First this requires a method to communicate to the microcontroller. Linux's minicom program or RealTerm which may be downloaded from the web has built in functionality to send and receive data via serial connected devices. The Olimex Board automatically outputs its terminal information and accepts inputs at the Linux terminal prompt from RS232_0 as shown in **Hardware Implementation**. Set minicom to communicate at 57600 Baud. Additionally set minicom to 8 data bits, no parity, and 1 stop bit, also known as 8N1. A Null Modem connector may be required when communicating between the laptop and the board; this

depends on the laptop's serial connection type. Next the board requires a method of transferring the client program to the board's on-board memory. Since the board comes pre-installed with Linux, use Linux's wget(1) function to download the program. Downloading requires a web server which can host files for transferring to the microcontroller board. We chose to use apache2(8) to create the web server on the laptop. The engineer may choose a different web server program or decide to create their own. Additionally, compiled programs may be transferred via minicom over serial however we advise against this due to the slow transfer rate and greater chance of a data transfer error for large files. After successfully loading the client program to the board, run ./server (which should automatically generate a pipe(2) and apply execlp(3) to ./PCMout) on the Linux terminal on the laptop and ./client on the Olimex board to run the program.

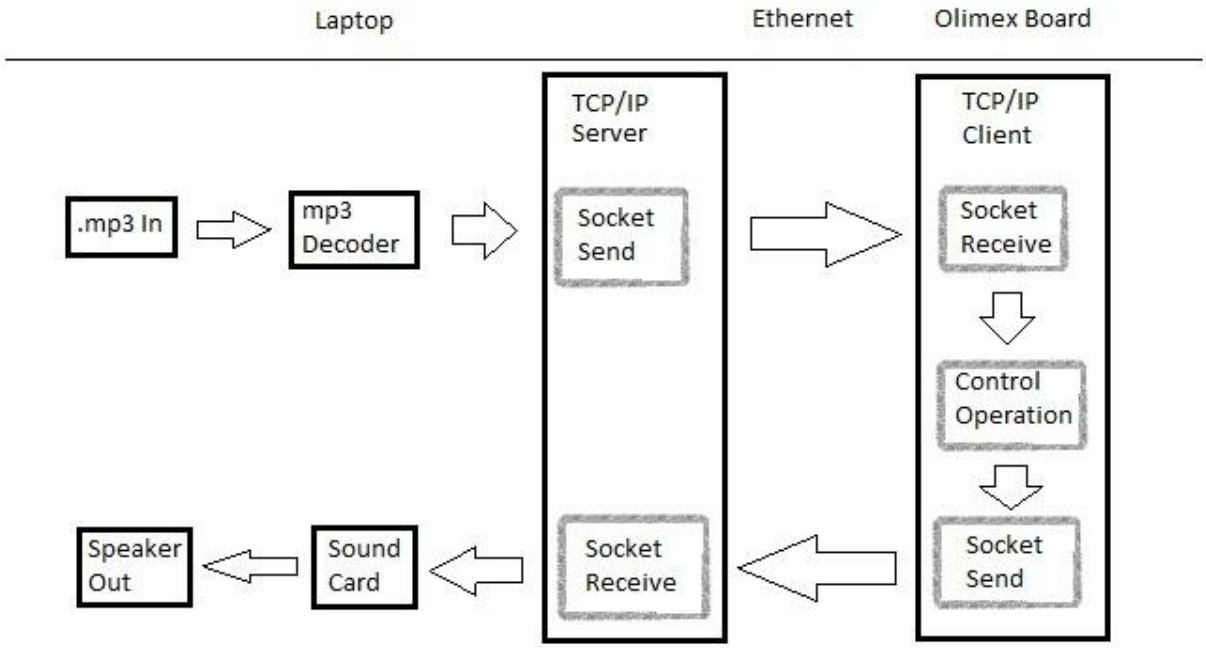


Figure 2: *Software control flow without the input device for ease of testing.*

Lastly the system needs an input device, in our case the touch screen, to send audio control commands to the client. In order to connect the touch screen to the Olimex board via a serial connection, add code to the client which alters the client to open the local serial device and receive commands from the touch screen through this device. Use "dmesg | grep tty" at the Linux prompt to check active terminal devices; on the Olimex board the terminal device is "/dev/ttyAM0". The engineer must use the termios(3) library to performs serial I/O control. Check the man pages to familiarize with termios(3). Once ready, set the I/O controls to 115200 Baud 8N1; the only connection options that the Reach touch screen board communicates with. If need be, one may experiment with termios(3) by creating a test program which reads a string from the serial port on the laptop and sends the data to

the Olimex board (and subsequently prints the data back on the laptops open session of minicom). However, remember to set the Baud rate on minicom to the Baud rate in the test program.

After properly configuring termios(3) on the client, create functionality that interrupts the client's normal operation using a signal handler to SIGIO that instantiates any time an input arrives via the serial connection to the touch screen. The signal handler should switch global flags for pause/play, volume, and echo filter on/off. Add at least one of these functions to the client. Pause/play is easiest because one just needs to add an "if statement" which simply skips past sending and receiving data from the local socket when the user sets the flag to pause. Like a traffic jam, when the server buffers fill up in TCP/IP, the sockets will block all traffic on the server side and wait until the client decides to start receiving data again before the server continues forward with any instructions; this process essentially pauses the entire system while the client waits for the user to push play. Next, connect the laptop to the Reach touch screen system via serial (this will not need a Null modem connector).The touch screen interface is created using a combination of bitmap and macro files loaded onto the flash memory and interpreted by the microprocessor for establishing functionality of the bitmap images with respect to user touch. Refer to the **Hardware Implementation** section for a more detailed explanation. After properly programming the touch screen, connect the laptop back to the Olimex board and run ./server on the laptop and ./client on the Olimex board. Promptly switch the serial connection between the Olimex board and laptop to the Olimex board and touch screen by unplugging the cable from the laptop and removing the Null Modem connector. Test the functionality by pressing the programmed pause/play button.

Finally, add functions for the volume control and echo filter. Similar to pause/play, these functions should be switched via global flags within the client that change value during a SIGIO. In order to change the volume the client program simply needs to multiply each incoming PCM value by the percent of total volume. For example, if the max volume is 5 and the current level is 3, multiply the PCM value by 3/5 to modify the amplitude. For the echo filter, create a ring buffer which stores current PCM values with some amplitude reduction. Make the buffer up to two to three times the sampling rate in size. After the buffer fills, starting from the beginning of the ring buffer the value stored should add to the output in a FIFO fashion. The new additive value then is again reduced in amplitude and takes the place of the value that was stored earlier. *Figure 3* below illustrates the echo filter. Block "z^{-R}" constitutes a delay formed by the ring buffer where "R" is the size of the buffer. The PCM data adds at "sigma" and "a" is the attenuation. Again, pick a buffer size equivalent to that of two to three times the sampling rate for nice long audible echoes. Making the buffer size (delay) too short will cause it to sound odd or possibly in audible. Pick an arbitrary value for "a" that attenuates the PCM value by about three quarters in value. Making it too large can

cause the output to clip forming unwanted distortion; too low will make the echo inaudible. Experimentation and patience is the key here. See **Appendix E** for final code results.

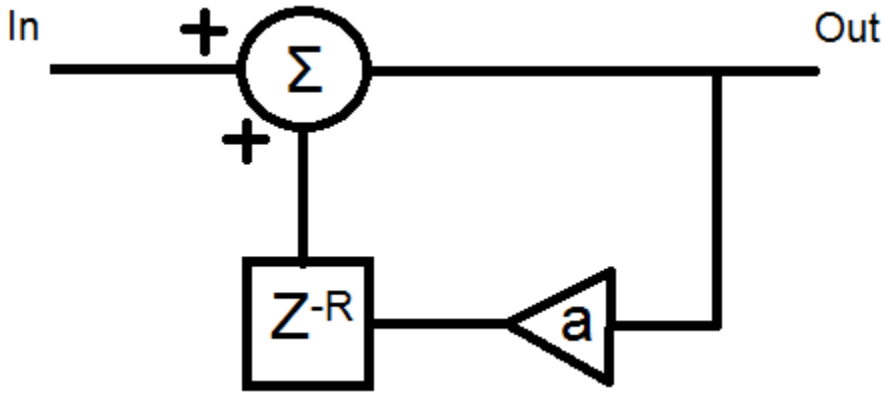


Figure 3: *Block diagram of a digital echo filter. Delays are created via a buffer of size "R" and "a" is a number between zero and one to attenuate the signal before stored in the buffer. The additive feedback process forms the echo like effect.*

After completing this portion of the code follow the procedure illustrated earlier to run the server and client and switch the serial connection from the laptop to the touch screen making sure to remove the Null Modem connector. Test the different functions by pushing virtualized buttons on the touch screen. If everything works the project is complete! The final control flow should function like *Figure 1* or *Figure 4* based on program names illustrates the software control below.

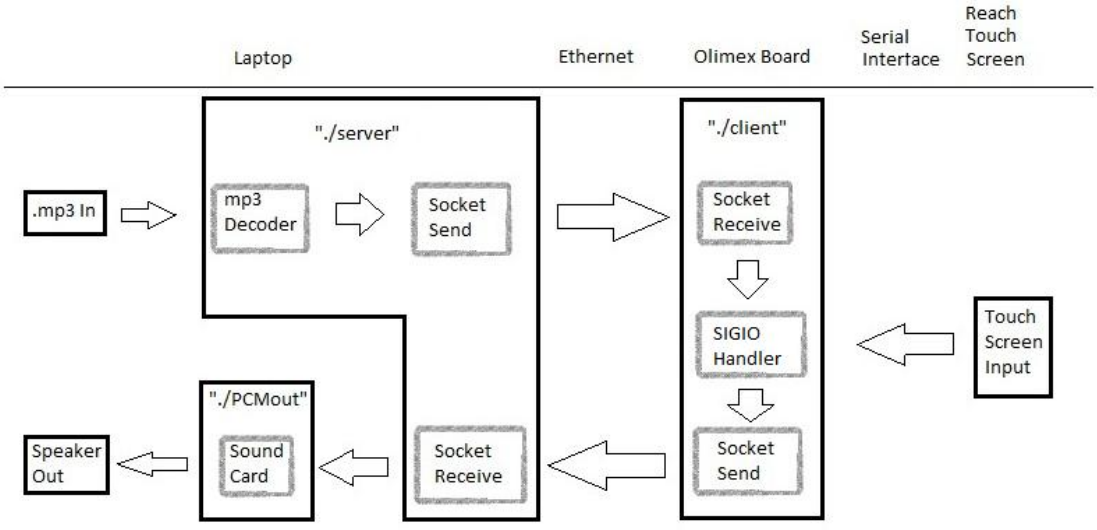


Figure 4: *Final software control flow diagram in terms of the programs that run and their functionality.*

5. Hardware Implementation

Olimex Development Board

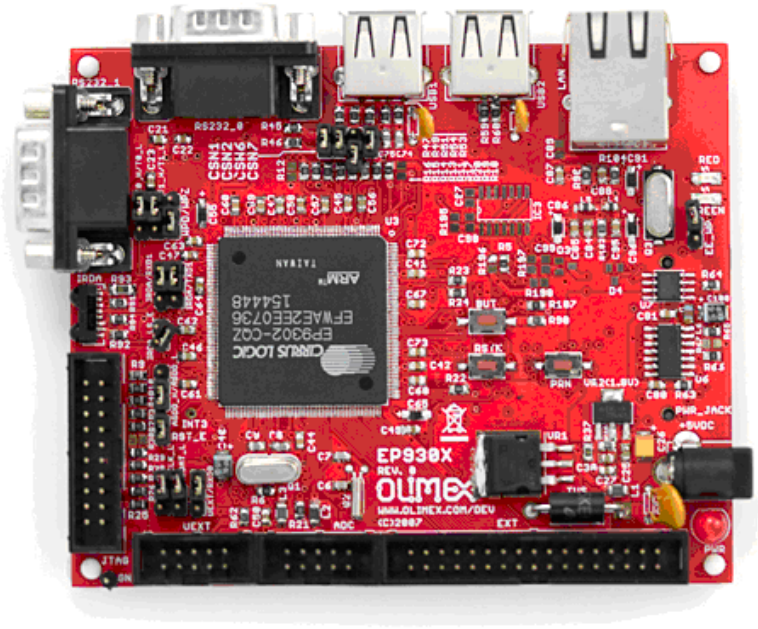


Figure 5: *Olimex Development Board*

The central hub of this project is based on the Olimex development board shown in Figure 1. It acts as the TCP/IP client and receives PCM data from the laptop server. To accomplish this we open up a session of RealTerm, a terminal emulator, on a PC connected to the board via an RS-232 cable set at 57.6 Kbaud with 8 data bits and 1 stop bit. A Linux shell prompt allows us to communicate with the device.

Establishing the Client on the Olimex board

In order to establish communication with the server, open up a session of RealTerm and apply wget(1) http://<the server's IP address>/<name of file>. This automatically downloads the designated file to the Olimex board. In this example we download the client program "./arm-client." The chmod 777 command provides the user with the permissions for reading, writing, and executing the arm-client file. Run the program by typing "./arm-client" <server's IP> <local tty serial port>. After completion, the Olimex board successfully sets up as a client to the laptop server. Figure 3 illustrates the process.

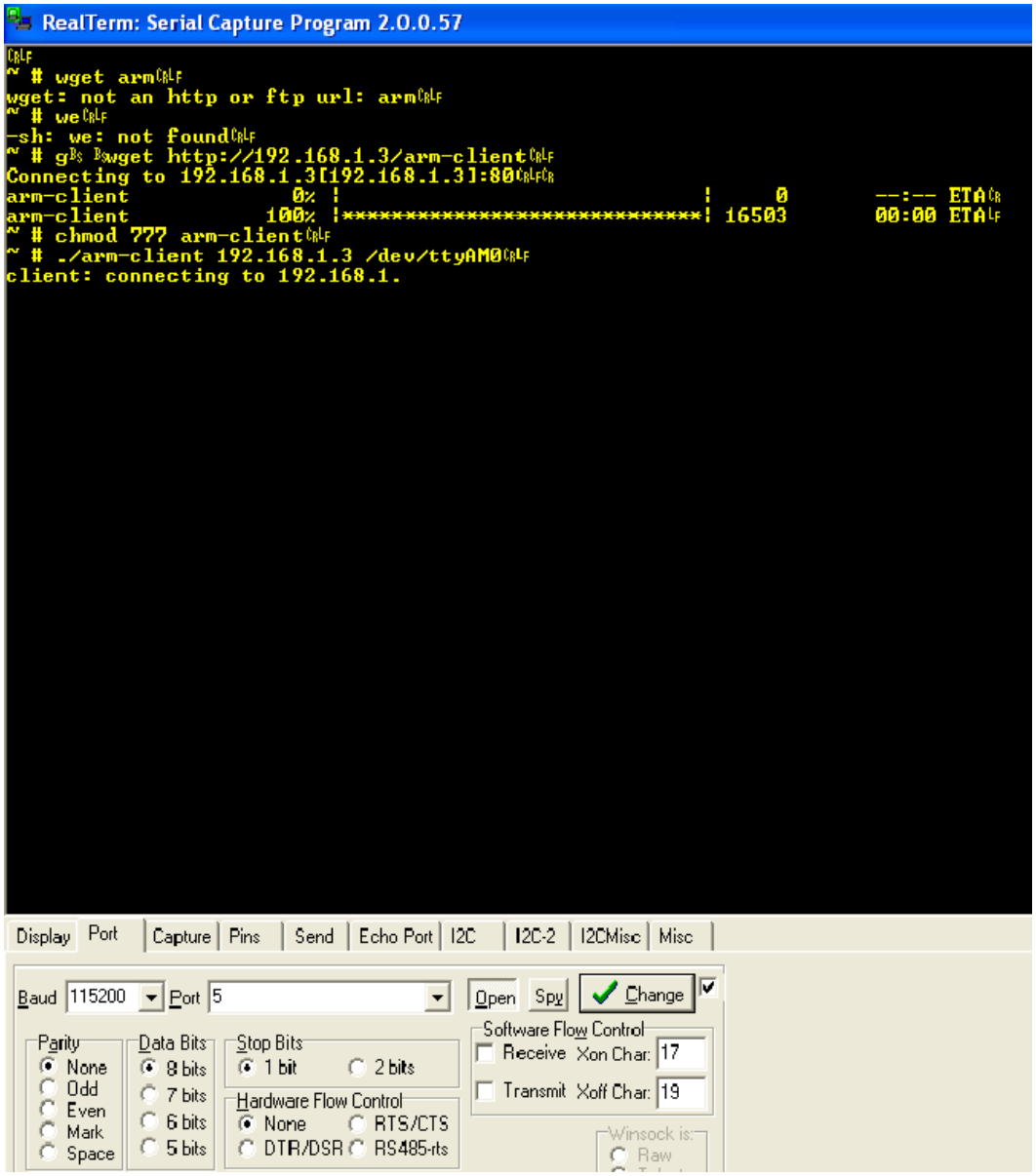


Figure 6: *RealTerm Session. Sets up communications with the Olimex board using an RS-232 serial connection operating at 57.6kbaud, 8 data bits, 1 stop bit. Commands inputted via a linux shell establish the client server relationship. Note: 115.2kbaud listed in the image due to a change in baud rate in an earlier session.*

The Embedded LCD Touch Screen

The touch screen used in this project is an SLCD43 TFT LCD Touch Screen developed by Reach Technologies. This embedded system has two RS-232 ports used for serial data transmission to the host microcontroller and a 32 bit CISC microprocessor with 4mb of flash memory. The bitmap images are loaded onto the flash memory for use as buttons/sliders etc.

The following figure is an RS-232 connection diagram with corresponding pins attributed to serial line communication between the embedded touch screen and the host device (in this case a PC)

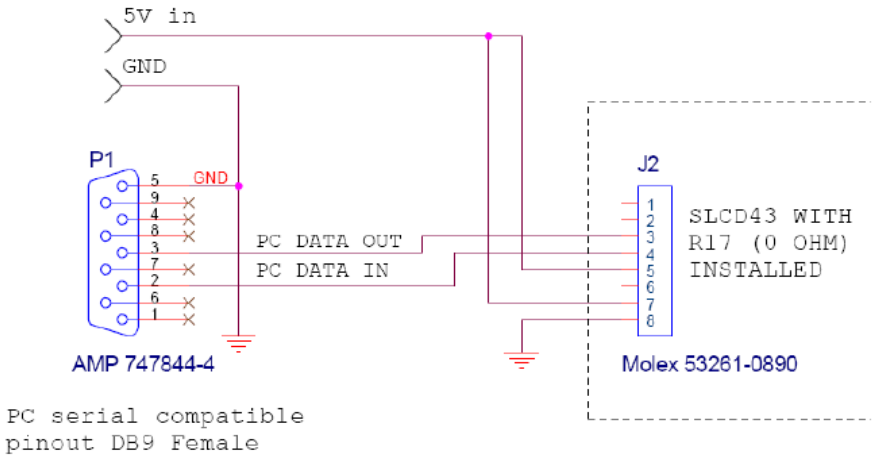


Figure 7: *Serial Data from RS-232 to J2 (taken from SLCD43 datasheet)*

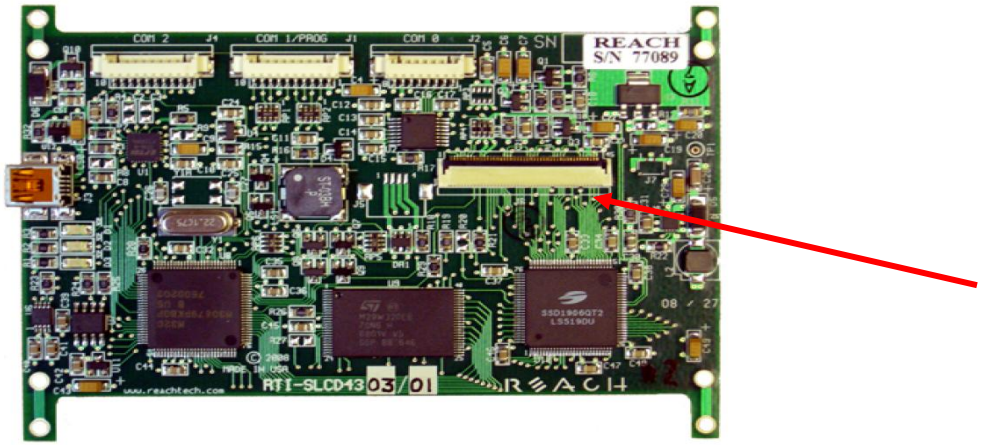
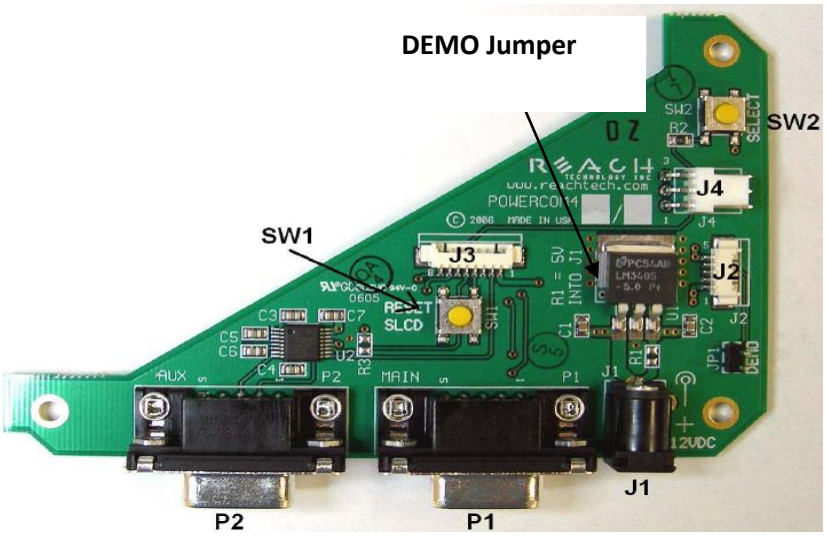


Figure 8: *SLCD43 Board Image (taken from SLCD43 datasheet)*



RS-232

Figure 9: *POWERCOM4 Board (taken from SLCD43 datasheet)*

SLCD43 Features

- 8 bits are allocated for RGB color.
- The screen is a 4.3'' active matrix LCD.
- WQVGA (480X272 pixels) resolution
- 4 wire resistive touch screen
- 32 bit CISC processor with 4mb of flash memory
- USB / RS-232 (3.3 V CMOS) serial interfaces

Programming the SLCD43

The BMPLoad program shown in the figure below is provided by Reach Technologies to program their embedded touch screen systems. Bitmaps to be loaded onto the flash are shown in the large window on the left part of the screen, whereas the macro files which define the functionality of those bitmaps (i.e. buttons, sliders, menu macro, etc.) is loaded using the Add Macro File button on the screen. The macro file is simply a text file with ASCII code that the SLCD43 interprets as commands for defining the bitmap functionality. The macro code is included in the

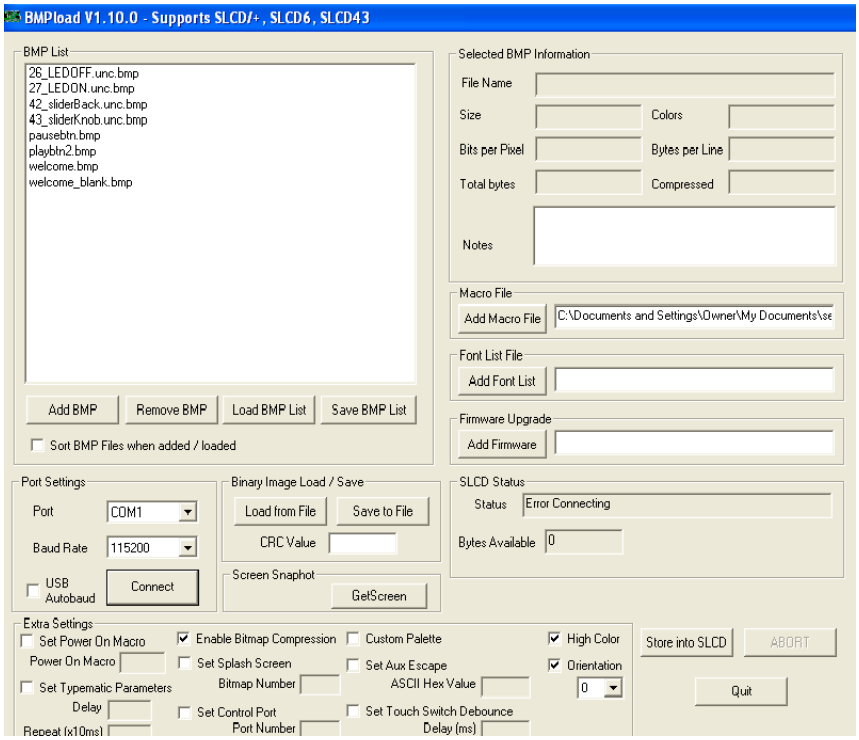


Figure 10: *Session of BMPload communicates with embedded touchscreen controller using a 115.2 kbaud, 8 data bit, 1 stop bit, no parity RS-232 serial connection.*

6. Software Implementation

Audio data is represented as 16-bit signed pulse-code modulated chunks. Music is sampled at 44.1 kHz and produces interlaced stereo output, meaning every other sample routes to one specific speaker output.

Pulse Code Modulation: Volume Control Example

The following source code shows how a 16-bit signed PCM block of sampled music data is manipulated to change the overall gain or volume of the sample. Control data comes from the LCD's volume slider.

```
void VolumeCntrl(void)
{
unsigned int value;             // Host Notification  l<128> : <value>
int size;
int samples = size/2
int16_t pcm[samples]; // read pcm data
int32_t pcmval;
uint8_t level = value
int value;
int multiplier = tan(value/5);

char inBuf[10];

//check for value.
if(EOF != gets(inBuf))
{
        value = inBuf[5] - '0';
        for (int x = 0; x < samples; x++)
        {
                pcmval = pcm[x] * multiplier;
                if (pcmval < 32767 && pcmval > -3278) {
                        pcm[x] = pcmval   //normal bounds
                } else if (pcmval > 32767) {
                        pcm[x] = 32767;  //max bound
                } else if (pcmval < -32768) {
                        pcm[x] = -32768; //min bound
                }
        }
```

}}

Terminal I/O in UNIX

Terminal I/O within the Unix operating system centers around the termios API. The interface is set up using the struct termios data structure defined below.

```
struct termios {
        tcflag_t   c_iflag;    // input flags
        tcflag_t   c_oflag;  // output flags
        tcflag_t   c_cflag;  // control flags
        tcflag_t   c_lflag;  // local flags
        cc_t         c_cc[NCCS]; // control characters
```

Taken from "Advanced Programming in the Unix Environment"

The input flags control the input of characters by the terminal device driver and the output flags of course control the driver output. The control flags affect the RS-232 serial lines and the local flags affect the interface between the user and the device driver.

In order to read and set parameters, following functions are used

- Tcgetattr: fetch attributes (termios structure)
- Tcsetattr: set attributes (termios structure)
- Cfgetispeed: get input speed
- Cfgetospeed: get output speed
- Cfsetispeed: set input speed
- Cfsetospeed: set output speed

Taken from "Advanced Programming in the Unix Environment"

Basic Serial Operations in Linux

To instantiate a serial connection in the linux environment, the following coding procedures are used for data read/write verification.

1. *Opening a serial port in Linux*

The following code opens a serial port in Linux.

In Linux, a serial port is simply a file which is referred to by a file descriptor labeled fd.

```
int fd = open("/dev/ttyS0", O_RDWR | O_NOCTTY | O_NDELAY);

        //  0_RDWR = enable read/write

        //  O_NOCTTY = Not controlling terminal of the port

        //  0_NDELAY = Does not care about state of DCD line
```

2. *Writing data to the port*

The function write() takes a file descriptor, data to write, and the number of bytes being sent as parameters.

```
n = write(fd, data, 4);

  if (n < 0)

  fputs("write() of 4 bytes failed!\n", stderr);

      // returns number of bytes sent or -1 if error
```

3. *To read data from the port…*

```
int iIn = read(fd, char* result, n_bytes);
```

We read in n_bytes from the file associated with file descriptor fd. In this case the file refers to a terminal device such as tty0. The data is then loaded into the character buffer result.

7. Dictionary of Terms

ADC: Analog to Digital Converter.

ALSA: A component in the Linux Kernel which provides sound card device drivers.

Apache2(8): Web server software for Unix machines.

Audio Controller: A device for controlling the audio signals in terms of their amplitude (volume) and frequency response.

Client: A software application that accesses a server from a network to obtain data. The client/server relationship in this project run on Unix domain sockets between the laptop and Olimex board.

Decoder: A device which reverses the encoding process so that the original data can be retrieved.

execlp(3): Function used to execute a file where the first argument is the path of the file to be executed and the following two are NULL pointer arguments.

LGPL: GNU Lesser General Public License is a software license used for software libraries.

Mp3: MPEG-1 Audio Layer 3 is an audio encoding format that uses a lossy form of data compression. The audio frequencies beyond the human auditory range are cut out to minimize the amount of data needed to represent the audio.

Mpg123: A Linux audio player software program used to play an mp3 file.

Microcontroller: An integrated circuit containing a small computer with programmable I/O, memory, and a core processor.

PCM: Pulse Code Modulation. An analog signal is sampled at a certain frequency and the signal's magnitude is represented in binary.

Pipe(2) : Creates a data pipe for communication in which the parameters are the file descriptors for the read and write end the pipe, respectively. Data entering the write end of the pipe is buffered by the Linux kernel until it is read from the read end of the pipe.

Server: A piece of hardware such as a computer which provides services to clients.

Sound Card: A PC card that provides audio I/O functionality on a computer.

TCP/IP: The internet protocol suite consisting of the Transmission Control Protocol / Internet Protocol. The TCP part controls data flow whereas the IP part is in charge of data communication across a network.

Touch Screen: An electronic display which reacts to user touch and interprets the coordinates of the touch as an input for triggering various operations.

Virtual Box: A software package which allows multiple guest operating systems to run on top of a host operating system. Each guest operating system runs in its own unique virtual environment.

Wav: Waveform Audio File Format. An IBM & Microsoft standard for storing an audio bitstream on a PC, which is usually encoded in pulse-code modulated segments.

Wget(1) : Retrieves files from a web address via HTTP, HTTPS, and FTP protocols.

8. Bibliography

Davie, Bruce S. & Peterson, Larry L. (2007). *Computer Networks: A Systems Approach.* San Francisco, CA: Morgan Kaufman.

Hall, Brian "Beej Jorgensen." (March 23, 2009). *Beej's Guide to Network Programming: Using Internet Sockets*. Retrieved January 25th, 2010, from < http://beej.us/guide/bgnet/output/html/multipage/index.html>.

Hankins, Greg & Lawyer, David. (December 2008). *Serial HOWTO*. Retrieved May 3rd, 2010, from <http://tldp.org/HOWTO/Serial-HOWTO.html>.

Hass, Prof. Jeffrey. (2005). *Introduction to Computer Music: Volume One*. "Chapter Three: MIDI." Bloomington, IN: Indiana University. Retrieved February 3rd, 2010, from <http://www.indiana.edu/~emusic/etext/MIDI/chapter3_MIDI.shtml>.

Hipp, Michael & Orgis, Thomas. (2010). *libmpg123 Documentation*. Retrieved April 15th, 2010, from <http://mpg123.org/api/>.

Nagorni, Matthias. (February 24, 2010). *ALSA Programming HOWTO*. Retrieved May 15th, 2010 from <http://www.suse.de/~mana/alsa090_howto.html>.

"PCM Audio | Part 3: Basic Audio Effects – Volume Control | YPass.net Blog." Main Page | Ypass.net 2010. Web. 27 July 2010. <http://www.ypass.net/blog/2010/01/pcm-audio-part-3-basic-audio-effects-volume-control/>.

Rago, Stephen A. & Stevens, Richard W. (2005). *Advanced Programming in the UNIX Environment*. Upper Saddle River, NJ: Addison-Wesley.

9. Conclusions & Future Modifications

The requirements of this project provide touch screen sound controls and Ethernet network capabilities to the disk jockey. The successful completion of these requirements opens the door for further modifications and additional features to make a more attractive and practical product.

This project proved that all necessary communications between the laptop server, Olimex client, and touch screen control communicate effectively. If we modify this product, we will begin by first managing all the power supplies from a voltage regulator switch-box to avoid using separate power cables for each component. Additionally, we would re-route the audio to a DAC contained within the sound controller. This would simplify outputting audio by providing a single solution which combines both our controls and audio output. With our current implementation, audio must be routed back to the laptop which consumes unnecessary Ethernet bandwidth.

From a software point of view, there are a limitless amount of extra features that can be included to give the sound controller more character and functionality. One idea is to add a dynamic digital filter feature for the user. Currently there the product has one filter that the user can access, but with a filter creator the user can draw out frequency and amplitude envelopes as well as selectable timing delays to create a unique audio effect on the fly. Also, the touch screen menu needs additional macros for handling multiple interfaces for added functionality.

This project signifies a first and important step towards developing a marketable product. The Ethernet functionality itself makes this particular sound controller unique, since most other products still use the MIDI protocol and do not process raw PCM data.

Appendices

Appendix A: Parts List & Prices

Part	Price (in USD)
Reach SLCD43 Embedded Touch Screen	0
Olimex Development/Prototyping Board	190
Gateway Laptop	1200
Netgear Router	20
2 USB to RS-232 Serial Converters	10 x 2 = 20
2 RS-232 Serial Cables	5 x 2 = 10
1 Ethernet Cable	5
5 VDC Supply	0
Null Modem Connector	11
Linux OS	0
RealTerm Terminal Emulator	0
BMPLoad	0
Total	**1456**

Appendix B: Code for decoding .mp3 to .wav file

```
/**
 * @author James Fenley
 * @author Jon Law
 *
 * mp3decode.c
 * Decodes an mp3 file and and writes the information to
a wave file.
 *
 * Code is an edited version of mpg123_to_wav.c provided
under the LGPL
 * by Nicholas Humfrey at http://mpg123.org. LGPL license
may be found at
 * http://www.gnu.org/licenses/lgpl-2.1.html
 */

#include <stdio.h>
#include <strings.h>
#include <mpg123.h>
#include <sndfile.h>

void cleanup(mpg123_handle *mh)
{
    mpg123_close(mh);
    mpg123_delete(mh);
    mpg123_exit();
}

int mp3send(mpg123_handle *mh, unsigned char *buffer,
size_t buffer_size,
            off_t *samples, SNDFILE *sndfile)
{
    int error = MPG123_OK;
    size_t sofar;

    error = mpg123_read(mh, buffer, buffer_size, &sofar);
    sf_write_short(sndfile, (short *)buffer,
sofar/sizeof(short));
```

```
    *samples += sofar/sizeof(short);

   return error;
}

int main(int argc, char *argv[])
{
    SNDFILE *sndfile = NULL;
    SF_INFO sfinfo;
    mpg123_handle *mh = NULL;
    unsigned char *buffer = NULL;
    size_t buffer_size = 0, done = 0;
    int channels = 0, encoding = 0, error = MPG123_OK;
    long rate = 0;
    off_t samples = 0;

    if(argc != 3)
    {
        fprintf(stderr, "usage: mp3decode <input>
<output>\n");
        exit(-1);
    }
    printf("Input file: %s\n", argv[1]);
    printf("Output file: %s\n", argv[2]);

    error = mpg123_init();

    /*Creates a handler, opens the file to the handler,
and get the sound
     *files format*/
    if(error != MPG123_OK || (mh = mpg123_new(NULL,
&error)) == NULL
        || mpg123_open(mh, argv[1]) != MPG123_OK
        || mpg123_getformat(mh, &rate, &channels,
&encoding) != MPG123_OK)
    {
        fprintf(stderr, "Error while creating, opening,
and formatting ");
        fprintf(stderr, "a handler: %s\n",
```

```
            mh == NULL ? mpg123_plain_strerror(error) :
mpg123_strerror(mh));
        cleanup(mh);
        exit(-1);
    }

    if(encoding != MPG123_ENC_SIGNED_16)
    {
        /*Confirms the output format at signed shorts;
this is the default*/
        cleanup(mh);
        fprintf(stderr, "Bad encoding: 0x%x!\n",
encoding);
        exit(-2);
    }

    /*Ensure that this output format will not change
     *(it could, when we allow it).*/
    mpg123_format_none(mh);
    mpg123_format(mh, rate, channels, encoding);
    /*Creates a buffer of the recommended output buffer
size.*/
    buffer_size = 1000;
    buffer = malloc(buffer_size);

    /*Sets up the information libsndfile will use to
write the decoded
     *data*/
    bzero(&sfinfo, sizeof(sfinfo));
    sfinfo.samplerate = rate;
    sfinfo.channels = channels;
    sfinfo.format = SF_FORMAT_PCM_16;
    printf("Creating 16bit PCM with %i channels
and %liHz.\n", channels, rate);

    if(!(sndfile = sf_open(argv[2], SFM_WRITE, &sfinfo))
    {
        fprintf(stderr, "Cannot open output file!\n");
        cleanup(mh);
```

```
        exit(-2);
    }

    error = MPG123_OK;
    while(error == MPG123_OK)
    {
        error = mp3send(mh, buffer, buffer_size, samples,
sndfile);
    }
    if(error != MPG123_DONE)
        fprintf(stderr, "Warning: Decoding ended
prematurely because: %s\n",
            error == MPG123_ERR ? mpg123_strerror(mh) :
mpg123_plain_strerror(error));

    sf_close(sndfile);

    samples /= channels;
    printf("%li samples written.\n", (long)samples);

    cleanup(mh);
    return 0;
}
```

Appendix C: Code for writing a .wav file via server/client connection

./server header file

```
/**
 * @author James Fenley
 * @author Jon Law
 *
 * server.h
 *
 * Defines macros, functions, and/or structures for
server.c
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>
#include <mpg123.h>
#include <sndfile.h>

#define PORT "3490" /*The port the users will connect
to*/
#define BACKLOG 2   /*How many pending connections the
queue will hold*/

/*A signal hander for reaping zombies*/
void sigchld_handler(int s);
/*Gets the IP address of appropriate form (IPv4/v6)*/
void *get_in_addr(struct sockaddr *sa);
/*A loop for generating sockets*/
```

```
void acceptLoop(int sd, struct sockaddr_storage
*their_addr);
/*Decodes mp3 data to PCM and sends it over the network*/
int mp3send(mpg123_handle *mh, off_t *samples, int sd);
/*Sets up mpeg handlers and file writer handlers
specifically for
 *witing signed 16bit PCM data*/
void mp3Decode(int sd);
/*Cleans all mp3 decoding stuff and exits the mpg123
library*/
void mpgClean(mpg123_handle *mh);
```

./server program:

```
/**
 * @author James Fenley
 * @author Jon Law
 *
 * server.c
 *
 * A stream socket server. Waits for a client to connect.
 * Decodes mp3 data via mpg123 library to PCM data
 * and sends this data out to the connected client.
 */

#include "server.h"

int main(void)
{
    /*Vars for network server structure*/
   int sd;                  //socket descriptor
   struct addrinfo hints, *servinfo, *p;
   struct sockaddr_storage their_addr; // connector's
address information
   struct sigaction sa;
   int yes = 1, ecode;

   memset(&hints, 0, sizeof(hints));
   hints.ai_family = AF_UNSPEC;
```

```
   hints.ai_socktype = SOCK_STREAM;
   hints.ai_flags = AI_PASSIVE; // use my IP

   if((ecode = getaddrinfo(NULL, PORT, &hints,
&servinfo)) != 0)
    {
      fprintf(stderr, "getaddrinfo: %s\n",
gai_strerror(ecode));
      return 1;
   }

   // loop through all the results and bind to the first
we can
   for(p = servinfo; p != NULL; p = p->ai_next)
    {
      if((sd = socket(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1)
        {
         perror("server: socket");
         continue;
      }

      if(setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, &yes,
            sizeof(int)) == -1)
        {
         perror("setsockopt");
         exit(1);
      }

      if(bind(sd, p->ai_addr, p->ai_addrlen) == -1)
        {
         close(sd);
         perror("server: bind");
         continue;
      }

      break;
   }
```

```
    if(p == NULL)
     {
       fprintf(stderr, "server: failed to bind\n");
       return 2;
    }

    freeaddrinfo(servinfo); // all done with this
structure

    if(listen(sd, BACKLOG) == -1)
     {
       perror("listen");
       exit(1);
    }

    sa.sa_handler = sigchld_handler; // reap all dead
processes
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    if(sigaction(SIGCHLD, &sa, NULL) == -1)
     {
       perror("sigaction");
       exit(1);
    }

    printf("server: waiting for connections...\n");

     acceptLoop(sd, &their_addr);

    return 0;
}

void acceptLoop(int sd, struct sockaddr_storage
*their_addr)
{
     int sdnew;
    char theirIP[INET6_ADDRSTRLEN];
    socklen_t sin_size;
```

```
    while(1)
    {
        sin_size = sizeof(*their_addr);
        sdnew = accept(sd, (struct sockaddr *)their_addr,
&sin_size);
        if(sdnew == -1)
        {
            perror("acceptLoop");
            continue;
        }

        /*Converts client's IP from integer form to a
string*/
        inet_ntop((*their_addr).ss_family,
            get_in_addr((struct sockaddr *)their_addr),
theirIP,
                        sizeof(theirIP));
        printf("server: got connection from %s\n",
theirIP);

        if(!fork())     //child process
        {
            mp3Decode(sdnew);
            shutdown(sdnew, SHUT_RDWR);
            close(sdnew);
            exit(0);
        }
        //parent
        close(sdnew);
    }
}

void mpgClean(mpg123_handle *mh)
{
    mpg123_close(mh);
    mpg123_delete(mh);
    mpg123_exit();
}
```

```
/*Main function for reading from the mp3 file and sending
 *the decoded data over a socket*/
void mp3Decode(int sd)
{
    /*Vars for mp3 decoding*/
    mpg123_handle *mh = NULL;
    int channels = 0, encoding = 0, error = MPG123_OK;
    long rate = 0;
    off_t samples = 0;
    char *mp3File = "./alt_a.mp3";

    error = mpg123_init();

    /*Creates a handler, opens the file to the handler,
and get the sound
     *files format*/
    if(error != MPG123_OK || (mh = mpg123_new(NULL,
&error)) == NULL
        || mpg123_open(mh, mp3File) != MPG123_OK
        || mpg123_getformat(mh, &rate, &channels,
&encoding) != MPG123_OK)
   {
        fprintf(stderr, "Error while creating, opening,
and formatting ");
        fprintf(stderr, "a handler: %s\n",
            mh == NULL ? mpg123_plain_strerror(error) :
mpg123_strerror(mh));
        mpgClean(mh);
        exit(-1);
    }

    /*Confirms the output format at signed shorts; this
is the default*/
    if(encoding != MPG123_ENC_SIGNED_16)
    {
        mpgClean(mh);
        fprintf(stderr, "Bad encoding: 0x%x!\n",
encoding);
        exit(-2);
```

```
    }

    /*Ensure that this output format will not change
     *(it could, when we allow it).*/
    mpg123_format_none(mh);
    mpg123_format(mh, rate, channels, encoding);

    printf("Creating 16bit PCM with %i channels
and %liHz.\n", channels, rate);

    /*As long as things are decoding properly continue to
send audio
     *down the stream*/
    error = MPG123_OK;
    fprintf(stderr, "here\n");
    while(error == MPG123_OK)
    {
        error = mp3send(mh, &samples, sd);
    }
    if(error != MPG123_DONE)
        fprintf(stderr, "Warning: Decoding ended
prematurely because: %s\n",
            error == MPG123_ERR ? mpg123_strerror(mh) :
mpg123_plain_strerror(error));

    samples /= channels;
    printf("%li samples written.\n", (long)samples);

    mpgClean(mh);
}

/*Decodes mp3 data to PCM and sends it over the network*/
int mp3send(mpg123_handle *mh, off_t *samples, int sd)
{
    int error = MPG123_OK;
    size_t sofar, buffer_size = 1000;
    unsigned char *buffer;

    /*Creates a buffer of the recommended output buffer
```

```
size.*/
    buffer = malloc(buffer_size);

    error = mpg123_read(mh, buffer, buffer_size, &sofar);
    send(sd, buffer, sofar, 0);
    *samples += sofar/sizeof(short);

    return error;
}

void sigchld_handler(int s)
{
  while(waitpid(-1, NULL, WNOHANG) > 0);
}

// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{
  if (sa->sa_family == AF_INET)
    {
     return &(((struct sockaddr_in*)sa)->sin_addr);
   }

   return &(((struct sockaddr_in6*)sa)->sin6_addr);
}
```

./client header:

```
/**
 * @author James Fenley
 * @author Jon Law
 *
 * client.h
 *
 * Defines macros, functions, and/or structures for
client.c
 */

#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#define PORT "3490"

#define MAXDATASIZE 1000

/*Get sockaddr, IPv4 or IPv6*/
void *get_in_addr(struct sockaddr *sa);
void soundrecv(int sd);
```

./client program:

```
/**
 * @author James Fenley
 * @author Jon Law
 *
 * client.c
 *
 * Streams PCM data from server and writes the raw data
to a
 * .raw file.
 */

#include "client.h"

/*Get sockaddr, IPv4 or IPv6*/
void *get_in_addr(struct sockaddr *sa)
{
   if(sa->sa_family == AF_INET)
    {
      return &(((struct sockaddr_in*)sa)->sin_addr);
```

```
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(int argc, char *argv[])
{
     /*Vars for the client network structure*/
    int sd;
    struct addrinfo hints, *servinfo, *p;
    int encode;
    char s[INET6_ADDRSTRLEN];

    if(argc != 2)
     {
        fprintf(stderr,"usage: client hostname\n");
        exit(1);
    }

    memset(&hints, 0, sizeof hints);
   hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    if((encode = getaddrinfo(argv[1], PORT, &hints,
&servinfo)) != 0)
     {
       fprintf(stderr, "getaddrinfo: %s\n",
gai_strerror(encode));
       return 1;
    }

    // loop through all the results and connect to the
first we can
    for(p = servinfo; p != NULL; p = p->ai_next)
     {
       if((sd = socket(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1)
         {
          perror("client: socket");
```

```
            continue;
        }

        if(connect(sd, p->ai_addr, p->ai_addrlen) == -1)
          {
            close(sd);
            perror("client: connect");
            continue;
        }

      break;
    }

    if (p == NULL)
     {
        fprintf(stderr, "client: failed to connect\n");
        return 2;
    }

    inet_ntop(p->ai_family, get_in_addr((struct sockaddr
*)p->ai_addr),
            s, sizeof s);
    printf("client: connecting to %s\n", s);

    freeaddrinfo(servinfo); // all done with this
structure

    soundrecv(sd);

    close(sd);

    return 0;
}

void soundrecv(int sd)
{
     int sofar;
    unsigned char buf[MAXDATASIZE+1];
     FILE *fp = fopen("alt_a.raw", "w");
```

```
    while(1)
    {
        if((sofar = recv(sd, buf, MAXDATASIZE, 0)) == -1)
        {
            perror("recv");
            exit(-2);
        }
        if(sofar == 0)
            break;
        fwrite(buf, sizeof(char), sofar, fp);
        fprintf(stderr, "%c", buf[0]);
    }
    fclose(fp);
}
```

Appendix D: PCMout Program for outputting PCM data to a speaker

./PCMout header file:

```
/**
 * @author James Fenley
 * @author Jon Law
 *
 * PCMout.h
 *
 * Type defs, structures, and functions for PCMout.c
 */

#include <alsa/asoundlib.h>
#include <string.h>

#define SRATE 44100
#define FRAME_SIZE 1000

void initializePCM(snd_pcm_t *pcmHandle,
snd_pcm_hw_params_t *hwparams,
                   snd_pcm_uframes_t *frames);
void audioOut(snd_pcm_t *pcmHandle, snd_pcm_hw_params_t
*hwparams,
                snd_pcm_uframes_t *frames);
```

./PCMout program:

```
/**
 * @author James Fenley
 * @author Jon Law
 *
 * PCMout.c
 *
 * The following code outputs PCM data to a speaker via
ALSA.
 */

#include "PCMout.h"
```

```
int main(int argc, char *argv[])
{
    /*Vars for the PCM audio output*/
    snd_pcm_t *pcmHandle;      /*Create a ALSA pcm
handler*/
    snd_pcm_stream_t stream = SND_PCM_STREAM_PLAYBACK;
/*Set the stream to playback*/
    /*Struct will contain information about the audio
hardware device
     *thus giving the program a range of specifications
it may use
     *to output sound (such as allowed output
frequencies)*/
    snd_pcm_hw_params_t *hwparams;
    snd_pcm_uframes_t frames = FRAME_SIZE;
    char *pcmName = "plughw:0,0";

    snd_pcm_hw_params_alloca(&hwparams);
    if(snd_pcm_open(&pcmHandle, pcmName, stream, 0) < 0)
    {
        fprintf(stderr, "Error opening PCM device %s\n",
pcmName);
       exit(-1);
    }

    /*Retreives the allowable hardware configurations
from the device
     *and puts the info in hwparams*/
    if(snd_pcm_hw_params_any(pcmHandle, hwparams))
    {
        fprintf(stderr, "Can not configure this PCM
device\n");
        exit(-1);
    }

    initializePCM(pcmHandle, hwparams, &frames);

    audioOut(pcmHandle, hwparams, &frames);
```

```
    snd_pcm_close(pcmHandle);

    return 0;
}

void audioOut(snd_pcm_t *pcmHandle, snd_pcm_hw_params_t
*hwparams,
                snd_pcm_uframes_t *frames)
{
    int c, pcmCheck;
    /*Set buffer size: 2 bytes/sample * 2 channels*/
    int size = (*frames) * 4;
    char *buffer = (char *)malloc(size);
    fprintf(stderr, "here\n");

    int fd = open("./alt_a_raw.wav", O_RDONLY, 0);
    if(fd == -1)
    {
        perror("Error opening file: ");
        exit(-1);
    }

    while(1)
    {
        c = read(0, buffer, size);
        if(c == 0)
        {
            fprintf(stderr, "end of file\n");
            break;
        }
        else if(c != size)
        {
            fprintf(stderr, "short read\n");
        }
        c = snd_pcm_writei(pcmHandle, buffer, *frames);
        if(c == -EPIPE)
            snd_pcm_prepare(pcmHandle);
        else if(c < 0)
```

```
        {
            fprintf(stderr, "error writei: %s\n",
snd_strerror(c));
        }
        else if(c != (int)(*frames))
        {
            fprintf(stderr, "short write, write %d
frames\n", c);
        }
    }
    close(fd);

    snd_pcm_drain(pcmHandle);
    free(buffer);
}

void initializePCM(snd_pcm_t *pcmHandle,
snd_pcm_hw_params_t *hwparams,
                   snd_pcm_uframes_t *frames)
{
    int rate = SRATE, dir = 0;
    unsigned int exactRate = SRATE; /*HW may not support
44.1K*/
    /*Set access type. Can be either:
     *SND_PCM_ACCESS_RW_INTERLEAVED or
SND_PCM_ACCESS_RW_NONINTERLEAVED.*/
    if(snd_pcm_hw_params_set_access(pcmHandle, hwparams,
        SND_PCM_ACCESS_RW_INTERLEAVED) < 0)
    {
        fprintf(stderr, "Error setting access.\n");
        exit(-1);
    }
    /*Set sample format: signed 16 bit little endian*/
    if(snd_pcm_hw_params_set_format(pcmHandle, hwparams,
        SND_PCM_FORMAT_S16_LE) < 0)
    {
        fprintf(stderr, "Error setting rate.\n");
        exit(-1);
    }
```

```
    /*Set sample rate, uses the nearest possible rate
supported
     *by hardware*/
    if(snd_pcm_hw_params_set_rate_near(pcmHandle,
hwparams, &exactRate,
                                       &dir) < 0)
    {
        fprintf(stderr, "Error setting rate.\n");
        exit(-1);
    }
    if(rate != exactRate)
    {
        fprintf(stderr, "The rate %d Hz not supported by
hardware.\n", rate);
        fprintf(stderr, "Using %d Hz instead.\n",
exactRate);
    }
    /*Set number of channels*/
    if(snd_pcm_hw_params_set_channels(pcmHandle,  hwparams,
2) < 0)
    {
        fprintf(stderr, "Error setting channels.\n");
        exit(-1);
    }
    /*Set period size: 1000 frames*/
    if(snd_pcm_hw_params_set_period_size_near(pcmHandle,
        hwparams, frames, &dir) < 0)
    {
        fprintf(stderr, "Error setting period size.\n");
        exit(-1);
    }
    /*Apply hw parameter settings to PCM device*/
    if(snd_pcm_hw_params(pcmHandle, hwparams) < 0)
    {
        fprintf(stderr, "Error setting HW params.\n");
        exit(-1);
    }
}
```

Appendix E: Final Code for Server/Client Programs and Touch Screen

Program for touch screen:

```
/*
  The following commands are written in
  ASCII text to be interpreted by the SLCD43
  after loading the macro files to the flash.

  The startup macro loads the menu macro
  when button 1 is pressed.

  The menu macro consists of a latching state
  button for play pause, a slider background
  and knob for volume control, and an LED off
  and on latching button for the echo filter.

  Host Notifications:

        button definitions: s <n><s><return>
                <n> = button number
                <s> = state (0 or 1)

         slider definition: l<idx>:<value>
                <inx> = index of slider
                <value> = inputted value on slider.
*/

#define startup

  xm 1 menu

  bd 1 100 100  2 "" "" 0 0 0 0 1 2


#end

#define menu

  bd 2 50 50 2 "" "" 0 0 0 0 3 4

  sl 1 5 80 50 6 0 0 0 0 5 0

  bd 3 100 50 2 "echo" "echo" 15 40 15 40 7 8

#end
```

Final server header code:

```
/**
 * @author James Fenley
```

```
 * @author Jon Law
 *
 * server.h
 *
 * Defines macros, functions, and/or structures for
server.c
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>
#include <mpg123.h>
#include <sndfile.h>

#define PORT "3490" /*The port the users will connect
to*/
#define BACKLOG 2   /*How many pending connections the
queue will hold*/
#define DOWNSAMPLE 4 /*Divides the number of samples and
sampling rate
                      *by this value, necesssary to keep
the microcontroller
                      *from causing buffer underrun,
remember to adjust
                      *SRATE in PCMout.h as well*/
#define MAXDATASIZE 1000
#define PIPE_R 0
#define PIPE_W 1

/*A signal hander for reaping zombies*/
```

```
void sigchld_handler(int s);
/*Gets the IP address of appropriate form (IPv4/v6)*/
void *get_in_addr(struct sockaddr *sa);
/*A loop for generating sockets*/
void acceptLoop(int sd, struct sockaddr_storage
*their_addr);
/*Decodes mp3 data to PCM and sends it over the network*/
int mp3send(mpg123_handle *mh, off_t *samples, int sd);
/*Sets up mpeg handlers and file writer handlers
specifically for
 *witing signed 16bit PCM data*/
void mp3Decode(int sd);
/*Cleans all mp3 decoding stuff and exits the mpg123
library*/
void mpgClean(mpg123_handle *mh);
/*Writes PCM data received from the client to stdout*/
void PCMrecv(int sd);
```

Final code for server program:

```
/**
 * @author James Fenley
 * @author Jon Law
 *
 * server.c
 *
 * A server that streams mp3 files from stdin.
 * It dynamically decodes the mp3 file and sends PCM data
 * via a TCP socket. The mpg123 library is used for
decoding.
 */

#include "server.h"

char *mp3File;

/*nulls out a buffer*/
void nullLine(unsigned char *buf, int length)
{
```

```
    int i;
    for(i = 0; i < length; i++)
        buf[i] = 0;
}

int main(int argc, char *argv[])
{
    /*Confirms mp3 file entered into command prompt*/
    if(argc != 2)
    {
        fprintf(stdout, "usage: ./server <*.mp3>\n");
        exit(1);
    }
    mp3File = argv[1];

    /*Vars for network server structure*/
   int sd;                  //socket descriptor
   struct addrinfo hints, *servinfo, *p;
   struct sockaddr_storage their_addr; // connector's
address information
   struct sigaction sa;
   int yes = 1, ecode;

   memset(&hints, 0, sizeof(hints));
   hints.ai_family = AF_UNSPEC;
   hints.ai_socktype = SOCK_STREAM;
   hints.ai_flags = AI_PASSIVE; // use my IP

   if((ecode = getaddrinfo(NULL, PORT, &hints,
&servinfo)) != 0)
    {
      fprintf(stderr, "getaddrinfo: %s\n",
gai_strerror(ecode));
      return 1;
   }

   // loop through all the results and bind to the first
we can
   for(p = servinfo; p != NULL; p = p->ai_next)
```

```
    {
      if((sd = socket(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1)
        {
         perror("server: socket");
         continue;
      }

      if(setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, &yes,
            sizeof(int)) == -1)
        {
         perror("setsockopt");
         exit(1);
      }

      if(bind(sd, p->ai_addr, p->ai_addrlen) == -1)
        {
         close(sd);
         perror("server: bind");
         continue;
     }

      break;
   }

   if(p == NULL)
    {
      fprintf(stderr, "server: failed to bind\n");
      return 2;
   }

   freeaddrinfo(servinfo); // all done with this
structure

   if(listen(sd, BACKLOG) == -1)
    {
      perror("listen");
      exit(1);
   }
```

```
   sa.sa_handler = sigchld_handler; // reap all dead
processes
   sigemptyset(&sa.sa_mask);
   sa.sa_flags = SA_RESTART;
   if(sigaction(SIGCHLD, &sa, NULL) == -1)
    {
      perror("sigaction");
      exit(1);
   }

   printf("server: waiting for connections...\n");

    acceptLoop(sd, &their_addr);

   return 0;
}

void acceptLoop(int sd, struct sockaddr_storage
*their_addr)
{
    int sdnew, i;
   char theirIP[INET6_ADDRSTRLEN];
   socklen_t sin_size;

    while(1)
   {
        sin_size = sizeof(*their_addr);
        sdnew = accept(sd, (struct sockaddr *)their_addr,
&sin_size);
        if(sdnew == -1)
        {
            perror("acceptLoop");
            continue;
        }

        /*Converts client's IP from integer form to a
string*/
        inet_ntop((*their_addr).ss_family,
```

```
            get_in_addr((struct sockaddr *)their_addr),
theirIP,
                        sizeof(theirIP));
        printf("server: got connection from %s\n",
theirIP);
        /*child process for sending decoded mp3 data*/
        if(!fork())
        {
            i = 0;
            close(sd);
            while(1)
            {
                mp3Decode(sdnew);
            }
           shutdown(sdnew, SHUT_WR);
            close(sdnew);
            exit(0);
        }
        /*child process for receiving filtered PCM data*/
        if(!fork())
        {
            close(sd);
            PCMrecv(sdnew);
           shutdown(sdnew, SHUT_RD);
            close(sdnew);
            fprintf(stderr, "Done receiving data\n");
            exit(0);
        }
        /*parent*/
        close(sdnew);
    }
}

void PCMrecv(int sd)
{
    int pd[2];
    ssize_t sofar;
    unsigned char buf[MAXDATASIZE+1];
```

```
    if(pipe(pd) == -1)
    {
        perror("PCMrecv() pipe error: ");
        exit(-3);
    }

    /*sub child for execing PCMout*/
    if(!fork())
    {
        if(dup2(pd[PIPE_R], STDIN_FILENO) < -1)
        {
            perror("PCMrecv() dup2 error in fork: ");
            exit(-1);
        }
        close(pd[PIPE_W]);
        execl("./PCMout", "PCMout", (char *)0);
    }

    close(pd[PIPE_R]);
    /*Reads PCM data coming in from the socket
     *and passes it to stdout*/
    while(1)
    {
        if((sofar = recv(sd, buf, MAXDATASIZE, 0)) == -1)
        {
            perror("PCMrecv() recv call: ");
            exit(-1);
        }
        if(sofar == 0)
            break;
        write(pd[PIPE_W], buf, sofar);
    }
}

void mpgClean(mpg123_handle *mh)
{
    mpg123_close(mh);
    mpg123_delete(mh);
    mpg123_exit();
```

```
}

/*Main function for reading from the mp3 file and sending
 *the decoded data over a socket*/
void mp3Decode(int sd)
{
    /*Vars for mp3 decoding*/
    mpg123_handle *mh = NULL;
    int channels = 0, encoding = 0, error = MPG123_OK;
    long rate = 0;
    off_t samples = 0;

    error = mpg123_init();

    /*Creates a handler, opens the file to the handler,
and get the sound
     *files format*/
    if(error != MPG123_OK || (mh = mpg123_new(NULL,
&error)) == NULL
        || mpg123_open(mh, mp3File) != MPG123_OK
        || mpg123_getformat(mh, &rate, &channels,
&encoding) != MPG123_OK)
    {
        fprintf(stderr, "Error while creating, opening,
and formatting ");
        fprintf(stderr, "a handler: %s\n",
            mh == NULL ? mpg123_plain_strerror(error) :
mpg123_strerror(mh));
        mpgClean(mh);
        exit(-1);
    }

    /*Confirms the output format at signed shorts; this
is the default*/
    if(encoding != MPG123_ENC_SIGNED_16)
    {
        mpgClean(mh);
        fprintf(stderr, "Bad encoding: 0x%x!\n",
encoding);
```

```
        exit(-2);
    }

    /*Ensure that this output format will not change
     *(it could, when we allow it).*/
    mpg123_format_none(mh);
    mpg123_format(mh, rate, channels, encoding);

    printf("Creating 16bit PCM with %i channels
and %liHz.\n", channels, rate);

   /*As long as things are decoding properly continue to
send audio
     *down the stream*/
    error = MPG123_OK;
    while(error == MPG123_OK)
    {
        error = mp3send(mh, &samples, sd);
   }

    if(error != MPG123_DONE)
        fprintf(stderr, "Warning: Decoding ended
prematurely because: %s\n",
            error == MPG123_ERR ? mpg123_strerror(mh) :
mpg123_plain_strerror(error));

    samples /= channels;
    printf("%li samples written.\n", (long)samples);

    mpgClean(mh);
}

/*Decodes mp3 data to PCM and sends it over the network*/
int mp3send(mpg123_handle *mh, off_t *samples, int sd)
{
    int error = MPG123_OK, i, j;
    size_t sofar, buffer_size = MAXDATASIZE;
    unsigned char *buffer, *buffer2;
```

```
    /*Creates a buffer of the recommended output buffer
size.*/
    buffer = malloc(buffer_size);
    buffer2 = malloc(buffer_size);
    nullLine(buffer2, buffer_size); nullLine(buffer,
buffer_size);

    error = mpg123_read(mh, buffer, buffer_size, &sofar);

    for(i = 0, j = 0; j < sofar; i++, j+=DOWNSAMPLE)
        ((short *)buffer2)[i] = ((short *)buffer)[j];

    send(sd, buffer2, sofar/DOWNSAMPLE, 0);
    *samples += sofar/sizeof(short);

    free(buffer); free(buffer2);

    return error;
}

void sigchld_handler(int s)
{
   while(waitpid(-1, NULL, WNOHANG) > 0);
}

// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{
   if (sa->sa_family == AF_INET)
    {
      return &(((struct sockaddr_in*)sa)->sin_addr);
   }

   return &(((struct sockaddr_in6*)sa)->sin6_addr);
}
```

Final code for client header:

```
/**
 * @author James Fenley
 * @author Jon Law
 *
 * client.h
 *
 * Header for client.c
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/signal.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <termios.h>

#define PORT "3490"

#define MAXDATASIZE 1000

/*The terminals file descriptor*/
int termfd;

/*Flags*/
int playSong = 1;
short volume = 5;
int echoSong = 0;

/*Get sockaddr, IPv4 or IPv6*/
void *get_in_addr(struct sockaddr *sa);
void terminalOpen(char *term);
void terminalClose(struct termios *options);
```

```
void iohandler(int status);
void soundrecv(int sd);
void volumeControl(unsigned char *buf, int sofar);
void initport(int fd, struct termios *newops, struct
termios *oldops);
void nullLine(unsigned char *buf, int length);
void lowpassechofilter(unsigned char *buf, unsigned char
*xdelay,
                    unsigned char *ydelay, int *xindex,
                    int *yindex, int sofar);
```

Final program code for client:

```
/**
 * @author James Fenley
 * @author Jon Law
 *
 * client.c
 *
 * Streams PCM data from server. Performs basic DSP
operations on
 * the data and sends it back to the server for playback.
 */

#include "client.h"
#define DELAY 2

int main(int argc, char *argv[])
{
    /*Vars for the client network structure*/
   int sd;
   struct addrinfo hints, *servinfo, *p;
   int encode;
   char s[INET6_ADDRSTRLEN];

   if(argc != 3)
   {
       fprintf(stderr,"usage: ./client <hostname>
<tty>\n");
```

```
        exit(1);
    }

     terminalOpen(argv[2]);

     /*Hints specifies the type of connection the client
will
      *search for*/
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    if((encode = getaddrinfo(argv[1], PORT, &hints,
&servinfo)) != 0)
     {
       fprintf(stderr, "getaddrinfo: %s\n",
gai_strerror(encode));
       return 1;
    }

    /*Finds the first connection result matching our hint
and attempts
      *to connect to it*/
    for(p = servinfo; p != NULL; p = p->ai_next)
     {
       if((sd = socket(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1)
         {
          perror("client: socket");
          continue;
       }

       if(connect(sd, p->ai_addr, p->ai_addrlen) == -1)
         {
          close(sd);
          perror("client: connect");
          continue;
       }
```

```
      break;
   }

   if (p == NULL)
    {
      fprintf(stderr, "client: failed to connect\n");
      return 2;
   }

   inet_ntop(p->ai_family, get_in_addr((struct sockaddr
*)p->ai_addr),
         s, sizeof s);
   printf("client: connecting to %s\n", s);

   freeaddrinfo(servinfo); // all done with this
structure

    soundrecv(sd);

    shutdown(sd, SHUT_RDWR);
   close(sd);

   return 0;
}

void terminalOpen(char *term)
{
    struct sigaction saio;

    /*Open terminal device (touchscreen)*/
    if((termfd = open(term, O_RDWR | O_NOCTTY |
O_NONBLOCK)) < 0)
    {
        perror("opening tty: ");
        exit(-1);
    }
    else
    {
        fcntl(termfd, F_SETFL, 0);
```

```
    }

    /*Set the signal handler*/
    saio.sa_handler = iohandler;
    sigemptyset(&saio.sa_mask);
    saio.sa_flags = 0;
    saio.sa_restorer = NULL;
    sigaction(SIGIO, &saio, NULL);

    /*Set terminal for SIGIO*/
    fcntl(termfd, F_SETOWN, getpid());
    /*Make terminal asynchronous*/
    fcntl(termfd, F_SETFL, FASYNC);
}

void terminalClose(struct termios *options)
{
   /*Flush the terminal and return to its original
settings*/
    tcflush(termfd, TCIOFLUSH);
    tcsetattr(termfd, TCSANOW, options);
}

/*Handle SIGIO from terminal device*/
void iohandler(int status)
{
    int sofar;
    char sbuf[255];

    /*Read from terminal (touch screen)*/
    if((sofar = read(termfd, sbuf, 255)) > 0)
    {
        /*Null carraige return*/
        sbuf[sofar-1] = 0;

        //fprintf(stderr, "terminal: %s\n", sbuf);
        /*Play*/
        if((strcmp(sbuf, "11") == 0) || (strcmp(sbuf,
"s11") == 0))
```

```
            playSong = 1;
        /*Pause*/
        else if((strcmp(sbuf, "10") == 0) || (strcmp(sbuf,
"s10") == 0))
            playSong = 0;
        /*Volume*/
        else if((strncmp(sbuf, "l128:", 5) == 0) ||
                (strncmp(sbuf, "128:", 4) == 0))
            volume = 5 - (sbuf[sofar-2] - '0');
        else if((strcmp(sbuf, "20") == 0) || (strcmp(sbuf,
"s20") == 0))
            echoSong = 0;
        else if((strcmp(sbuf, "21") == 0) || (strcmp(sbuf,
"s21") == 0))
            echoSong = 1;
        tcflush(termfd, TCIFLUSH);
    }
}

void soundrecv(int sd)
{
    int sofar, xindex = 0, yindex = 0;
    struct termios newops, oldops;
   unsigned char buf[MAXDATASIZE];
    unsigned char *ydelay = (unsigned char
*)malloc(MAXDATASIZE*DELAY);
    unsigned char *xdelay = (unsigned char
*)malloc(MAXDATASIZE*DELAY);
    bzero(xdelay, MAXDATASIZE*DELAY);
    bzero(ydelay, MAXDATASIZE*DELAY);

    /*Initialize the RS232 port*/
    initport(termfd, &newops, &oldops);

    /*Process the data*/
    while(1)
    {
       if(playSong)
        {
```

```
            /*Get PCM packet via TCP/IP*/
            if((sofar = recv(sd, buf, MAXDATASIZE, 0)) ==
-1)
            {
                perror("soundrecv() recv call: ");
                exit(-2);
            }
            /*Check if server sent last packet*/
            if(sofar == 0)
                break;
            if(echoSong)
            {
                lowpassechofilter(buf, xdelay, ydelay,
&xindex, &yindex, sofar);
            }
            volumeControl(buf, sofar);
            if((sofar == send(sd, buf, sofar, 0)) == -1)
            {
                perror("soundrecv() send call: ");
                exit(-1);
            }
        }
    }
    fflush(stdout);
    free(ydelay); free(xdelay);
    terminalClose(&oldops);
}

void lowpassechofilter(unsigned char *buf, unsigned char
*xdelay,
                       unsigned char *ydelay, int *xindex,
                       int *yindex, int sofar)
{
    int i;
    short output;
    for(i = 0; i < sofar/2; i++)
    {
        output = (((short *)buf)[i]>>2)
               + ((((short *)xdelay)[*xindex])>>3)
```

```
                + ((13*((short *)ydelay)[*yindex])>>4);
            ((short *)ydelay)[*yindex] = output;
            ((short *)xdelay)[*xindex] = ((short *)buf)[i];
            (*xindex)++; (*yindex)++;
            ((short *)buf)[i] = output;
            if(*yindex >= MAXDATASIZE*DELAY)
                *yindex = 0;
            if(*xindex >= MAXDATASIZE*DELAY)
                *xindex = 0;
        }
    }

    void volumeControl(unsigned char *buf, int sofar)
    {
        int i;
        if(!volume)
            nullLine(buf, sofar);
        else
            for(i = 0; i < sofar/2; i++)
                ((short *)buf)[i] = ((short *)buf)[i]>>(5-
    volume);
    }

    /*Initializes the tty port*/
    void initport(int fd, struct termios *newops, struct
    termios *oldops)
    {
        /*Get the current options for the port*/
        tcgetattr(fd, oldops);
        tcgetattr(fd, newops);
        /*Set the baud rates to 115200*/
        cfsetispeed(newops, B115200);
        cfsetospeed(newops, B115200);
        /*Enable the receiver and set the local mode*/
        (*newops).c_cflag |= (CLOCAL | CREAD);
        /*Set 8N1*/
        (*newops).c_cflag &= ~PARENB;
        (*newops).c_cflag &= ~CSTOPB;
        (*newops).c_cflag &= ~CSIZE;
```

```
    (*newops).c_cflag |= CS8;
    /*XON/XOFF software flow control*/
    (*newops).c_iflag |= (IXOFF | IXON);
    /*Set the new options for the port*/
    tcflush(fd, TCIOFLUSH);
    tcsetattr(fd, TCSANOW, newops);
}

/*Get sockaddr, IPv4 or IPv6*/
void *get_in_addr(struct sockaddr *sa)
{
   if(sa->sa_family == AF_INET)
    {
      return &(((struct sockaddr_in*)sa)->sin_addr);
   }

   return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

/*Nulls a buffer*/
void nullLine(unsigned char *buf, int length)
{
    int i;
    for(i = 0; i < length; i++)
        buf[i] = 0;
}
```

Refer back to **Appendix D** for final PCMout program code. Final code remains unchanged.

Makefile:

```
targets = server client arm-client PCMout

all: $(targets)

# On largefile-aware systems you might want to use these
instead:
#MPG123_CFLAGS  := $(shell pkg-config --cflags
libmpg123_64)
#MPG123_LDFLAGS := $(shell pkg-config --libs
```

```
libmpg123_64)
# This works on sane setups where off_t is off_t, and
just that.
MPG123_CFLAGS  := $(shell pkg-config --cflags libmpg123)
MPG123_LDFLAGS := $(shell pkg-config --libs   libmpg123)
SND_CFLAGS     := $(shell pkg-config --cflags sndfile)
SND_LDFLAGS    := $(shell pkg-config --libs   sndfile)

# Oder of libs not that important here...
compile = $(CC) $(CPPFLAGS) $(CFLAGS) $(LDFLAGS)
$(MPG123_CFLAGS) $(MPG123_LDFLAGS)

server: server.c server.h
   $(compile) $(SND_CFLAGS) $(SND_LDFLAGS) -Wall -o
server server.c server.h

PCMout: PCMout.c PCMout.h
   gcc -Wall -lasound -o PCMout PCMout.c PCMout.h

client: client.c client.h
   gcc -Wall -g -o client client.c client.h

arm-client: client.c client.h
   arm-linux-gcc-4.1.1 -Wall -o arm-client client.c
client.h

clean:
   rm -vf $(targets)
```