# Barenaked Lasers

## Senior Design Final Report

### EE 41440 – Spring 2013

**Matt Clary**
**Tyler Gregory**
**Billy Kearns**
**Chris Newman**

# Table of Contents

## 1. INTRODUCTION

Musical instruments have come a long way throughout history, from early flutes to pianos to electric guitars and synthesizers. We plan on creating the musical instrument of the future—the Laser Harp. This is not our own idea, as laser harps have been built in the past (and even used in large concerts). Despite this fact, we are taking a different approach to the laser harp, as almost every other one that we have come across uses Arduino technology, ours will run completely through our Pic32 microprocessor. Most current laser harps are also custom built and are therefore very pricey and useful only for large-scale venues. Our laser harp will be accessible to beginning musicians at a relatively low cost. It will be portable and not rely on excessive equipment or an interface with a PC.

Our laser harp will utilize lasers and distance sensing technology to allow a user to play their favorite songs in a visually stunning way while emulating multiple instruments for an exciting audio experience. This project will incorporate numerous challenging aspects of electrical engineering, including microcontroller use and programming, digital signal processing, circuit design, signal control, and many more. We hope to demonstrate our mastery of these skills while producing a top of the line laser harp. The information herein outlines in detail the methodology, approach, and design decisions that were made in the process of developing a functional laser harp prototype.

### 1.1 PROBLEM STATEMENT

Oftentimes, people without musical gifts are left out of jam sessions and find it difficult to play along with their musically talented friends. Instruments such as guitars, keyboards, and drum sets are commonplace, and so someone without musical ability playing such instruments will have difficulty contributing musically to his or her friends' band. Our aim in this project is to design and build a new type of musical instrument that allows non-musicians without a musical background to generate in-key tones and play a diversity of songs on a versatile, easy-to-use, and cost-effective instrument. The users should be able to partake in and contribute to "jam sessions" with their more musically inclined friends without feeling left out. Although we do not have a real world problem to solve with widespread societal impact (such as reduced energy consumption or carbon emissions) we intend to design and create a commercially successful laser harp for pure entertainment. Our laser harp will be a new, futuristic musical instrument that will be easy to play, visually impressive, and able to emulate multiple instruments.

### 1.2 PROPOSED SOLUTION

We propose to design and build a "laser harp." Our laser harp will be designed on the basic concept of having multiple lasers shining vertically next to each other and having the instrument generate a tone whenever the user covers up one of the lasers with his or her hand. We will use an infrared sensor placed next to the laser source to detect when the user's hand is covering the beam and how far away it is from the source. This will generate a signal that, when processed through the audio processing equipment, will generate the desired tone. The laser harp does not require the user to be coordinate, as any type of disruption to the sensor will trigger a note to play. In this way, users do not need to know the subtleties of posture or good form required to play traditional instruments such as the piano or guitar. Users also do not need to know music theory, as each note is automatically played in key. A multitude of different instruments will also be available to users, allowing them to contribute to any group of musicians and any genre of music.

Certain aspects of our final design met expectations and certain aspects did not. The strengths and shortcomings of our project are discussed in more detail later in this document, but here we present a brief overview of how our design met expectations. The finished product played pleasing musical notes based on the sensor that was covered. It also featured a user interface that allowed the user to select which instrument to play. These features met all expectations and allow a user to play along with his more musically inclined friends. However, the lasers we used were not visible in a well-lit environment and required the use of fog in order to be visible. They were also unreliable. These factors diminished from the overall impressiveness of the finished product.

## 1.3 SYSTEM REQUIREMENTS

### 1.3.1 POWER SUPPLY

Our laser harp will need a power supply in order to run the lasers, IR sensors, speaker output, and associated processing and conversion circuitry. The IR range finders require anywhere from 3-5 V for operation. The lasers we used require a 3.3 V source. The microcontroller we used requires a 3.3 V source. In order for the laser harp to be long-lasting and provided with sufficient power, we chose to use an adjustable 4.4A power jack connected to a wall outlet, as well as two 3 V battery packs. The wall outlet initially powers the board/microprocessor. A voltage regulator is then used to convert the 5V to 3.3V for use by the microcontroller. This avoids the lossiness inherent in voltage dividers. Each battery pack will be used to power the lasers and the IR sensors. We chose to use battery packs because the lasers and IR sensors were drawing too much current through our voltage regulator, causing our board to overheat and fail. These battery packs will run off of two D batteries.

### 1.3.2 RANGE DETECTION

In order to play the instrument, a mechanism needs to be used to sense when the user's hand is covering the laser. Since the microcontroller we have identified has easy-to-use analog to digital conversion capabilities (as demonstrated in the project demo) we will use a detector with an analog voltage output. The distance vs. voltage output characteristics of this device will also need to be steep in the distance ranges of interest, i.e. the analog voltage output should change a lot with a small change in hand distance over the operation heights. This allows the microcontroller to distinguish hand height with a high degree of precision, so that we can use a threshold for sensing the user's hand. The sensors we decided to use have a maximum range of up to 150 cm, but that is if they receive 5 V input voltage. Because we will only be supplying 3.3, the range was reduced to about 2 feet.

### 1.3.3 VISIBLE SENSOR LOCATIONS

In order to play notes, the user must interfere with the infrared beam emitted by the range finders. The user must therefore be able to see the locations of the infrared beams in order to play the desired notes with precision. Infrared light is out of the visible spectrum, so our solution is to use lasers in the visible spectrum to indicate the location of the IR beam. The lasers need to be located very close to their corresponding range finders (so need to be small) and need to be strong enough to be visible in a dark room. However, they cannot be so powerful as to be potentially dangerous to the user. Thus, we chose 3.3 V green lasers that run at 40 mA. We acquired these lasers from green laser pointers that we stripped the casing off of. By doing this, we also were able to use the laser pointers focusing lenses and biasing circuitry.

### 1.3.4 GENERATING AUDIO SIGNAL FROM SENSOR OUTPUT

Once the analog signal is generated by the IR range finder, this signal must be converted into an audio signal whose frequency is determined by the specific laser being covered. The analog voltages are read by a microcontroller. The voltages are converted to a digital value and stored sequentially in buffers in the microcontroller. If the user's hand covers the laser, the voltage will pass a certain threshold value and the microcontroller (reading the digital information in the register) will set a flag corresponding to that channel, indicating that a note should be played. This flag will retrieve a HEX array that represents a .wav file of the desired note. This HEX array will be sent through the processor adjusting the duty cycle of a pulse width modulation signal output, effectively creating a sinusoidal wave which will recreate the desired note.

### 1.3.5 SOUND OUTPUT

In order to generate musical notes, our laser harp needed a means of converting the electrical audio signal into audio waves. This was accomplished using speakers. The speakers needed to be volume adjustable (achieved using an amplifier), audible from a reasonable distance, and small enough to fit in the laser harp packaging. For these reasons, we chose a single 20W rated 40W max 103mm x 103mm x 50mm speaker.

### 1.3.6 COMPUTER INTERFACE FOR PROGRAMMING

The laser harp microcontroller had to interface with a PC in order to be programmed to complete the particular tasks required of it. This required the use of a serial connection to the microcontroller and pins on the board to connect to an external programmer such as the PICkit 3. This connection allowed the basic functionality of the microcontroller to be programmed and it will also allow the user to make customizable alterations later on. There is also a set of serial pins to attach a memory card slot if the user wanted to be able to store more than three instruments on the device.

### 1.3.7 USER INTERFACE

Our laser harp required a means for the user to select which instrument they wanted to play. To accomplish this, we used a 4x20 LCD display with three buttons. With this display, the user would be able to read an "About" tab, which gave a general how-to-use summary, as well as see and select which instrument they wanted the tones to be from.

### 1.3.8 DURABLE AND PORTABLE PACKAGING

The sensitive electronics of our laser harp needed to be protected from environmental wear and tear, and the users had to be protected from the various electronics inside. In order to do this and hold the entire system together, we had to construct some sort of external packaging. It had to be durable and compact but lightweight and cheap. We also wanted it to be able to be conveniently transported. We therefore chose to create a wooden box with one side being a hinged door. This allows one to access it if it needs to be fixed. The box was made from ¼" MDF board, and it is 2' x 8" x 8". By using MDF board the entire device is not too heavy and the dimensions allow it to be carried by just one person safely. We also needed the lasers to be mounted so as to point vertically up in the air. We created a bridge across the top of the box which aims each laser and sensor at a slightly different angle, creating a fanned out appearance. This allows the user to easily see the lasers and cover them individually to play notes.

## 1.4 REVIEW OF FINAL RESULTS

After completing the project, the final result was successful. We were able to create a laser harp which sensed a user's hand breaking a laser beam, and output a clear audio signal. The lasers we used were not of the highest quality and tended to vary in strength from barely lasing to full power. As a result, it was rare to see all eight lasers working at full strength. Our IR sensors worked great and were able to accurately sense the user's hand as long as it was within the sensor's threshold.

We had a goal that we would like the user to be able to select from multiple instruments and we were also able to accomplish this goal. In our final demo, we were able to show that our laser harp could not only produce nice, simple tones of a piano and clarinet, but that it could also generate very complex sounds. To show this off, we chose to use animal sounds. Our laser harp was able to very clearly play animal sounds including that of a dolphin, a jaguar, a dog, and a duck. Quality audio generation was a very important goal of ours and we feel that we fully accomplished this.

We also wanted to create a user interface that was easy to use and helpful for a user who had never seen a laser harp before. By using the 3D printer, we created a face plate to hold the buttons and LCD screen as well as help the user know which buttons do what. The LCD screen also had an “About” menu which instructed a user what they had to do to generate music with the laser harp. Lastly, the LCD allowed the user to easily change the instrument that was being heard. As soon as a different instrument was selected, the user could immediately hear the new instrument. Originally, we had hoped to have a master volume control with our user interface, but unfortunately, amplifier issues made this something that we were unable to accomplish.

# 2 DETAILED PROJECT DESCRIPTION

## 2.1 SYSTEM THEORY OF OPERATION

Our laser harp works by monitoring the position of the user’s hand and using this information to output the desired musical note. The first step in the operation process is to switch the board on, supplying power to the microcontroller and LCD screen. Also, both battery packs need to be connected to supply power to the lasers and the sensors. When conditions are dark enough and a fogger is used to saturate the air with particles, the lasers are clearly visible and indicate the locations at which the infrared sensors can detect a reflection. The instrument defaults by starting in the piano instrument setting. When the user places their hand above one of the infrared sensors, the sensor outputs a voltage signal proportional to the number of detected reflected photons over the total number of photons emitted by the infrared LED. These eight signals are processed in a scanning analog to digital conversion process on the microcontroller. The PIC32MX795F512H microcontroller serves as the main control hub for the entire system. The software on it analyzes the 8 signals and identifies the strongest signal. It then finds the corresponding sound file stored within its memory and begins to sequentially output each byte of data by adjusting the duty cycle of a pulse width modulation signal. The series of average voltages of the PWM signal represents the series of amplitudes of each sample of the sound file. The signal traces out an audio signal which is passed through an amplifier into a speaker. Throughout all of this the microcontroller constantly monitors three user interface buttons connected to i/o pins. These buttons control the operation of the LCD screen, which enables the user to select different instruments.

## 2.2 SYSTEM BLOCK DIAGRAM

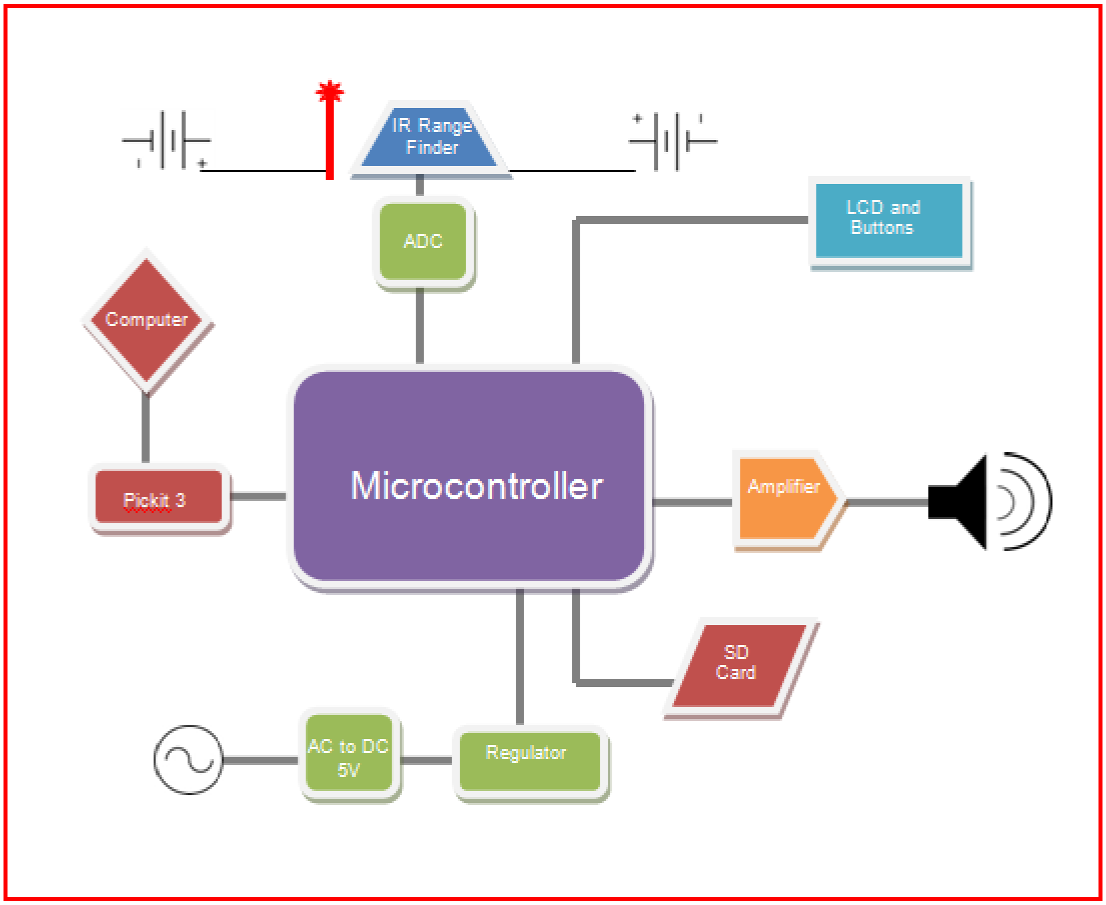


Figure 2.2: System Block Diagram

## 2.3 DETAILED OPERATION OF SUBSYSTEMS

### 2.3.1 POWER SUPPLY

The laser harp needed to supply 3.3 V to three separate places within the device. The board, the sensors, and the lasers were all going to be run off of 3.3 V. Unfortunately, the lasers and sensors could not be powered from the board because they drew too much current and caused the board to overheat. This problem forced us to find a new solution for powering everything. To satisfy all of our power needs, we had to use a power jack connected to a wall outlet as well as two separate 3V battery packs. The power jack was able to connect right from the wall into an input plug on the board. Once plugged in, the voltage went through a voltage regulator where it was brought down to 3.3 V, which is what we needed it at for the microprocessor.

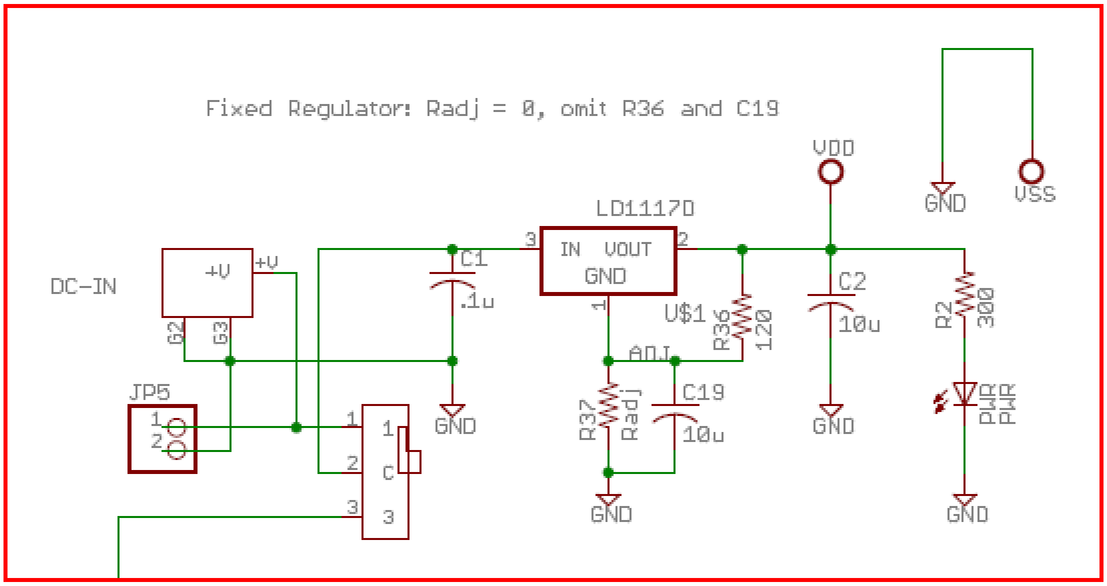


Figure 2.3.1: Power Supply Schematic

The 3.3V voltage regulator powered the microcontroller and was also used to power the LED buttons on our separate button board for our LCD screen. The board we used also included a 5V pin next to one of the SPI ports that we used to power our LCD screen. Our amplifier ran separately on its own 9V battery. We also used batteries at 3.3V to power both the lasers and the sensors. This was to reduce the current draw through the board, which was unable to handle it.

The LCD screen ended up being the greatest current drawer on our power source, however we had originally intended to power the lasers, sensors, and amplifier with the board as well. After hooking up some of these devices, we discovered the voltage regulator on the board was getting extremely hot and overworked. We purchased battery holders for D size batteries and found they were outputting about 3.2V when placed in series, well within the voltage requirements of the lasers and sensors. We eventually discovered the necessity of having a single common ground in using so many different power sources. Before we connected our sensors to the same electrical ground as the microcontroller, they would output signals that would register far too strong when compared against the reference voltages of the microcontroller's analog to digital converter. Once we connected all of our separate grounds together the sensors continued working correctly.

We also ran into power issues when we attempted to integrate our SD card subsystem for file storage. When we used the SD card, the voltage regulator quickly became overworked and hot, which did not make sense since the card does not consume an excessive amount of power. We ended up concluding that there must have been a short circuit somewhere in the SD card's wiring causing an excessive current draw from the voltage regulator. We only discovered this shortly before the deadline for having our project ready for demonstration and were not able to make use of our SD card subsystem. Instead of storing our audio files externally we decided to hardcode them into the program and flash memories of the microcontroller.

### 2.3.2 LASERS

The only function of the lasers in the laser harp are to mark where the user is supposed to place their hands. The only connections that the lasers have are a 3 V power supply and ground. The lasers do not need to interface with any other aspect of the laser harp, they simply must turn on when the harp needs to be used and turn off when it is not in use.

The lasers made us make many decisions that had to account for functionality, price, and engineering. We decided that we would use green lasers as they are considered the easiest laser to see with the naked eye. Unfortunately green lasers are much more expensive than red lasers. We also had to decide if we should buy laser diodes or try to figure out how to utilize the laser diodes built in to laser pointers. If we used laser diodes, we would have to create biasing circuitry and also would need to focus the beams and construct some kind of housing for all of this, but the laser diodes would be more reliable. If we chose to use laser pointers, they would come with a housing that contained the lenses for focusing and the biasing circuitry, but the quality would be less as we would not be able to buy nice laser pointers.

We ended up buying green (532 nm) laser pointers and stripping off most of the unneeded parts and casings. Figure 2.3.3 shows a general diagram of the laser pointers that we used. If you look at the diagram, we essentially only used from the circuitry to the right. We attached a ground lead past the button switch and the positive lead to the positive pin on the bottom of the circuitry board. The figure also shows the elaborate housing that the laser pointer provides.

The only issues we ran into with the laser pointers is that they tended to fluctuate with their output. The leads were soldered to the right connections, so it was not the connections, it just had to do with the quality of the laser diodes that came in these laser pointers. It was rare to have all eight of the lasers at full power at the same time.

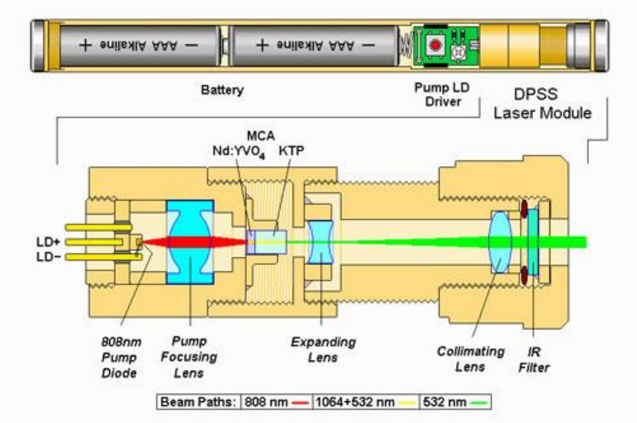


Figure 2.3.3: General Diagram of laser pointers used

### 2.3.3 SENSORS

In order to determine whether or not the user wanted to play a note, we used infrared range finders. The IR range finders that we used had a sensing distance of 20-150 cm. This would allow the player to stand up while the laser harp itself is resting on the floor. Also, the higher powered range finders have a better resolution and are more suitable for our project. The sensors require 3-5V and will output an analog signal between 0 and 2V.

Before we ordered the sensors that we used, we experimented with a lower power sensor. The prototype sensor had a sensing distance of 30cm and only had an output signal of up to 1.6V. This sensor worked great for testing our code and design, however, our final project required more resolution and a greater sensing distance. We had originally hoped to allow the user to change the volume based on the proximity of the hand to the sensor, which is why we needed a greater resolution. This functionality, however, did not make it to the final product, but the greater resolution did allow for a more reliable sensing of the hand.

We had very few problems with the range finders; they were very reliable. One problem that did occur was when the user would obstruct the sensor near its threshold. This would result in a distorted signal being sent to the microcontroller. One other issue that occurred was when we introduced fog in order for the user to see the lasers. If the fog was too thick in the area over the sensor, a false reading would occur. These issues might have been resolved with a 'debouncing' mechanism within the code, where the received signal is averaged over a period of time to ensure that a false signal is not translated as someone covering the sensor.

### 2.3.4 ANALOG SENSOR SIGNAL TO DIGITAL CONVERSION

In order to make use of the signals from each of our infrared range sensors, we needed to execute an analog to digital conversion for the sensors. In our original testing with the first prototype sensor, we had an easy time running a single analog to digital conversion. We ran into far more problems later on when trying to connect all eight to the microcontroller at the same time.

The PIC32 microcontroller series has the ability to scan through multiple analog inputs (up to 16), spending a short time at each one before moving on to the next. We set our code to scan through our eight sensors hooked up to analog pins 2, 3, 4, 5, 6, 7, 9, and 13. The other analog pins were needed to interface with other peripheral devices via SPI. We set our code to spend slightly longer than the minimum conversion time on each sensor to make sure we had an accurate conversion. Our analog to digital period was set as four clock pulses while our sampling time was set at two periods, so each sample time lasted eight clock pulses. The longer the analog signal is sampled, the more precise the digital conversion can be.

Once each sensor is converted to a digital value, it is stored in a buffer until the next time that sensor's input is checked. After each scanning sequence of all eight sensors, we switched off the conversion and saved the buffer contents into new variables before enabling the conversion to repeat its scan so that we did not end up reading from the buffers while they were being written to. We could then use these variables to check the strength of each sensor's signal. We coded our harp to play the note corresponding to the single sensor with the strongest signal. Because of the pulse width modulation method we used for sound generation in tandem with .wav files, we could not overlap multiple sound files and generate chords.

The flowchart for the code used in this section is shown below:

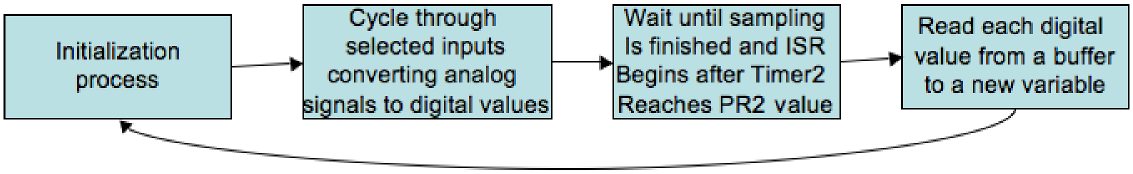

Initialization process
Cycle through selected inputs converting analog signals to digital values
Wait until sampling Is finished and ISR Begins after Timer2 Reaches PR2 value
Read each digital value from a buffer to a new variable

Figure 2.3.4: Flowchart for sensor analog to digital conversion

2.3.5 GENERATING AUDIO SIGNALS

To generate our audio, we relied on the concept of pulse width modulation. We began by finding .wav files on the internet that provided faithful reproductions of the instruments we wanted to include. We decided to use piano as our primary instrument voicing since it is considered a very standard instrument. We decided on clarinet as a secondary instrument since it sounded somewhat electronic and provided an even volume level throughout each tone sample. We took advantage of this to make the files loop; when the clarinet instrument is selected and the user's hand remains over the sensors, the audio output sounds constant. We found suitable audio samples at the University of Iowa Electronic Music Studio website included in the references section. These files were high quality and available for use in our project without any copyright restrictions. We also found various animal noises online to demonstrate the versatility of our sound generation method. Once we had obtained all the proper .wav files, we converted them all to a common sampling frequency of 16000 samples per second and converted them to the 8-bit unsigned .wav format to reduce the number and size of samples we needed to handle. We then used a free software program called Hexedit to convert the sound files into hexadecimal data values. We placed these values in an array for each individual note and stored each array in the memory of the microcontroller.

Our algorithm for pulse width modulation allowed us to essentially use a digital i/o pin to output an analog signal. By varying the duty cycle of each PWM pulse, we could control the average voltage seen by the audio output subsystem. This process is illustrated in the following graph:

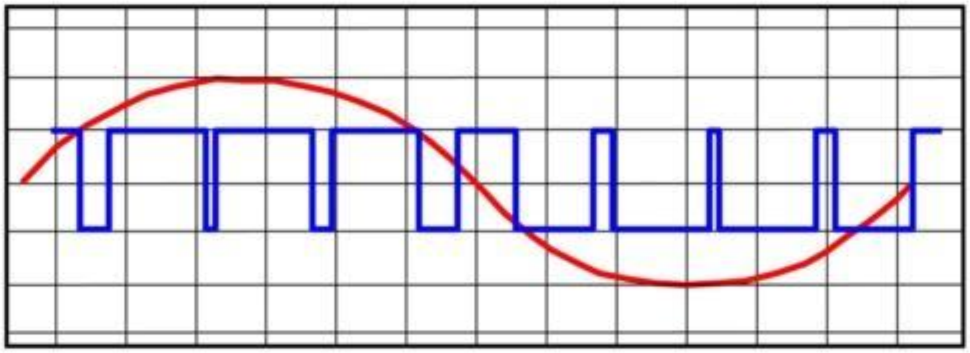


Figure 2.3.5: Image illustrating pulse width modulation
http://eetimes.com/design/audio-design/4015803/Digital-amplifier-overview

The blue graph represents the high and low states of the digital PWM pin, while the red graph represents the average voltage output produced by the blue signal. This entire process was interrupt driven in the software and controlled by the microcontroller's Timer2. The flow chart for this software process is show on the following page. We created an algorithm to scan through a selected array and convert each sample's individual 8-bit hexadecimal value into a corresponding duty cycle for the PWM output. The main equation we wrote that allowed us to do this was:

$$OC1RS = (hex+1)*((PR2+1)/256)$$

This equation takes an individual hex value (hex), and manipulates it into a duty cycle duration (OC1RS). PR2+1 is the total period for each PWM pulse, OC1RS is the buffer for duty cycle values to be stored, and the 256 corresponds to the 8-bit nature of each entry in the array (2^8 = 256). Basically, we divide the total period into 256 sections and make the duty cycle duration proportional to the amplitude measurement of a sample given by its array entry. This equation is part of a loop that lasts as long as the loop counter is a value smaller than the size of the array.

The generated PWM signal was output through the OC1 pin of our microcontroller, which was hooked straight into the amplifier then the speaker. We were very pleased with the quality of audio we obtained through the PWM method; we did not need to obtain any external hardware and the clarity of our sound output was very good.

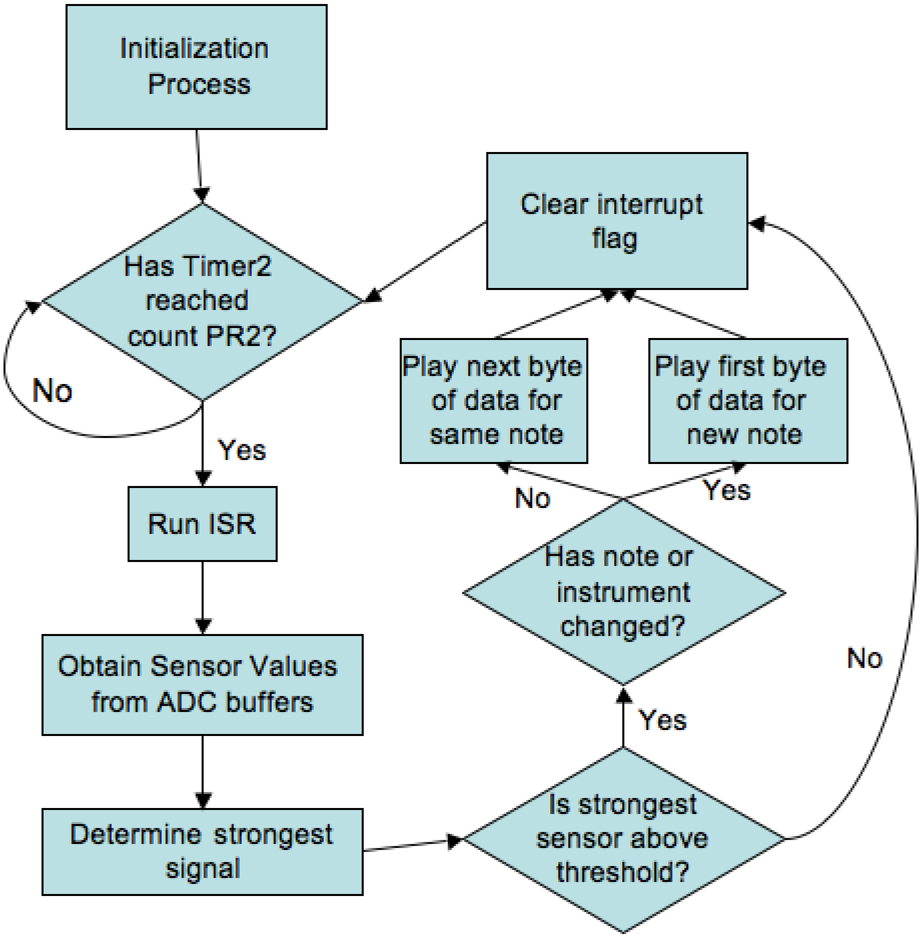

Initialization Process
Has Timer2 reached count PR2?
No
Yes
Run ISR
Obtain Sensor Values from ADC buffers
Determine strongest signal
Is strongest sensor above threshold?
Yes
No
Has note or instrument changed?
No
Yes
Play next byte of data for same note
Play first byte of data for new note
Clear interrupt flag

Figure 2.3.5.1: Flowchart for audio generation

## 2.3.6 AUDIO OUTPUT

After the audio signal was generated using PWM, pin OC1 was used to connect to the amplifier. Since we were unable to get more power from our board, we had to use an external amplifier with a 9V battery in order to amplify our audio signal sufficiently. The amplifier connected between pin OC1 and the speaker.

The speaker was mounted at the very front of the box to allow for optimal audio clarity. The speaker we used was a 20W, 8 ohm speaker which we hooked up with speaker wire. One of these wires connected directly to ground and the other wire contained the audio signal.

The amplifier was contained in a box with the battery and the entire casing was placed inside our laser harp housing. In order to connect to the amplifier, we needed to attach 3.5mm cables to both the wire from OC1 and the speaker wires. Where OC1 is only one wire, we connected another wire to ground that would serve to eliminate some noise in the audio. This amplifier provided the functionality that we needed by making the speaker play the music distinctly louder.

It is unfortunate that we could not use an on board amplifier to enhance the audio, but given the circumstances, we were unable to do so. If further improvements were to be made, we would provide an amplifier that has the ability to generate sound that is much louder. The whole purpose of a laser harp is to create an enjoyable music show. In essence, the louder the better. Also, an amplifier that had greater audio functionality could be used to greater control the sound output and tweak it to the user's desire.

### 2.3.7 USER INTERFACE

In order for users to interact with the laser harp and select the instruments they wish to play, a user interface is required. This subsystem must be easy to use and familiar to the user. It must allow a user to scroll through a number of options and make selections while providing visual confirmation of their choices. It also must interface with the microcontroller and other subsystems, so that the user input can affect changes in the overall system.

In order to meet these requirements, we chose to use a Newhaven 4x20 LCD display, along with an external board we designed to house three LED push buttons for user input. The LCD and board are connected to our main board via jumper wires. The microcontroller communicates with the LCD over an SPI interface, much like the SD card, while the buttons are connected to simple digital i/o ports whose status is read by the microcontroller.

The decision to house the buttons and LCD on external boards was made because we mounted the user interface on the top of our laser harp package for easy access while the main board was placed inside. LED buttons and a bright LCD screen were chosen so that users can see the interface even if they are playing in the dark. We chose the Newhaven 4x20 screen because of its ease to program and its SPI interface. The larger screen enabled us to display multiple instruments in a single menu.

#### 2.3.7.1 EXTERNAL PUSH BUTTON BOARD

The LED push buttons for the user interface are housed on an external board that is mounted on top of the laser harp package. The button board is connected to the main board via jumper wires and ensures easy access to the buttons for the user. The board is positioned adjacent to the LCD display so that the user can scroll through the menus and make selections easily. A schematic and board diagram for the buttons is shown below.

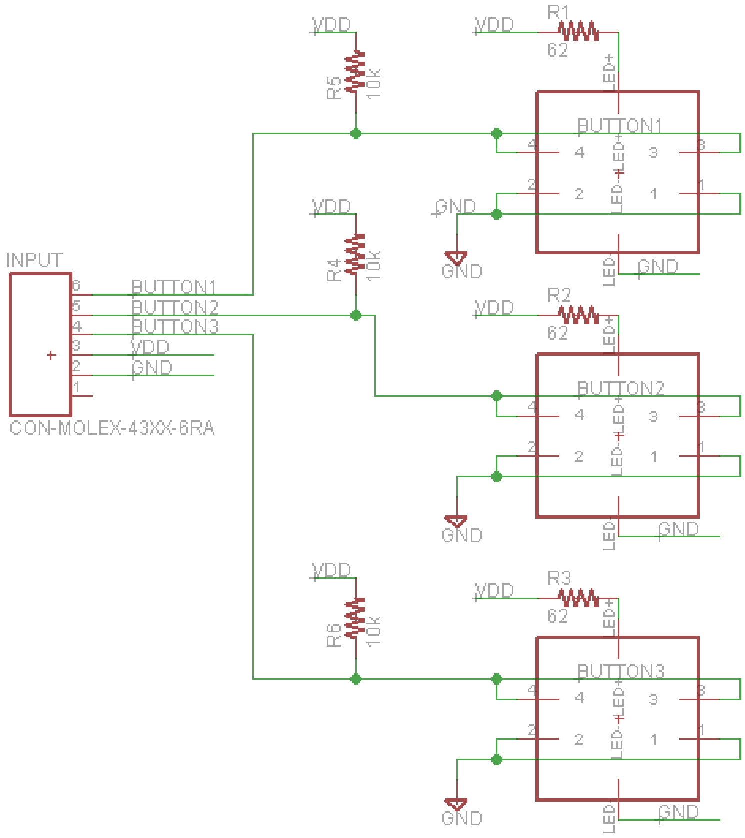


Figure 2.3.7.1.1: Button Schematic

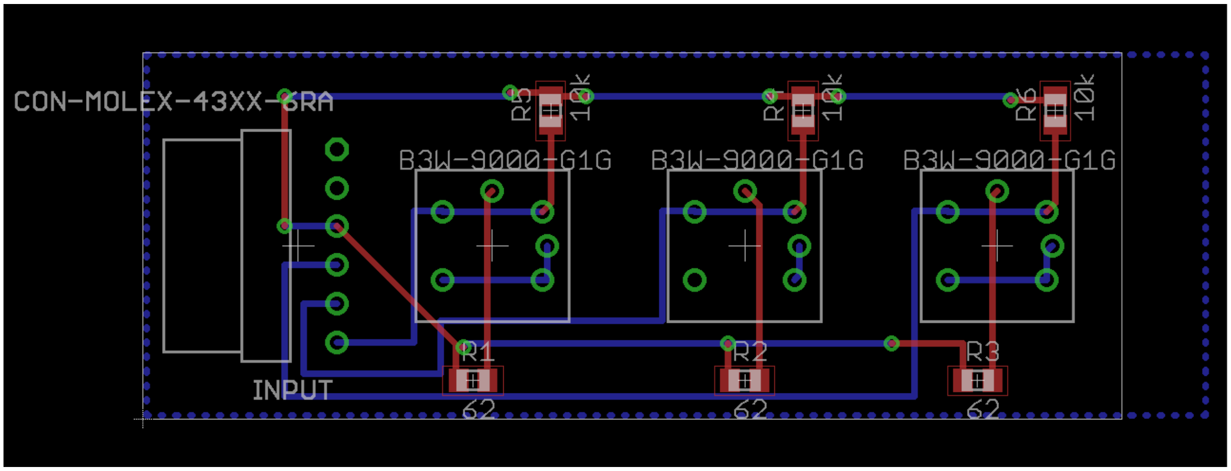


Figure 2.3.7.1.2: Button Board Layout

The board includes three Omron Electronics LED push buttons arranged in a row. The buttons have two separate Vdd (3.3V) and ground connections: one for lighting the LED and another for switching. A 62 ohm current limiting resistor is placed at the LED driver input, preventing the LEDs from drawing too much current and burning out. A 10 kohm pull-up resistor is placed between the output pin and Vdd. This pulls the output to a high level (Vdd) when the button is not pressed and a low level when the button is pressed and the output is tied to ground. It allows current to flow during switching so that the microcontroller pins are not required to source the current. A six pin molex connector on the board interfaces the buttons to the main board via jumper wires. Button 1, button 2, button 3, ground, and Vdd are connected to pins F1, F0, and D7 on the main board, respectively. Two buttons are SCROLL buttons (F1, F0) and can be used by the user to scroll through the several options on the lcd display. The other button (D7) is the SELECT button and is used to choose an option.

After soldering the components to the board, we noticed that the SELECT button was stuck in the "pushed" position. We determined that the button had melted as a result of excessive heat from the soldering iron. Since we did not have extra LED buttons, we used a simple push button instead for our final demonstration. This actually served to differentiate the SELECT button and does not significantly affect the user's experience.

#### 2.3.7.2 LCD HARDWARE

The LCD display our team used was the Newhaven 4x20 serial liquid crystal display module, part number NHD-0420D3Z-FL-GBW-V3. A link to the documentation is included in the references section. The display required a 5V power supply that was provided by the main board. We chose to operate the LCD screen in SPI mode, rather than I2C, because we understood how to establish that interface with the microcontroller from our work on the SD card. In order for the Newhaven board to operate in SPI mode, however, we first needed to ensure that the proper jumper connections were made on the board. For SPI mode, R1 is an open circuit and R2 is a short circuit. When a low slave select (or chip select) is set, the idle clock level should be set to high and data is sampled on the rising clock edge at a maximum frequency of 100 kHz. These settings were set in the SPI configuration register in software.

The J2 connector was used to make the jumper wire connections from the LCD display to the main board. Pin 1 is the slave select and we connected it to pin D6 on the microcontroller, a standard digital output pin. Pin 6 is for the 5V power connection and pin 5 is the ground connection. The remaining pins are associated with the microcontroller's SPI3 interface. Pin 4 is the data in port and is connected to pin D3 on the microcontroller. Pin 3 is for the clock signal and is connected to pin D1 on the microcontroller. Pin 2 is for data out and corresponds to pin D2 on the microcontroller, but is not used for the LCD display since it does not output any serial data on the SPI interface.

#### 2.3.7.3 USER INTERFACE SOFTWARE

Using the LCD display and pushbuttons, the user has the ability to navigate through a number of screens to select the instrument of choice, adjust settings, and learn about laser harp operation. The software flow of the user interface is defined in the main function, but there are several LCD-specific functions, as well. The code is placed in an infinite while loop and a variable is used to keep track of the current menu. Each menu has a corresponding "if" statement that polls the menu variable and displays the corresponding letters on the LCD screen. For each menu, a function is called to print the menu title at the top of the screen and any related information below it. The push buttons pins (F0, F1, and D7) are also constantly checked, waiting for user input. If a button is pressed, the corresponding changes will be made to select a new instrument, adjust settings, or change menus. The user can scroll through menus using the "up" and "down" buttons and see which option is selected through the use of a blinking cursor that hovers over the current choice. By keeping track of the cursor's position, the software will know which option is selected when the "select" button is pushed. Based on the user's choice of instrument (which defaults to the piano and is set by the LCD functions) the main function loads the required note to be played into an array when the user covers a sensor. A detailed flow chart of the user interface software is shown below.

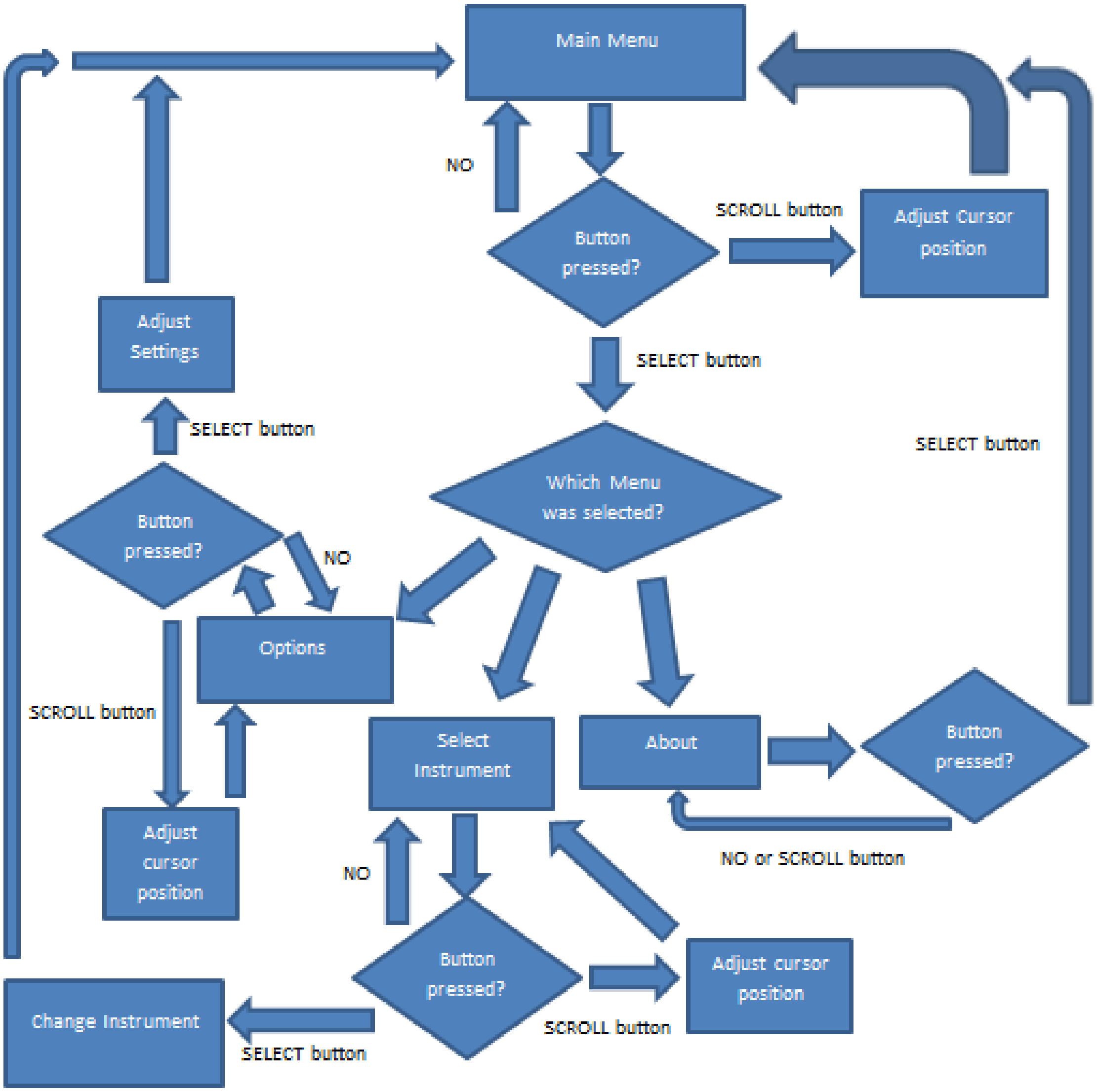


Figure 2.3.7.3: User Interface Flow Chart

The list of functions employed for the user interface includes:

void init_lcd_spi()

This function enables the microcontroller SPI3 interface for compatibility with the LCD screen. The clock (D1), data out (D3), and chip select (D6) are configured as outputs, clock frequency is set at 100 kHz, and the appropriate bits are set in the SPI3CON register.

unsigned char display_char(unsigned char)

This function is the low level routine that directly interfaces the microcontroller's SPI 3 module with the LCD screen. It takes a single 8 bit byte as input and writes this byte to the the SPI3BUF shift register, which clocks the byte out on the data out pin to the LCD screen. The function then waits for the transaction to be complete, then clears the contents of the SPI3BUF. This low-level read/write routine is essentially the same as the xchg_spi function for the sd card.

void display_string(unsigned char[])

This function uses the display_char function to write the 20 element character array argument to the lcd screen.

void send_command(unsigned char, unsigned char)

This function is used to send commands to the LCD. The Newhaven LCD screens respond to a defined set of commands, in addition to simply writing characters, that adjust various LCD settings. This project uses commands to clear the screen, set cursor position, and set the cursor so that it is blinking. The command must be preceded by a start token (0xFE) that tells the screen to expect a command. Certain commands require the user to send a corresponding parameter. This sent immediately after the command. All commands also take a certain amount of time to be processed by the screen. Therefore a delay function is included after the command is sent.

int button_debounce(int)

This function checks the status of the SCROLL buttons and returns the updated cursor position. By incorporating a flag that is set if the button is pressed, and then waiting in a while loop until the button is released, the function debounces the button.

int screen_select(int, int)

This function checks the status of the SELECT button and debounces it. If the button is pressed, the next screen variable is updated based on the position of the cursor. Based on the current menu, the function then returns the next menu the LCD should display. If the current menu is the instrument select menu, then the return information is used to determine the next instrument to be played.

void display_mainmenu()

Clears the LCD, displays the characters for the main menu, and sets the cursor position.

void display_instrmenu()

Clears the LCD, displays the characters for the instrument select menu, and sets the cursor position.

void display_options()

Clears the LCD, displays the characters for the options menu, and sets the cursor position.

void display_about()

Clears the LCD, displays the characters for the about menu, and sets the cursor position.

### 2.3.7.4 TESTING THE USER INTERFACE

In order to test that our user interface was functioning properly, we first attached the USBee Logic Analyzer leads to the microcontroller SPI3 pins (clock, data out, and chip select). We verified that the SPI interface was functioning properly and the correct 8 bit ASCII characters were being transmitted. We also made sure that the configuration bits for clock frequency, idle clock level, and transition edge were set correctly.

After verifying the functionality of the microcontroller output, we connected the SPI3 pins to the corresponding pins on the Newhaven LCD display and attempted to write individual characters. We created a separate MPLAB project specifically for this purpose, and used the low level SPI function we had developed for interfacing with the SD card. After ensuring this worked by viewing the display, we systematically added and tested higher level functions for sending commands and writing strings of characters. We continued with this strategy until the entire subsystem was complete.

One thing we noticed while testing the display is that, if a user desires to write a string of characters, he must define the array of characters as follows:

```
char char_array[] = {‘A’, ‘B’, ‘C’};
```

rather than:

```
char char_array[] = {"ABC"};
```

When defined by the second method, the LCD display will display most of the characters correctly, but a few may be corrupted. In this case, the displayed characters are also corrupted when the user attempts to scroll through the menu options. This is strange since, in the C programming language, a string(“ABC”) is nothing more than a character array (‘A’, ‘B’, ‘C’).

### 2.3.8 SD CARD EXTERNAL MEMORY

The PIC32MX795512H microcontroller we used had a maximum internal flash memory capacity of only 512 kilobytes. This was enough space to store at most three instruments worth of sound data (or 24 notes total). However, our goal was to allow the user to play more than just three instruments, and even be able to load their own sound data and play it. In order to accomplish this, we needed to include some sort of external memory.

Our decision to use an SD (secure digital) card for external storage was based on a number of design constraints. The memory needed a high enough data access rate so that the microcontroller could retrieve the sound data quick enough. When a new note was played, the microcontroller would look to a specified address in memory and store that note's data in an array which would be used by the PWM module to set the duty cycle and play the note. If the access rate was too slow, there would be a significant lag in the time between covering a sensor and hearing the note.

The memory also needed a large enough capacity to store a significant number of notes. We estimated this value to be a minimum of several hundred megabytes. This would allow the user to load their own instruments and have the potential to select from dozens of sounds. Furthermore, the external memory needed to be easily accessible from the microcontroller and have the ability to be both read from and written to by the user.

The SD card met each of our subsystem requirements as an external memory source. The specific model we used was a 4GB SanDisk Ultra SDHC (SD High Capacity) card with a baud rate of 30 MB/sec, which we purchased from BestBuy. This card meets the SD Association Specification standards set forth in the simplified specifications link, included in the references section. Another online resource that was particularly helpful in developing the SD card subsystem is also included in references.

This card choice exceeds our required memory capacity with 4GB of total space. This is approximately enough memory to store 180,000 notes, or 22,000 instruments. This well exceeds the number of sounds any user might wish to incorporate, but since the cost of increased memory space on SD cards is relatively low, we decided to purchase the higher capacity card. The data access rate of 30 MB/s is also quick enough for our uses. We verified on the logic analyzer that the time required to send the appropriate commands and read a full note from the SD card into an array is approximately 60 ms. If a new note is read each time the user covers one of the sensors, this means a 60 ms delay will be introduced which is practically imperceptible to human hearing. Lastly, the SD card could be communicated with using standard SPI (Serial Peripheral Interface). The microcontroller supports an SPI module with specified data in, data out, and clock pins. Plus, this expertise could be utilized to interface with the lcd screen, which also requires either an SPI or I2C interface.

Though our team successfully developed a working subsystem prototype for utilizing the SD card, we were not able to incorporate it into our final demonstration. After verifying the functionality of both the read and write functions using the logic analyzer and writing the data for each note to a known address in memory (discussed in the software section), we ran into problems when attempting to integrate the SD card with the rest of our subsystems. After connecting the appropriate pins and programming our device, we noticed that none of the subsystems were functioning (neither the SD card nor the lcd screen, sensors, audio output, etc.). After running numerous tests, we realized that the problem was not software related, but occurred solely due to the connection between our main board and the external SD card board. When the external board was connected, the voltage regulator became excessively hot, and so we determined that the SD card was drawing far too much current. With the SD card disconnected the rest of our subsystems worked fine.

Upon closer inspection, we noticed that a number of the SD card pins appeared to have melted. We did not have time to write all of our sound data onto a new SD card and so decided to simply store the arrays in the microcontroller's internal flash memory. We believe that this error is a result of making an incorrect connection to the SD card with the power on, causing the SD card to fry and a short circuit to form which, in turn, drew excessive current from our main board. However, we are including the information in our final documentation for any future groups that may wish to incorporate an SD card into their projects. We are confident that our functional subsystem could be integrated into a complete project.

### 2.3.8.1 SD CARD HARDWARE

Initially, we integrated the SD card adapter with our main board that we designed. However, since we did not use our designed board for the final demonstration, we decided to use the external SD card board that we had been using for testing. Both this board and our designed board (shown in the appendices) connected the SD card to SPI interface number 4, so all of the connections will be identical. The Eagle schematic for the external SD card board is shown below.

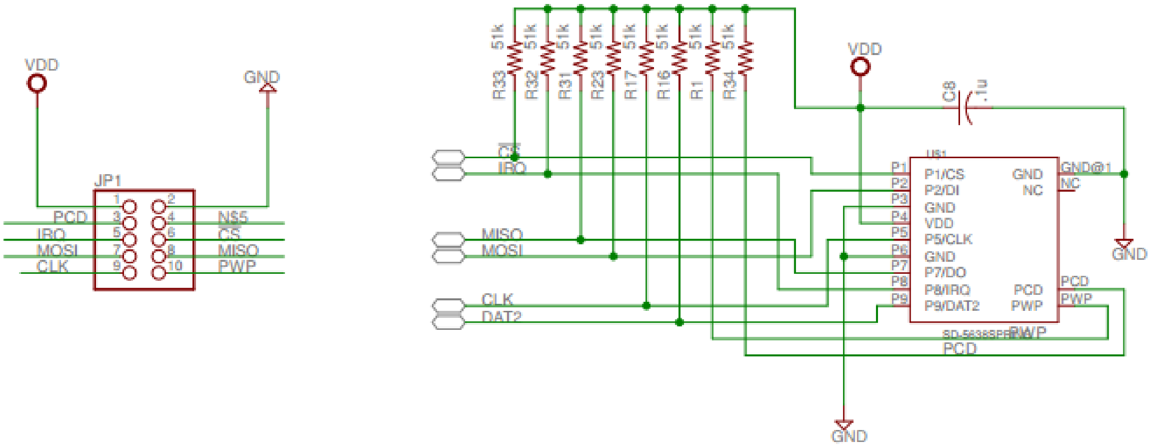
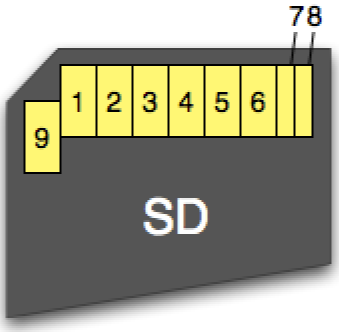


Figure 2.3.8.1.1: SD Card Schematic

The SD card adapter (on the right hand side of the schematic) requires 3.3 V power (supplied by the main board) and ground connections at the appropriately labeled ports, as shown. Each of the other pins may be dynamically changing between Vdd and ground, so pull up resistors are included between the pins and supply voltage to help deflect excess switching current away from the microcontroller and SD card pins. Furthermore, a decoupling capacitor is placed between Vdd and ground to smooth out any high frequency fluctuations on the power supply. The remaining SD adapter pins are connected to a 2x5 pin molex connector that interfaces with the main board via jumper wires.

A standard pinout of the SD card that will be inserted into the adapter is shown below, with pin 1 (CS) corresponding to P1 on the adapter.



| Pin | SD | SPI |
|---|---|---|
| 1 | CD/DAT3 | CS |
| 2 | CMD | DI |
| 3 | VSS1 | VSS1 |
| 4 | VDD | VDD |
| 5 | CLK | SCLK |
| 6 | VSS2 | VSS2 |
| 7 | DAT0 | DO |
| 8 | DAT1 | X |
| 9 | DAT2 | X |

Figure 2.3.8.1.2: SD Card Pin Connections

As mentioned earlier, we decided to operate the SD card in SPI mode, rather than I2C or SD mode, because of its ease of use, flexibility, and simple interfacing with the microcontroller. For this reason, pins P8 and above on the external board were not used in this project.

Pin 1, or the Chip Select (CS) pin, is used to activate or "select" the SD card device and tell it look at its data in and clock pins for more information. This pin must be set to a low level (ground) whenever commands are sent to the device or whenever data is being read or written. Though the microcontroller has certain SPI pins dedicated for this specific purpose (called SS or Slave Select pins) we decided to use B12, a simple digital I/O pin, instead. This increases the flexibility of our system as we can select or deselect the SD card at any time, and does not add any significant complexity.

Pin 5 is the SD clock pin and, since the microcontroller is in master mode and the SD card is the slave, it is generated by the microcontroller. It is used to coordinate the reading of input data and the timing of output data by the SD card. Since we used the SPI4 interface for the SD card, the clock pin was connected to pin B14 on the main board. The SD card, however, expects the clock to be configured in a particular way, which must be set in the SPI configuration register in software. We ensured that the idle clock level was set low and data out transitions occurred on falling clock edges (since the sd card reads data on the rising clock edge). The clock frequency must also be set depending on the operation that is being performed. This is discussed in detail in the SD Software section.

Pin 2 is the SD Card Data In pin (or Master Out Slave In (MOSI) pin) and it is where incoming data is read by the device. This pin is connected to pin F5 on the microcontroller on the main board via the molex connector and jumper wires. The last pin used by the SPI module (excluding power and ground pins) is pin 7, the Data Out (or Master In Slave Out(MISO)) pin. Here, the SD card outputs data to the corresponding microcontroller pin, F4. The SPI module in the microcontroller stores the incoming data in a LIFO register that can be read from in software. This same register is written to in software in order to send data to the SD card.

### 2.3.8.2 SD CARD SOFTWARE

Using the C programming language and the MPLAB IDE with the xc32 compiler, we developed a set of functions for interfacing the SD card with the PIC32 microcontroller. Both low level and high level functions were written for reading/writing individual bytes, initializing the SPI module, initializing the SD card into SPI mode, sending commands, reading and writing multiple data blocks, as well as finding the memory address and block length of the desired sound.

The only functions the programmer needs to implement in the main function are get_mblocks, put_mblocks, and find_address. These high level functions are used to retrieve data, write data, and determine the memory address of the desired sound. These high level functions call low level functions such as sd_command and xchg_spi that are used to send SD commands and send and receive individual data bytes. Lastly there are two initialization functions that initialize the SPI module on the microcontroller for compatibility with the SD card as well as initialize the SD card in SPI mode. These initialization functions must be called first, before any high level functions are called. Once the proper initializations have been made, the get_mblocks, put_mblocks, and find_address functions can be called as many times as necessary. Descriptions for the various functions are given below, and the code is included in the appendices.

#### 2.3.8.2.1 INITIALIZATION FUNCTIONS

void spi_init()

This is the first function that must be called and initializes the microcontrollers SPI4 interface for SD card compatibility. It consists primarily of setting configuration bits in the appropriate registers. These include disabling JTAG mode (a default setting for one of the SPI4 pins), configuring the SPI4 pins for digital I/O, setting the SPI clock to an appropriate value for initialization (for our SDHC card, this was 400 kHz), and making sure the clock polarity and idle clock level were set appropriately. We also configured the module to send 8 bits data bytes rather than 16 bit or 32 bits, since our sound data was in 8 bit format.

void sd_init()

This function initializes the SD card for SPI communications. It begins by sending a number of dummy clock pulses with the chip select deactivated in order to clear the spi registers, then a number of dummy clock pulses with chip select activated in order to ready the sd card to receive data. It then sends the following SD commands in sequence before setting the clock rate to 20 MHz for faster read/write operations: CMD0, CMD8, ACMD41 (until response = 0x00). CMD0 resets the SD card into idle mode, CMD8 ensures the SD card can operate in the required voltage range and obtains data about the specific SD card model (SDHC, etc.), and ACMD41 is sent repeatedly until the SD card is responds that it is in SPI mode.

#### 2.3.8.2.2 LOW LEVEL FUNCTIONS

unsigned char xchg_spi(unsigned char)

This function is used for both reading and writing an individual byte to the SD card. Passing a byte into the function (usually given in hex format) writes that byte to the SPI4BUF register. This is LIFO buffer that simultaneously clocks data out to the F5 pin (SD card data in) while reading in new data from the F4 pin (SD card data out). After a byte is written to the register, the function waits for a flag to be set, indicating that the transfer is complete. It then reads the incoming data from the buffer and returns it. In order to read data, a dummy value of 0xFF is typically passed into the function.

sd_command(unsigned char[], unsigned char[])
This function utilizes xchg_spi to write commands to the SD card and obtain responses. It first sends a dummy byte, followed by the 6 byte long command. It then reads the response from the SD card which can either be one or five bytes long depending on the type of command that was sent.

#### 2.3.8.2.3 HIGH LEVEL FUNCTIONS

void put_mblocks(unsigned char[], int, unsigned char[])
This function takes as input the array to be written, the number of blocks (SDHC exclusively supports 512 byte data blocks) that will be written, and the starting address in memory that the array will be written to. CMD25 is first sent using sd_command to write multiple data blocks. After a dummy byte, nested for loops are used to send the desired number of 512 byte blocks. each data block must begin with a start token (in this case 0xFC) and must end with two CRC bytes (which can be anything since CRC is not used in SPI mode). After each data block is sent, the microcontroller must wait for a response indicating that the data has been accepted and successfully written to memory. After all of the blocks are sent, the microcontroller must send a stop tansfer token (0xFD) indicating that the transfer is complete, and then must wait until a response from the SD card indicates that the data has been received successfully.

void get_mblocks(unsigned char[], int, unsigned char[])
This function is used to read multiple blocks from a certain location in memory into an array. It begins by sending CMD18, then uses nested for loops to read in each 512 byte data block. For each data block, the microcontroller waits for the SD card to send the transfer token (0xFE) indicating that it is ready to output data. Next, 512 byte data is read into the array and the two crc bytes are discarded. In order to terminate the read operation, CMD12 is sent and the microcontroller waits until the SD card is no longer busy.

int find_address(unsigned char[], int, int)
This function takes a pointer to the start_address array to be updated and the note and instrument information as input. Since, for our project, notes were written by software into user defined locations in memory, we used nested case statements to find the memory location of the note given the note pitch (A, B, C, etc.) and the instrument type (piano, clarinet, etc.). The start_address array is updated accordingly and the function returns the number of blocks that note requires.

#### 2.3.8.3 SD CARD TESTING

In order to test that the SD card and microcontroller were communicating properly over the SPI interface, the USBee logic analyzer was used extensively. Placing leads on the chip select (B12), clock (B14), MISO (F4), and MOSI (F5) pins, we were able to see the data and corresponding clock pulses being sent back and forth. After verifying the correct clock timing, commands, and responses, we began writing real audio data to the SD card. We did this by hardcoding audio data from each individual note into an array in software, then writing that note to a known location in memory using put_mblocks. We did this for each note. In order to verify that the notes were written correctly, we used the get_mblocks function to read a particular note into an array from one of the defined memory locations. The first element in the array was then output to port E so that the LEDs should display the proper byte. By this process we verified that each note had been written properly and could be read into an array for audio generation. This method of writing each note into SD memory manually is not plausible for users. A future enhancement of our project would include a primitive file system that allows users to drop .wav files onto the SD card from their PC. The LCD screen would then automatically update, allowing users to select and play their newly added notes. More about this enhancement is included in Section 5: To Market Design Changes.

### 2.3.9 INTERFACES BETWEEN DIFFERENT SUBSYSTEMS

Once we established the functionality of each of our working subsystems, we needed to integrate each of them into a cohesive unit. Many of the interfaces between subsystems was discussed in the subsystem sections, but a brief overview of each interface is given below for clarity.

External SD card to main board: SPI interface on 4 wires
User Interface to main board: 3 SPI wires for the LCD and 3 I/O wires for the buttons
Sensors to ADC: Analog voltage to analog pins on the PIC32 via wires
PWM sound to audio output: wire
Power Supply to subsystems: wire

These are all of the physical interfaces between the laser harp subsystems, but interfaces also needed to be created in software to integrate the PWM, ADC, SD, and LCD modules. Beginning with the main function code for the LCD display, which defines the user interface flow, the other software subsystems were integrated fairly easily. The ADC, PWM, and SD modules are initialized at the beginning of the main function. We first needed to make sure no settings overlapped and no configurations were being overwritten.

The PWM module enables interrupts to occur whenever a predefined timer counts up to a certain value. At that point, an interrupt service routine (ISR) is called which sets the new duty cycle of the output PWM signal. Since the LCD module continuously checks user input to see whether an instrument change has occurred, the ADC and SD module are simply integrated with the PWM module in the ISR. Each time the ISR is called, the ADC checks the value at the analog input pins. If it is within the threshold for a sound to be played, the duty cycle will be updated with the next byte value in the audio data array. If a new note is detected, the audio data array will be filled with data from the SD card corresponding to the selected instrument and the particular sensor that is being covered. In this way, the main file is kept relatively simple and the programmer only needs to worry about using a handful of functions from each software subsystem. In our demonstration prototype, we did not use the SD card. Instead, we hardcoded the audio data arrays for each of the 24 notes (three instruments, eight notes each) into software and stored them in the microcontroller's internal flash memory. The ISR then retrieved the selected note from flash memory rather than the SD card.

# 3 SYSTEM INTEGRATION TESTING

## 3.1 INTEGRATION TESTING

After Integrating all of our subsystems, we needed to test the functionality of our laser harp to ensure proper interfacing. Initially, we had planned to power all of the subsystems directly from the board. However, after plugging in the power jack and making the proper wire connections between the SD card, LCD screen, lasers, and sensors, we noted that the final system was not operating as expected. After programming the microcontroller with our completed code, we noted that neither the LCD screen nor the sound output were functioning at all. For debugging and testing purposes, we programmed the LEDs to flash when the corresponding sensor was covered. This did not work either.

In an effort to determine the problem, we disconnected the power brick and instead powered the board with a 3.3V DC voltage supply. The supply displayed that the amount of current being drawn exceeded 1.5 Amps. Furthermore, the voltage regulator was very hot to the touch. These observations convinced us that the subsystems were attempting to draw more power than the board was able to source. After disconnecting the lasers and sensors from the main board and powering them with 3V D battery packs, we still found that the too much power was being drawn from the board. This was unexpected, since the board worked well when interfacing solely with the LCD screen, and the SD card draws very low current. This is when we realized that the SD card most likely contained a short circuit, as discussed in section 2.3.9. The physical package of the card we were using appeared to be melted.

After disconnecting the SD card, our assembled system functioned properly. When the sensors were covered, the LEDs lit up and the correct note could be heard from the speakers. The lasers were visible in a dark, foggy environment. The user could navigate through each of the menu screens and options on the LCD display using the push buttons and successfully change the instrument being played. The amplifier increased volume to a substantial level without loss of sound quality. All the while the voltage regulator on the main board remained cool, indicating the board was not being required to source too much current. We still decided to power the sensors and lasers with the battery packs to reduce the total strain on the board.

Lastly, we attempted to test our working prototype under conditions that could arise if an inexperienced user were to use our product. For example, we attempted to change the display screen while playing a note. We pushed buttons on the user interface rapidly and without order. We also covered multiple sensors at the same time. In each of these scenarios, no errors occurred and the laser harp continued functioning as expected. In the course of our testing, we did notice a few glitches that might be improved upon in future products. These improvements will be discussed more extensively in section 5, but one particular improvement would include cleaning up the output sound when the user’s hand is very close to the sensors or at a distance right at the sensor threshold value.

### 3.2 MEETING DESIGN REQUIREMENTS

The tests we ran on our laser harp proved that the finished product met all of our design requirements. The user interface allowed the user to intuitively select from a number of different instruments. When a sensor was covered below the threshold distance, the correct instrument and pitch were output through the speakers. The sound was not only recognizable as a piano, clarinet, or animal sound, it was also pleasing to listen to. Laser harp users will also have the ability to play songs with rapidly changing notes, since the delay time to load a new note is negligible. Additionally, the lasers were visible given the right environmental conditions (dark and with a smoke machine) allowing users to see where to place their hands. Users also have the ability to tune the volume of the output sound via a knob on the amplifier, allowing the laser harp to be played in both public and private settings.

The laser harp packaging was also tested and found to be portable and reasonably durable. One person could easily carry the instrument around Stinson-Remick, up and down flights of stairs, with minimal effort and without loss of functionality. The external frame was composed of solid wood fastened together with screws and right angle fasteners. The user interface was secured on a customized 3D printout and the sensors were screwed into an 8 sided wooden protrusion that allowed the sensors to “fan out” from the frame. Thus the exterior of the laser harp was both durable and portable. The internal circuitry, however, was not held in place and so the user needed to be careful not to jostle the instrument too much. A future enhancement would include securing all of the internal boards, batteries, and amplifiers for improved durability.

Lastly, our prototype was powered using batteries and a power brick so it can be used anywhere with an outlet. The current strain on the board was minimized and verified based on the low temperature of the voltage regulator. This will improve the product’s reliability and lifetime. The swing door on the back of the laser harp also allows users to easily replace the batteries, further increasing the lifetime of the product.

## 4 USERS MANUAL

### 4.1 GETTING STARTED

The first thing the user must do is plug in the laser harp power brick into a wall outlet. This will provide power to most of the electronics housed inside the casing. The user must also make sure that the four D batteries and the 9V battery inside the amplifier are sufficiently charged. If any of the lasers, sensors, or sound do not work, this may be the issue. Next, the user should open the back of the laser harp and ensure that both the board and the amplifier are turned on. The board has a switch and when powered on, a green LED will indicate that the board is receiving power.

The lasers and sensors receive their power from the two battery packs. Ensure that all wires are connected before proceeding. Also, when done using the laser harp, remember to turn off the amplifier and disconnect the battery packs from the laser and sensors to stop power from being drawn unnecessarily from the batteries.

Once everything is powered up, test that the laser harp works by placing your hand in front of one of the sensors. If there is no audio or the audio is faint, proceed to turn up the volume on the side of the amplifier until you reach a comfortable level. Once everything is as desired, close the back of the laser harp and place it on the floor or on a table for playing.

### 4.2 CHANGING INSTRUMENTS

The laser harp comes with three built in instruments: Piano, Clarinet, and Animal Noises. To change the instrument, the user should use the user interface on top of the laser harp. The LCD screen menus will provide guidance in selecting an instrument. After selecting the desired instrument, proceed to test the laser harp to make sure that the decision was registered.

In the event of an error or trouble, proceed to turn off the board on the inside of the laser harp and turn it back on. Most times this will solve any issues. if the issue is still not resolved, turn off the board and unplug the power cord for at least 10 minutes. This will allow the components to cool down and function properly next time.

At present, there is no easy way for users to add their own instruments. Future enhancements will allow users to place wave files on the SD card, which automatically updates the user interface with additional instrument options. However, the current method requires access to the system software, a programmer, and IDE. The wave files must first be converted to 8 bit C BYTE arrays with the first 44 bytes (used for file identification) removed. These arrays can then be hardcoded into the software and stored in the microcontroller's internal flash memory.

## 5 TO-MARKET DESIGN CHANGES

By building our prototype laser harp we learned all about how we could have done things even better. If we were going to redesign our project and sell it commercially we would certainly make some changes.

The first step in making our prototype market-ready would be to fix the parts of our prototype that were unsuccessful. This process would begin with making our board work properly. To do so, we would likely take some time to reanalyze our board design to make sure it did exactly what we wanted it to. We probably could have laid out the board better, eliminating all the empty space in the middle section. We would definitely eliminate the board's ability to power both the sensors and the lasers in addition to the LCD screen and the SD card. Our prototype showed us that asking the board to power all of our peripheral hardware was far too much of a current strain on the voltage regulator. Furthermore, the voltage regulator that was intended to provide -3.3V was not wired correctly. The regulator's specification sheet provides resources for building a circuit to do this, so it would be an easy fix on future boards.

We did not use our custom board because we did not understand why it was not functioning. In hindsight, we believe this was because we were configuring the microcontroller to run on an external crystal clock source. The kit boards include an external crystal, but our board did not. To fix this problem, we could simply have configured the microcontroller to run on an internal oscillator.

We would certainly also focus on fixing the SD card to enable the harp to have more than three instruments. We would try to make the system user friendly so that users could download new audio files and put them on the SD card so that they could essentially develop their own instrument sounds. We would also have to program the LCD screen user interface to work with these changes and display the user-added instruments. In order to do this, we would need to establish a primitive file system that searches through the SD card memory and finds the correct folder and file names to be read.

One of the functions we had intended to include on our prototype was the ability to change volume depending on the height of the user's hand. This functionality depended on the amplifier we had picked out for our board, which had a 4-bit programmable gain setting as well as a low pass filter. For a commercial version of the laser harp, we would make sure to have this amplifier working on the board so we could incorporate the volume changing functionality. The low pass filter would also eliminate the slight presence of high-frequency noise due to the PWM process.

One of the more disappointing aspects of our prototype is the lasers. Since we were on a strict budget, we could not spend the money on nice lasers and instead simply tore apart cheap laser pointers to use their laser diodes and circuitry. We would definitely obtain more reliable lasers so that all eight beams would be visible in the right conditions (darkness with fog). The unreliability of our lasers forced us to use more fog than we would have liked to see the beams; this also interfered with our infrared sensors. Hopefully by having better lasers and less fog this problem would be eliminated, but if not we could change our software to change the sensing condition to cause the system to be less sensitive and not be triggered by the fog.

Another concern was the output of poor quality sound when the user's hand was either too close to the sensors or right at the threshold of the sensing distance. Here, the sensors switched back and forth between being "on" and being "off." This prevented the sound from outputting a continuous signal and degraded the quality. In order to fix this problem, a function could be incorporated in the software to effectively "debounce" the sensor. It could wait for a specified period of time, after which the note is played only if the sensor reading is within threshold.

Lastly, our prototype showed us that our box frame was far larger than necessary. We would definitely try to make our final product more compact and elegant. The largest necessary component is the curved holder for the lasers and sensors, so the box it is mounted to could be greatly downsized if the user interface were placed in front of the curved piece. Furthermore, the circuitry housed within the laser harp package should be secured to improve portability. This can be achieved by fastening the main board to a mount with screws, binding all loose wires together, and gluing the battery packs and amplifier to the package.

# 6 CONCLUSIONS

Building our laser harp was an extremely rewarding process and we were very proud of our finished product. We were able to make use of content from throughout our college curriculum and produce a polished device. The senior design project allowed us to experience the ups and downs of the design process firsthand and our laser harp serves as a successful prototype for a futuristic and stunning electronic musical instrument.

## 7 REFERENCES

SD Card Documentation

https://www.sdcard.org/downloads/pls/simplified_specs/part1_410.pdf

SD Card Reference

http://elm-chan.org/docs/mmc/mmc_e.html

Newhaven LCD Display Documentation

http://www.newhavendisplay.com/specs/NHD-0420D3Z-FL-GBW-V3.pdf

LED Push Button Documentation:

http://components.omron.com/components/web/pdflib.nsf/0/398A72F5A701DE668625738A006A7E33/$file/B3W-9_1110.pdf

PIC32 Documentation:

http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en545655#documentation

PIC32 PWM Reference

http://umassamherstm5.org/tech-tutorials/pic32-tutorials/ubw32-tutorials

Sound File Source

http://theremin.music.uiowa.edu/MISguitar.html

LTC1564 Amplifier/Low Pass Filter documentation:

http://cds.linear.com/docs/en/datasheet/1564fa.pdf

Flash Memory Documentation (we did not use this)

http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en548647#pricindAndSamples

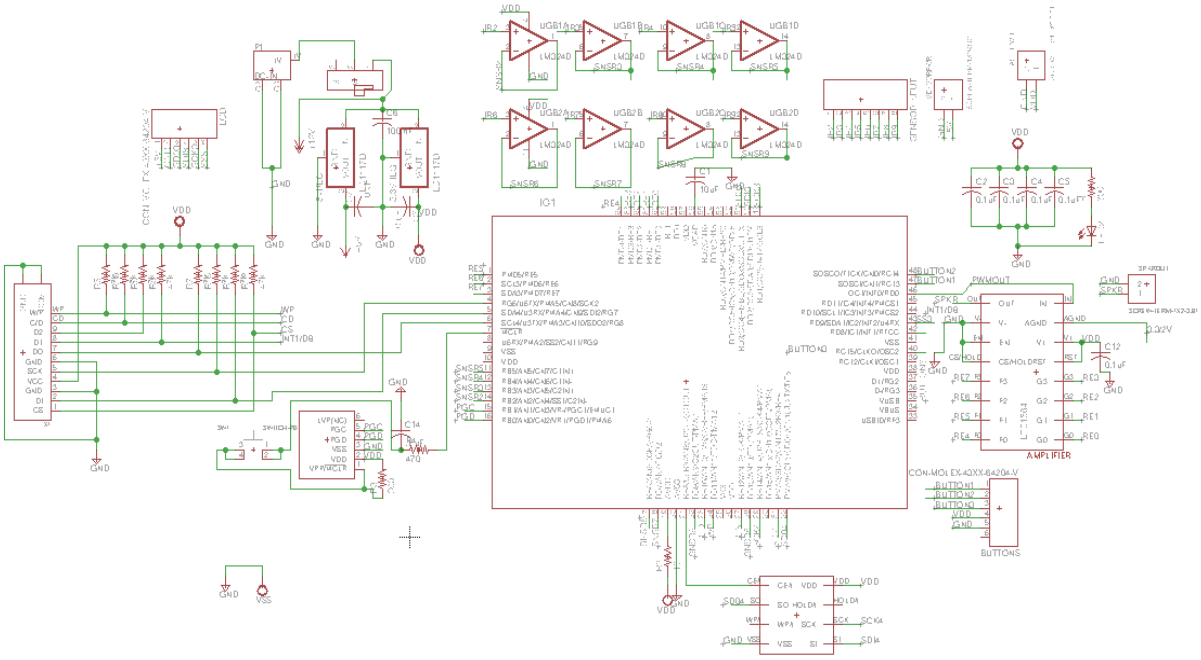
PIC32 ADC Reference

http://umassamherstm5.org/tech-tutorials/pic32-tutorials/pic32mx220-tutorials/adc

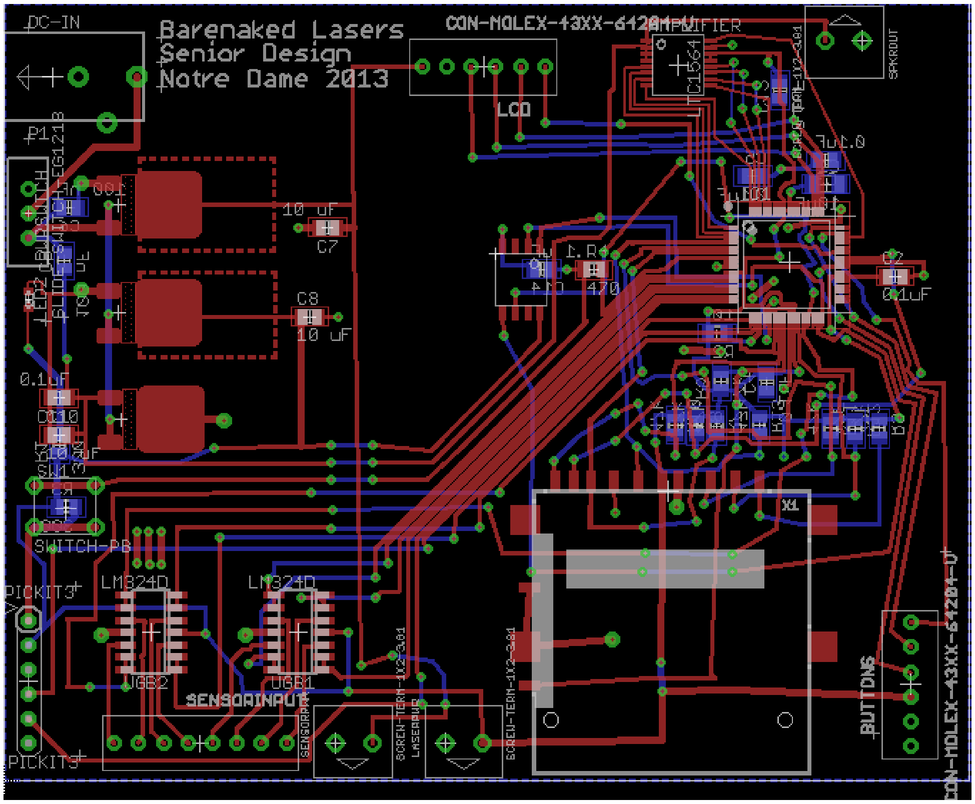
# 8 APPENDICES

## 8.1 HARDWARE SCHEMATICS

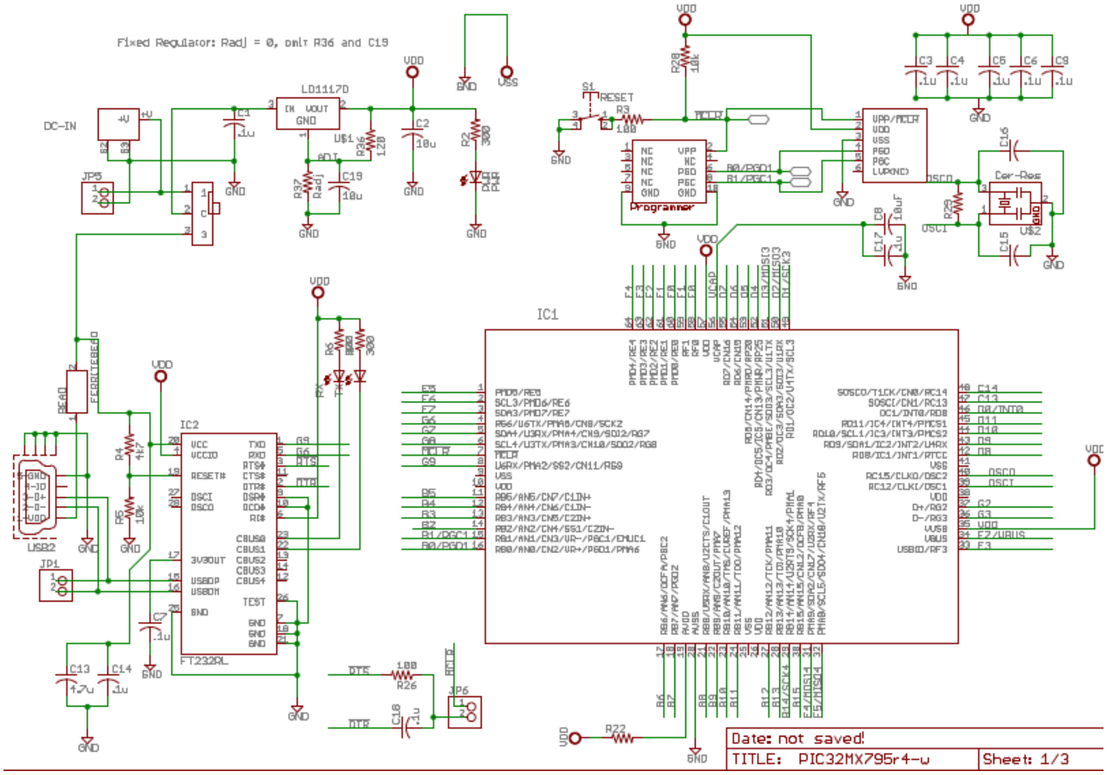
### 8.1.1 Custom Board Schematic:

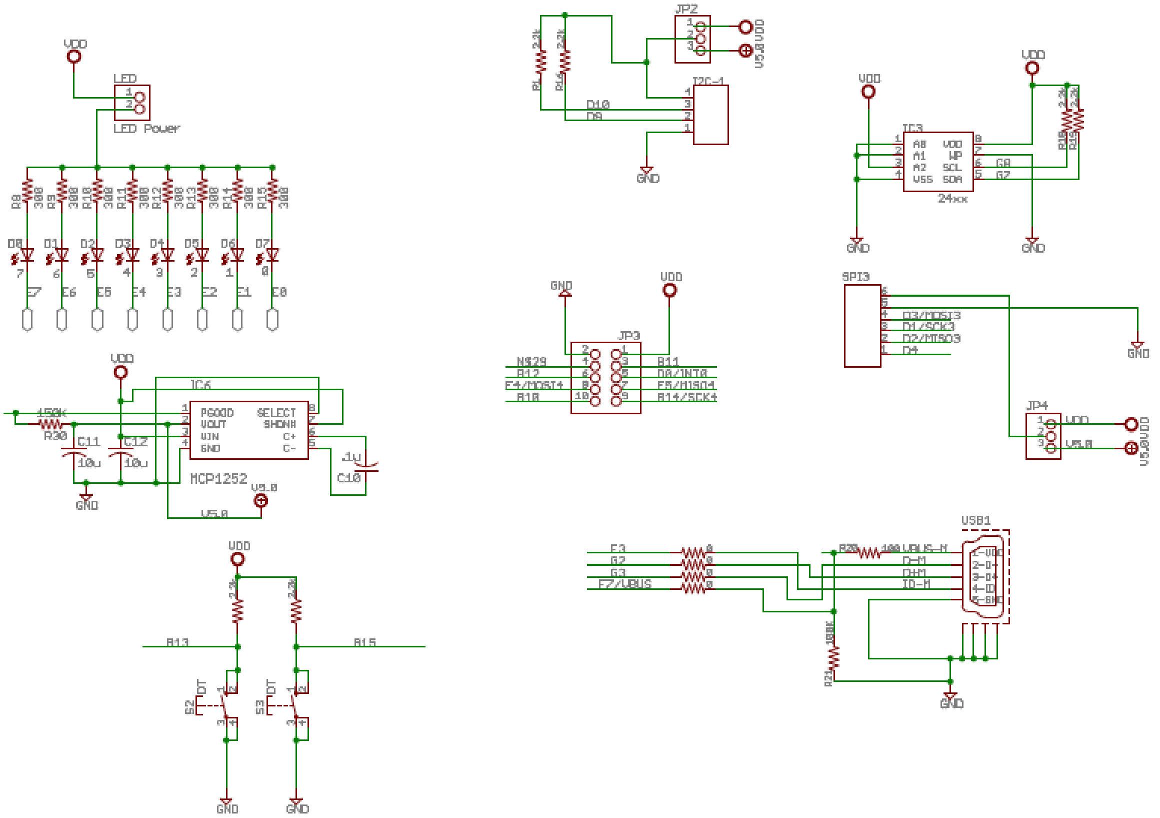


### 8.1.2 Custom Board Layout:



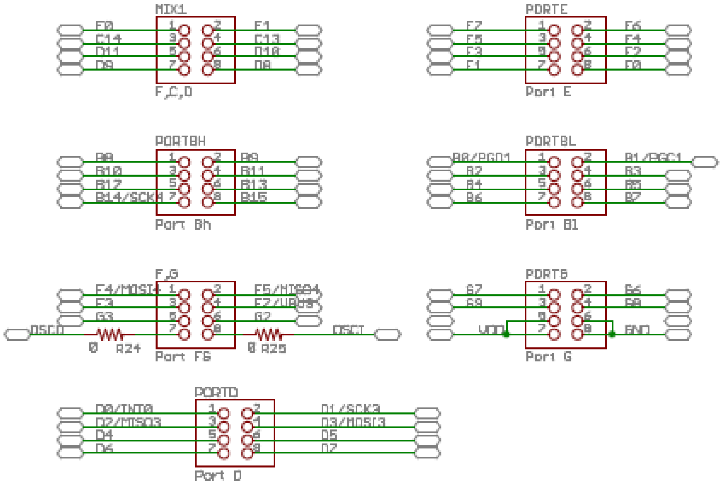
## 8.1.3 Kit Board Schematic:



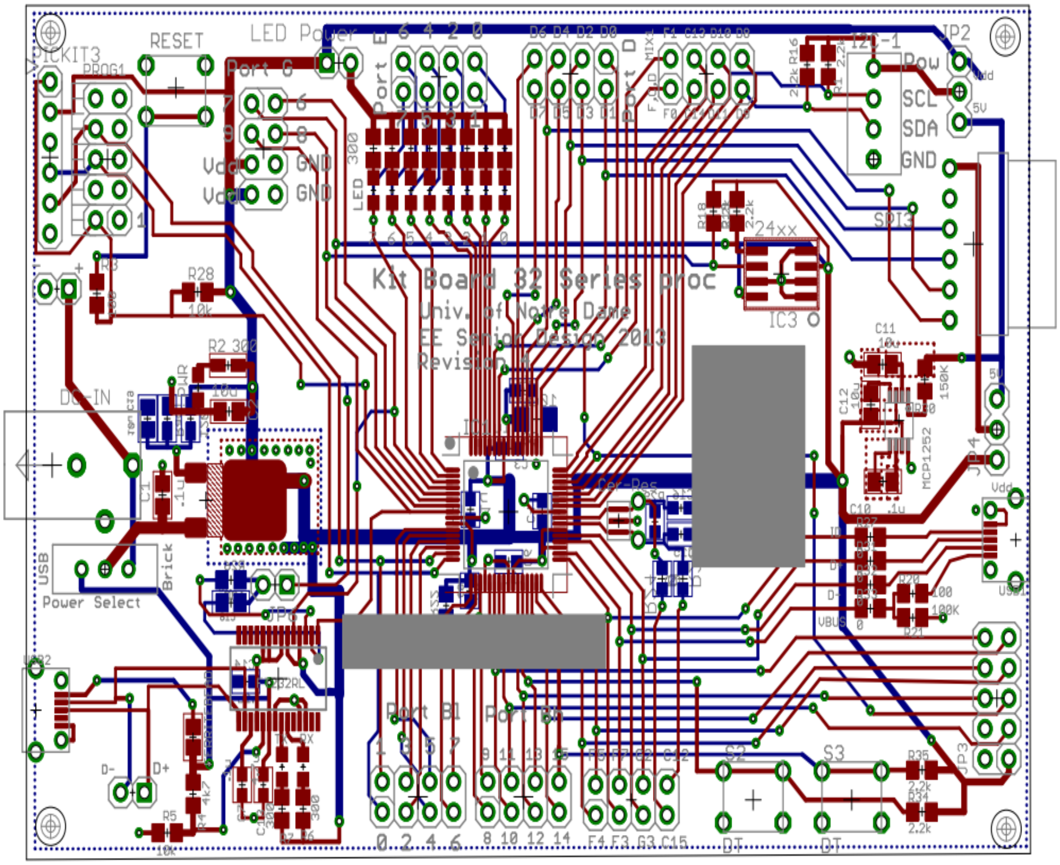


Date: not saved!

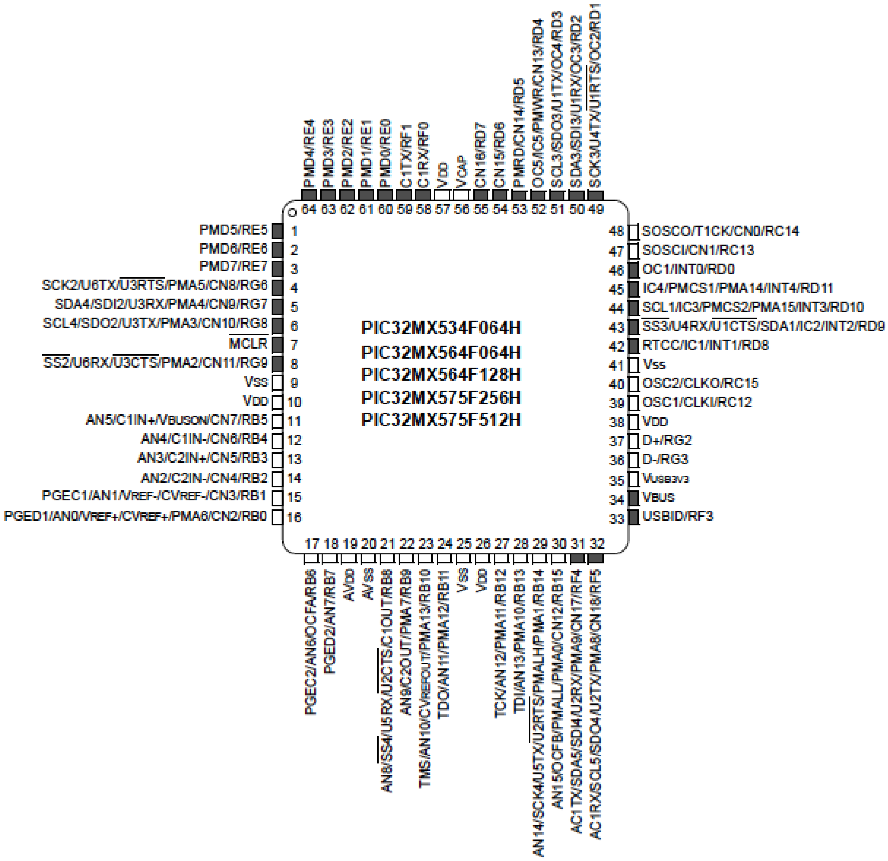
TITLE: PIC32MX795r4-w Sheet: 2/3



### 8.1.4 Kit Board Layout



### 8.1.5 PIC32MX795F512H Schematic:



## 8.2 SOFTWARE LISTINGS

### **8.2.1 main file:**

```
/*
 * File:   newmain.c
 * Author: Chris
 *
 * Created on April 29, 2013, 7:00 PM
 */
//#pragma config FSOSCEN = ON, FNOSC = FRCPLL //using internal FRC 8MHz
//#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_20, FPLLODIV=DIV_1 //SYS
CLK = 80MHz
//#pragma config FPBDIV=DIV_4, FWDTEN=OFF, CP=OFF, BWP=OFF

#pragma config FNOSC = PRIPLL // Oscillator selection
#pragma config POSCMOD = HS // Primary oscillator mode
#pragma config FPLLIDIV = DIV_5 // PLL input divider (20 -> 4)
#pragma config FPLLMUL = MUL_20 // PLL multiplier  ( 4x20 = 80)
#pragma config FPLLODIV = DIV_1 // PLL output divider
#pragma config FPBDIV = DIV_8 // Peripheral bus clock divider 20 mhz
#pragma config FSOSCEN = OFF // Secondary oscillator enable
// Clock control settings

#pragma config IESO = OFF // Internal/external clock switchover
#pragma config FCKSM = CSDCMD // Clock switching (CSx)/Clock monitor (CMx)
#pragma config OSCIOFNC = OFF

#pragma config UPLLEN = ON // USB PLL enable
#pragma config UPLLIDIV = DIV_2 // USB PLL input divider
#pragma config FVBUSONIO = OFF // VBUS pin control
#pragma config FUSBIDIO = OFF // USBID pin control
// Other Peripheral Device settings

#pragma config FWDTEN = OFF // Watchdog timer enable
#pragma config WDTPS = PS1024 // Watchdog timer post-scaler
#pragma config FSRSSEL = PRIORITY_7 // SRS interrupt priority

#pragma config      ICESEL      = ICS_PGx1


#include <stdio.h>
```

```
#include <stdlib.h>
#include <xc.h>
#include <string.h>
#include <plib.h>
#include <p32xxxx.h>
#include "delays.h"
#include "lcd.h"
#include "sd.h"
#include <sys/attribs.h>
#include "adc.h"
#include "pwm.h"
#include "notes.h"


/*
 *
 */
BYTE hex;
BYTE __attribute__ ((space(data))) recv_buffer[0x9000];



int z = 1;
int previousSensor = 8;
int previousInstrument = Piano;
int instrument = Piano;

int main(int argc, char** argv) {

    SYSTEMConfigPerformance(80000000L);
    IEC1CLR = 0xFFFFFFFF;

    init_adc();
    init_pwm();
    init_lcd_spi();

    int screenChange = 1;
    int currentCursor = 1;
    int nextCursor = 1;
    int currentScreen = MainMenu;
    send_command(blinking_cursor_on, 0x00);
send_command(clear_lcd, 0x00);

while (1) {

    nextCursor = currentCursor;

    if (currentScreen == MainMenu) {
        if (screenChange == 1) {
            display_mainmenu();
            screenChange = 0;
        }
        currentCursor = button_debounce(currentCursor);

        if (currentCursor != nextCursor) {

            if (currentCursor < 1) {
                send_command(set_cursor, 0x54);
                currentCursor = 3;
            } else if (currentCursor > 3) {
                send_command(set_cursor, 0x40);
                currentCursor = 1;
            } else {
                switch (currentCursor) {
                    case 1:
                    {
                        send_command(set_cursor, 0x40);
                        break;
                    }
                    case 2:
                    {
                        send_command(set_cursor, 0x14);
                        break;
                    }
                    case 3:
                    {
                        send_command(set_cursor, 0x54);
                        break;
                    }
                }
            }
        }
    //change screen...
    currentScreen = screen_select(currentScreen, currentCursor); //make sure this is
right!!!!!!
    if (currentScreen != MainMenu) {
      currentCursor = 1;
      screenChange = 1;
    }

  } else if (currentScreen == InstrumentSelect) {
    if (screenChange == 1) {
      display_instrmenu();
      screenChange = 0;
    }
    currentCursor = button_debounce(currentCursor);

    if (currentCursor != nextCursor) {
      if (currentCursor < 1) {
        send_command(set_cursor, 0x54);
        currentCursor = 3;
      } else if (currentCursor > 3) {
        send_command(set_cursor, 0x40);
        currentCursor = 1;
      } else {
        switch (currentCursor) {
          case 1:
          {
            send_command(set_cursor, 0x40);
            break;
          }
          case 2:
          {
            send_command(set_cursor, 0x14);
            break;
          }
          case 3:
          {
            send_command(set_cursor, 0x54);
            break;
          }
        }
      }
```

```
        }
        //change screen
        currentScreen = screen_select(currentScreen, currentCursor);
        if (currentScreen != InstrumentSelect) {
             //set instrument
            instrument = currentCursor;
            currentCursor = 1;
            screenChange = 1;
        }

    } else if (currentScreen == Options) {
        if (screenChange == 1) {
            display_options();
            screenChange = 0;
        }
        currentCursor = button_debounce(currentCursor);

        if (currentCursor != nextCursor) {
            if (currentCursor < 1) {
                send_command(set_cursor, 0x54);
                currentCursor = 3;
            } else if (currentCursor > 3) {
                send_command(set_cursor, 0x40);
                currentCursor = 1;
            } else {
                switch (currentCursor) {
                    case 1:
                    {
                        send_command(set_cursor, 0x40);
                        break;
                    }
                    case 2:
                    {
                        send_command(set_cursor, 0x14);
                        break;
                    }
                    case 3:
                    {
                        send_command(set_cursor, 0x54);
                        break;
                    }
                    }
                }
            }
            //change screen
            currentScreen = screen_select(currentScreen, currentCursor);
            if (currentScreen != Options) {
                currentCursor = 1;
                screenChange = 1;
            }

        } else if (currentScreen == About) {
            if (screenChange == 1) {
                display_about();
                screenChange = 0;
            }
            //change screen
            currentScreen = screen_select(currentScreen, currentCursor);
            if (currentScreen != About) {
                currentCursor = 1;
                screenChange = 1;
            }
        }
    }

    return (EXIT_SUCCESS);
}

void __ISR(_TIMER_2_VECTOR, ipl7) T2_IntHandler (void)
{

    int sensorData[8];
    while( ! IFS1bits.AD1IF); //wait until buffers contain new samples
        AD1CON1bits.ASAM = 0;     //stop automatic sampling (shut down ADC basically)

        sensorData[0] = ADC1BUF0;
        sensorData[1] = ADC1BUF1;
        sensorData[2] = ADC1BUF2;
        sensorData[3] = ADC1BUF3;
        sensorData[4] = ADC1BUF4;
        sensorData[5] = ADC1BUF5;
        sensorData[6] = ADC1BUF6;
```

```
       sensorData[7] = ADC1BUF7;
       IFS1bits.AD1IF = 0;
       AD1CON1bits.ASAM = 1;  //restart ADC and sampling
   int sensor=0;
   int sensorNumber=0;
   int i=0;
   for(i=0;i<8;i++){
       if(sensorData[i]>sensor){
           sensor=sensorData[i];
           sensorNumber=i;
       }
   }
   if(sensorNumber != previousSensor){
       z=1;
   }

if(sensor>300){
  if (sensorNumber == 0 && instrument == 1){
     if  (z<sizeof(pianoLowC)){
           hex=pianoLowC[z];
           OC1RS = (hex+1)*((PR2+1)/256);
           z++;
           IFS0CLR = 0x0100;
    }
     else{
         z=1;
         IFS0CLR = 0x0100;
     }
   }
  else if  (sensorNumber == 1 && instrument == 1){
     if  (z<sizeof(pianoD)){
           hex=pianoD[z];
           OC1RS = (hex+1)*((PR2+1)/256);
           z++;
           IFS0CLR = 0x0100;
    }
     else{
         z=1;
         IFS0CLR = 0x0100;
  }
  }
```

```
else if  (sensorNumber == 2 && instrument == 1){
   if  (z<sizeof(pianoE)){
         hex=pianoE[z];
         OC1RS = (hex+1)*((PR2+1)/256);
         z++;
         IFS0CLR = 0x0100;
  }
   else{
       z=1;
       IFS0CLR = 0x0100;
}
}
else if  (sensorNumber == 3 && instrument == 1){
   if  (z<sizeof(pianoF)){
         hex=pianoF[z];
         OC1RS = (hex+1)*((PR2+1)/256);
         z++;
         IFS0CLR = 0x0100;
  }
   else{
       z=1;
       IFS0CLR = 0x0100;
}
}
else if  (sensorNumber == 4 && instrument == 1){
   if  (z<sizeof(pianoG)){
         hex=pianoG[z];
         OC1RS = (hex+1)*((PR2+1)/256);
         z++;
         IFS0CLR = 0x0100;
  }
   else{
       z=1;
       IFS0CLR = 0x0100;
}
}
else if  (sensorNumber == 5 && instrument == 1){
   if  (z<sizeof(pianoA)){
         hex=pianoA[z];
         OC1RS = (hex+1)*((PR2+1)/256);
         z++;
```

```
        IFS0CLR = 0x0100;
 }
  else{
      z=1;
      IFS0CLR = 0x0100;
}
}
else if  (sensorNumber == 6 && instrument == 1){
  if  (z<sizeof(pianoB)){
        hex=pianoB[z];
        OC1RS = (hex+1)*((PR2+1)/256);
        z++;
        IFS0CLR = 0x0100;
 }
  else{
      z=1;
      IFS0CLR = 0x0100;
}
}
else if  (sensorNumber == 7 && instrument == 1){
  if  (z<sizeof(pianoHighC)){
        hex=pianoHighC[z];
        OC1RS = (hex+1)*((PR2+1)/256);
        z++;
        IFS0CLR = 0x0100;
 }
  else{
      z=1;
      IFS0CLR = 0x0100;
}
}
else if  (sensorNumber == 0 && instrument == 2){
  if  (z<sizeof(clarinetLowC)){
        hex=clarinetLowC[z];
        OC1RS = (hex+1)*((PR2+1)/256);
        z++;
        IFS0CLR = 0x0100;
 }
else{
      z=1;
      IFS0CLR = 0x0100;
```

```
}
}
else if  (sensorNumber == 1 && instrument == 2){
  if  (z<sizeof(clarinetD)){
      hex=clarinetD[z];
      OC1RS = (hex+1)*((PR2+1)/256);
      z++;
      IFS0CLR = 0x0100;
 }
else{
     z=1;
     IFS0CLR = 0x0100;
}
}
else if  (sensorNumber == 2 && instrument == 2){
  if  (z<sizeof(clarinetE)){
      hex=clarinetE[z];
      OC1RS = (hex+1)*((PR2+1)/256);
      z++;
      IFS0CLR = 0x0100;
 }
else{
     z=1;
     IFS0CLR = 0x0100;
}
}
else if  (sensorNumber == 3 && instrument == 2){
  if  (z<sizeof(clarinetF)){
      hex=clarinetF[z];
      OC1RS = (hex+1)*((PR2+1)/256);
      z++;
      IFS0CLR = 0x0100;
 }
else{
     z=1;
     IFS0CLR = 0x0100;
}
}
else if  (sensorNumber == 4 && instrument == 2){
  if  (z<sizeof(clarinetG)){
      hex=clarinetG[z];
      OC1RS = (hex+1)*((PR2+1)/256);
      z++;
      IFS0CLR = 0x0100;
 }
else{
     z=1;
     IFS0CLR = 0x0100;
}
}
else if  (sensorNumber == 5 && instrument == 2){
  if  (z<sizeof(clarinetA)){
      hex=clarinetA[z];
      OC1RS = (hex+1)*((PR2+1)/256);
      z++;
      IFS0CLR = 0x0100;
 }
else{
     z=1;
     IFS0CLR = 0x0100;
}
}
else if  (sensorNumber == 6 && instrument == 2){
  if  (z<sizeof(clarinetB)){
      hex=clarinetB[z];
      OC1RS = (hex+1)*((PR2+1)/256);
      z++;
      IFS0CLR = 0x0100;
 }
else{
     z=1;
     IFS0CLR = 0x0100;
}
}
else if  (sensorNumber == 7 && instrument == 2){
  if  (z<sizeof(clarinetHighC)){
      hex=clarinetHighC[z];
      OC1RS = (hex+1)*((PR2+1)/256);
      z++;
      IFS0CLR = 0x0100;
 }
else{
     z=1;
     IFS0CLR = 0x0100;
}
}

else if  (ADC1BUF0 > 400 && instrument == 3){
  if  (z<sizeof(catSound)){
      hex=catSound[z];
      OC1RS = (hex+1)*((PR2+1)/256);
      z++;
      IFS0CLR = 0x0100;
 }
else{
     z=1;
     IFS0CLR = 0x0100;
}
}
else if  (ADC1BUF1 > 400 && instrument == 3){
  if  (z<sizeof(dogSound)){
      hex=dogSound[z];
      OC1RS = (hex+1)*((PR2+1)/256);
      z++;
      IFS0CLR = 0x0100;
 }
else{
     z=1;
     IFS0CLR = 0x0100;
}
}
else if  (ADC1BUF2 > 400 && instrument == 3){
  if  (z<sizeof(dolphinSound)){
      hex=dolphinSound[z];
      OC1RS = (hex+1)*((PR2+1)/256);
      z++;
      IFS0CLR = 0x0100;
 }
else{
     z=1;
     IFS0CLR = 0x0100;
}
}
```

```
else if  (ADC1BUF3 > 400 && instrument == 3){
  if  (z<sizeof(donkeySound)){
      hex=donkeySound[z];
      OC1RS = (hex+1)*((PR2+1)/256);
      z++;
      IFS0CLR = 0x0100;
 }
else{
     z=1;
     IFS0CLR = 0x0100;
}
}
else if  (ADC1BUF4 > 400 && instrument == 3){
  if  (z<sizeof(duckSound)){
      hex=duckSound[z];
      OC1RS = (hex+1)*((PR2+1)/256);
      z++;
      IFS0CLR = 0x0100;
 }
else{
     z=1;
     IFS0CLR = 0x0100;
}
}
else if  (ADC1BUF5 > 400 && instrument == 3){
  if  (z<sizeof(frogSound)){
      hex=frogSound[z];
      OC1RS = (hex+1)*((PR2+1)/256);
      z++;
      IFS0CLR = 0x0100;
 }
else{
     z=1;
     IFS0CLR = 0x0100;
}
}
else if  (ADC1BUF6 > 400 && instrument == 3){
  if  (z<sizeof(jaguarSound)){
      hex=jaguarSound[z];
      OC1RS = (hex+1)*((PR2+1)/256);
      z++;
```

```
          IFS0CLR = 0x0100;
    }
  else{
        z=1;
        IFS0CLR = 0x0100;
  }
  }
  else if  (ADC1BUF7 > 400 && instrument == 3){
     if  (z<sizeof(lionSound)){
          hex=lionSound[z];
          OC1RS = (hex+1)*((PR2+1)/256);
          z++;
          IFS0CLR = 0x0100;
    }
  else{
        z=1;
        IFS0CLR = 0x0100;
  }
  }
 }
  else{
      z=1;
      OC1RS = 0;
      IFS0CLR = 0x0100;
  }
   previousSensor = sensorNumber;

}
```

### 8.2.2 lcd.c file:

```
#include <stdio.h>
#include <stdlib.h>
#include <xc.h>
#include <string.h>
#include <plib.h>
#include <p32xxxx.h>
#include "delays.h"
#include "lcd.h"
#include "sd.h"

void init_lcd_spi()
```

```
{
    //DDPCONbits.JTAGEN = 0; //disable jtag mode
    TRISDbits.TRISD1 = 0; //SCK3 Output
    TRISDbits.TRISD2 = 1; //SDI3 Input
    TRISDbits.TRISD3 = 0; //SDO Output
    TRISDbits.TRISD6 = 0; //CS = Output

    TRISFbits.TRISF1 = 1;
    TRISFbits.TRISF0 = 1;
    TRISDbits.TRISD7 = 1;

    LATDbits.LATD6 = 1;

    //IEC1CLR = 0xFFFFFFFF;
    int rData = SPI3BUF; // clears the receive buffer
    //IPC6bits.SPI3IP = 0b11;
    //IPC6bits.SPI3IS = 0b01;
    //IEC0bits.SPI3TXIE = 1;//0x03800000; // Enable RX, TX and Error interrupts
    //IEC0bits.SPI3RXIE = 1;
    SPI3BRG = 400; // use FPB/50 clock frequency->100khz for init mode
    SPI3STATCLR = 0xFFFF; //0x40; // clear the Overflow
    SPI3CON = 0b00000000000000000000000001100000;

    SPI3CONbits.ON = 1;
    LATDbits.LATD6 = 0;
    //send_command(clear_lcd, 0x00);
}

unsigned char display_char(unsigned char dataOut)
{
    SPI3BUF = dataOut;
    while(!SPI3STATbits.SPIRBF);
    return SPI3BUF;
}

void display_string(unsigned char phrase[])
{
    int i;
    for(i=0; i<20; i++)
    {
        display_char(phrase[i]);
        delay_us(100);
    }
}

void send_command(unsigned char cmd, unsigned char param)
{
    display_char(0xFE);
    display_char(cmd);
    if(cmd == 0x45)
    {
        display_char(param);
    }
    if(cmd == clear_lcd)
    {
        delay_us(1500);
    }
    else
    {
        delay_us(100);
    }

}

int button_debounce(int currentCursor) {
    int flag = 0;
    while (PORTFbits.RF0 == 0) {
        if (flag == 0) {
            currentCursor++; //send_command(set_cursor, 0x14);
            flag = 1;
        }
    }
    flag = 0;
    while (PORTFbits.RF1 == 0) {
        if (flag == 0) {
            currentCursor--; //send_command(set_cursor, 0x40);
            flag = 1;
        }
    }
    return currentCursor;
}
```

```
int screen_select(int currentScreen, int currentCursor) {
   int flag = 0;
   int nextScreen = currentScreen;
   while (PORTDbits.RD7 == 0) {
      if(flag == 0)
      {
         nextScreen = currentCursor;
         flag = 1;
      }
   }

   if(currentScreen == MainMenu)
   {
      return nextScreen;
   }
   else if(currentScreen == InstrumentSelect)
   {
      if(flag == 0)
      {
         return InstrumentSelect;
      }
      else if(nextScreen != 8)
      {
         send_command(clear_lcd, 0x00);
         send_command(set_cursor, 0x40);
         unsigned char loading[] = {' ', ' ', ' ', ' ', ' ', 'L', 'o', 'a', 'd', 'i', 'n', 'g', '.', '.', '.', ' ', ' ', ' ', '
', ' '};
         display_string(loading);
         //call functions to read from sd card...
         return MainMenu;
      }
      else //if "back" selected
      {
         return MainMenu;
      }
   }
   else if(currentScreen == Options)
   {
      if(nextScreen == 3)
      {
         return MainMenu;
```

```
        }
        else
        {
            return Options;
        }
    }
    else if(currentScreen == About)
    {
        if(flag == 0)
        {
            return About;
        }
        else
        {
            return MainMenu;
        }
    }

    return nextScreen;
}

void display_mainmenu()
{
    unsigned char mainMenu0[] = {'L', 'a', 's', 'e', 'r', ' ', 'H', 'a', 'r', 'p', ' ', 'M', 'a', 'i', 'n', ' ',
'M', 'e', 'n', 'u'};
    unsigned char mainMenu2[] = {'2', ' ', 'O', 'p', 't', 'i', 'o', 'n', 's', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '
', ' '};
    unsigned char mainMenu1[] = {'1', ' ', 'S', 'e', 'l', 'e', 'c', 't', ' ', 'I', 'n', 's', 't', 'r', 'u', 'm', 'e',
'n', 't', ' '};
    unsigned char mainMenu3[] = {'3', ' ', 'A', 'b', 'o', 'u', 't', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '
', ' '};

    send_command(clear_lcd, 0x00);
    send_command(set_cursor, 0x00);
    display_string(mainMenu0);
    send_command(set_cursor, 0x40);
    display_string(mainMenu1);
    send_command(set_cursor, 0x14);
    display_string(mainMenu2);
    send_command(set_cursor, 0x54);
    display_string(mainMenu3);
    send_command(set_cursor, 0x40);
}

void display_instrmenu() {
    unsigned char instrMenu0[] = {'S', 'e', 'l', 'e', 'c', 't', ' ', 'I', 'n', 's', 't', 'r', 'u', 'm', 'e', 'n', 't',
' ', ' ', ' '};
    unsigned char instrMenu1[] = {'1', ' ', 'P', 'i', 'a', 'n', 'o', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
' '};
    unsigned char instrMenu2[] = {'2', ' ', 'C', 'l', 'a', 'r', 'i', 'n', 'e', 't', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '
', ' '};
    unsigned char instrMenu3[] = {'3', ' ', 'A', 'n', 'i', 'm', 'a', 'l', ' ', 'N', 'o', 'i', 's', 'e', 's', ' ', ' ', '
', ' ', ' '};
    send_command(clear_lcd, 0x00);
    send_command(set_cursor, 0x00);
    display_string(instrMenu0);
    send_command(set_cursor, 0x40);
    display_string(instrMenu1);
    send_command(set_cursor, 0x14);
    display_string(instrMenu2);
    send_command(set_cursor, 0x54);
    display_string(instrMenu3);
    send_command(set_cursor, 0x40);
}

void display_options()
{
    unsigned char optmenu1[] = {'1', ' ', 'B', 'r', 'i', 'g', 'h', 't', 'n', 'e', 's', 's', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
' '};
    unsigned char optmenu2[] = {'2', ' ', 'C', 'o', 'n', 't', 'r', 'a', 's', 't', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
' '};
    unsigned char optmenu3[] = {'3', ' ', 'B', 'a', 'c', 'k', ' ', 't', 'o', ' ', 'M', 'a', 'i', 'n', ' ', 'M', 'e',
'n', 'u', ' '};
    unsigned char optmenu0[] = {'O', 'p', 't', 'i', 'o', 'n', 's', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '
'};
    send_command(clear_lcd, 0x00);
    send_command(set_cursor, 0x00);
    display_string(optmenu0);
    send_command(set_cursor, 0x40);
    display_string(optmenu1);
    send_command(set_cursor, 0x14);
    display_string(optmenu2);
    send_command(set_cursor, 0x54);
    display_string(optmenu3);
    send_command(set_cursor, 0x40);
}

void display_about()
{
    unsigned char aboutmenu3[] = {'^', ' ', 'B', 'a', 'c', 'k', ' ', 't', 'o', ' ', 'M', 'a', 'i', 'n', ' ', 'M', 'e', 'n', 'u', ' '};
    unsigned char aboutmenu0[] = {'P', 'l', 'a', 'y', ' ', 'n', 'o', 't', 'e', ' ', 'b', 'y', ' ', 'c', 'o', 'v', 'e', 'r', '-', ' '};
    unsigned char aboutmenu1[] = {'i', 'n', 'g', ' ', 'l', 'a', 's', 'e', 'r', '.', ' ', 'D', 'i', 'f', 'f', 'e', 'r', 'e', 'n', 't'};
    unsigned char aboutmenu2[] = {'l', 'a', 's', 'e', 'r', 's', ' ', 'f', 'o', 'r', ' ', 'p', 'i', 't', 'c', 'h', '.', ' ', ' ', ' '};
    send_command(clear_lcd, 0x00);
    send_command(set_cursor, 0x00);
    display_string(aboutmenu0);
    send_command(set_cursor, 0x40);
    display_string(aboutmenu1);
    send_command(set_cursor, 0x14);
    display_string(aboutmenu2);
    send_command(set_cursor, 0x54);
    display_string(aboutmenu3);
    send_command(set_cursor, 0x54);
}
```

### 8.2.3 sd.c file:

```
#include "sd.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <xc.h>
#include <string.h>
#include <plib.h>
#include <p32xxxx.h>

unsigned char xchg_spi(unsigned char dataOut)
{
    SPI4BUF = dataOut;
    while(!SPI4STATbits.SPIRBF);
```

```
   return SPI4BUF;
}

void sd_command(unsigned char command [], unsigned char response[])
{
   int i, limit = 1;

   if (command[0] == 0x40 || command[0] == 0x58 || command[0] == 0x51){
     limit = 1;}
   if (command[0] == 0x48 || command[0] == 0x7A){
     limit = 5;}

   xchg_spi(0xFF);

   for (i=0; i<6; i++)
   {
      xchg_spi(command[i]);
   }
   if(command[0] == 0x4C)
   {
      response[0] = xchg_spi(0xFF);
   }
   do{
      response[0] = xchg_spi(0xFF);

   } while (response[0] == 0xFF);

   //fill response buffer
   if (response[0] == 0x04) // if illegal command
   {
      limit = 1;
   }

   for (i = 1; i < limit; i++) {
      response[i] = xchg_spi(0xFF);
   }

      return;
}

void spi_init()
{
   DDPCONbits.JTAGEN = 0; //disable jtag mode

   AD1PCFGbits.PCFG12 = 1;  //disable analog to digital
   AD1PCFGbits.PCFG14 = 1;


   TRISBbits.TRISB12 = 0; //CS is output
   TRISBbits.TRISB14 = 0; //SCK4 is output

   TRISFbits.TRISF4 = 1; //configure data pins for I/O
   TRISFbits.TRISF5 = 0; //disable input change notice on pins

   LATBbits.LATB12 = 1;

   BYTE rData;
//init SPI
   //IEC1CLR = 0xFFFFFFFF; //03800000; // disable all interrupts
   //SPI4CON = 0; // Stops and resets the SPI1.
   rData = SPI4BUF; // clears the receive buffer

  // IPC8bits.SPI4IP = 0b011; // Set IPL=3, Subpriority 1
  // IPC8bits.SPI4IS = 0b01;
  // IEC1bits.SPI4TXIE = 1;//0x03800000; // Enable RX, TX and Error interrupts
  // IEC1bits.SPI4RXIE = 1;
   SPI4BRG = 99; // use FPB/50 clock frequency->400khz for init mode
   SPI4STATCLR = 0xFFFF; //0x40; // clear the Overflow

   SPI4CON = 0b00000000000000000000000100100000; //CKP (bit 8) and CKE
need to be correct

   //IFS1bits.SPI4RXIF = 0; //clear spi4 receive interrupt flag
   //IFS1bits.SPI4TXIF = 0; //should change all RX to TX (because I'm transferring
data)
   SPI4CONbits.ON = 1;
}

void sd_init() {
   spi_init();

   int i;
```

```
unsigned char response[] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
unsigned char command[] = {0x40, 0x00, 0x00, 0x00, 0x00, 0x95};
//SPI4STATbits.SPIRBF = 0;

//send 80 clock cycles
for (i = 0; i < 10; i++) {
   xchg_spi(0xFF);
}

LATBbits.LATB12 = 0; //CS = 0

//send 160 clock cycles (delay) waiting for sd card to reach operating point
for (i = 0; i < 20; i++) {
   xchg_spi(0xFF);
}

sd_command(command, response); //cmd0

//CMD8
command[0] = 0x48;
command[1] = 0x00;
command[2] = 0x00;
command[3] = 0x01;
command[4] = 0xAA;
command[5] = 0x87; //CRC = 0x87
sd_command(command, response);

do {
   command[0] = 0x77;
   command[1] = 0x0;
   command[2] = 0x0;
   command[3] = 0x0;
   command[4] = 0x0;
   command[5] = 0x95; // CMD55
   sd_command(command, response); //R1

   command[0] = 0x69;
   command[1] = 0x40; //40
   command[2] = 0x0;
   command[3] = 0x0;
   command[4] = 0x00;
```

```
        command[5] = 0x95; // ACMD41
        sd_command(command, response); //

    } while (response[0] != 0x00);

    //read OCR register
    command[0] = 0x7A;
    command[1] = 0x00;
    command[2] = 0x00;
    command[3] = 0x00;
    command[4] = 0x00;
    command[5] = 0x95;
    sd_command(command, response);

    SPI4BRG = 3;
}

void put_mblocks(unsigned char outdata [], int numBlocks, unsigned char startAddress
[])
{
    int i, r;
    unsigned char command[6];
    unsigned char response[5];
    command[0] = 0x59; //command index
    command[1] = startAddress[3]; //start address
    command[2] = startAddress[2]; //...
    command[3] = startAddress[1]; //...
    command[4] = startAddress[0]; //start address
    command[5] = 0x95; //crc
    sd_command(command, response); //cmd25 to write multiple blocks
    //LATE = response[0];
    xchg_spi(0xFF);

    for (i = 0; i < numBlocks; i++) {

        xchg_spi(0xFC);

        for (r = 0; r < 512; r++) {
            xchg_spi(outdata[r + i * 512]);
        }

        for (r = 0; r < 2; r++) //CRC
        {
            xchg_spi(0x00);
        }
        //LATE = xchg_spi(0xFF); //DATA accepted
        do {
            response[0] = xchg_spi(0xFF);
        } while (response[0] != 0xFF);

    }
    //LATE = response[0];
    //stop tran token
    xchg_spi(0xFD);
    xchg_spi(0xFF);
    do {
        response[0] = xchg_spi(0xFF);
    } while (response[0] != 0xFF);

}

void get_mblocks(unsigned char indata [], int numBlocks, unsigned char startAddress
[])
{
    int i, r;
    unsigned char response[5];
    unsigned char command[6];
    unsigned char crc[2];
     //multiple block data read
    command[0] = 0x52; //command 18 index
    command[1] = startAddress[3]; //start address
    command[2] = startAddress[2]; //...
    command[3] = startAddress[1]; //...
    command[4] = startAddress[0]; //start address
    command[5] = 0x95;
    sd_command(command, response);
    //LATE = response[0];
    for (r = 0; r < numBlocks; r++) {
        do {
            response[0] = xchg_spi(0xFF);
        } while (response[0] != 0xFE);

        for (i = 0; i < 514; i++) {
            if (i < 512) {
               indata[i + 512 * r] = xchg_spi(0xFF);
            } else {
               crc[i - 512] = xchg_spi(0xFF);
            }
        }

    }

    //send cmd12 to end read
    command[0] = 0x4C; //command 18 index
    command[1] = 0x00; //start address
    command[2] = 0x00; //...
    command[3] = 0x00; //...
    command[4] = 0x00; //start address
    command[5] = 0x95;
    sd_command(command, response);

    //wait until no longer busy
    do {
        response[0] = xchg_spi(0xFF);
    } while (response[0] != 0xFF);

}

int find_address(unsigned char startAddress [], int instrumentID, int noteID)
{
    switch(instrumentID){
        case(Piano):
        {
            switch (noteID) {
               case(lowCnote): {
                  startAddress[3] = 0x00;
                  startAddress[2] = 0x00;
                  startAddress[1] = 0x00;
                  startAddress[0] = 0x00;
                  return 47; //returns known value of numBlocks based on note size
                  break;}
               case(Dnote): {
                  startAddress[3] = 0x00;
                  startAddress[2] = 0x00;
                  startAddress[1] = 0x00;
                  startAddress[0] = 0x2F;
                  return 48; //returns known value of numBlocks based on note size
                  break; }
               case(Enote): {
                  startAddress[3] = 0x00;
                  startAddress[2] = 0x00;
                  startAddress[1] = 0x00;
                  startAddress[0] = 0x5F;
                  return 47; //returns known value of numBlocks based on note size
                  break; }
               case(Fnote): {
                  startAddress[3] = 0x00;
                  startAddress[2] = 0x00;
                  startAddress[1] = 0x00;
                  startAddress[0] = 0x8E;
                  return 49; //returns known value of numBlocks based on note size
                  break; }
               case(Gnote): {
                  startAddress[3] = 0x00;
                  startAddress[2] = 0x00;
                  startAddress[1] = 0x00;
                  startAddress[0] = 0xBF;
                  return 47; //returns known value of numBlocks based on note size
                  break; }
               case(Anote): {
                  startAddress[3] = 0x00;
                  startAddress[2] = 0x00;
                  startAddress[1] = 0x00;
                  startAddress[0] = 0xEE;
                  return 46; //returns known value of numBlocks based on note size
                  break; }
               case(Bnote): {
                  startAddress[3] = 0x00;
                  startAddress[2] = 0x00;
                  startAddress[1] = 0x01;
                  startAddress[0] = 0x1C;
                  return 47; //returns known value of numBlocks based on note size
                  break; }
               case(highCnote): {
                  startAddress[3] = 0x00;
```

```
      startAddress[2] = 0x00;
      startAddress[1] = 0x01;
      startAddress[0] = 0x4B;
      return 47; //returns known value of numBlocks based on note size
      break; }
  }
}
case(Clarinet):
{
  switch (noteID) {
    case(lowCnote): {
      startAddress[3] = 0x00;
      startAddress[2] = 0x00;
      startAddress[1] = 0x01;
      startAddress[0] = 0x7A;
      return 65; //returns known value of numBlocks based on note size
      break;}
    case(Dnote): {
      startAddress[3] = 0x00;
      startAddress[2] = 0x00;
      startAddress[1] = 0x01;
      startAddress[0] = 0xBB;
      return 63; //returns known value of numBlocks based on note size
      break; }
    case(Enote): {
      startAddress[3] = 0x00;
      startAddress[2] = 0x00;
      startAddress[1] = 0x01;
      startAddress[0] = 0xFA;
      return 59; //returns known value of numBlocks based on note size
      break; }
    case(Fnote): {
      startAddress[3] = 0x00;
      startAddress[2] = 0x00;
      startAddress[1] = 0x02;
      startAddress[0] = 0x35;
      return 61; //returns known value of numBlocks based on note size
      break; }
    case(Gnote): {
      startAddress[3] = 0x00;
      startAddress[2] = 0x00;
      startAddress[1] = 0x02;
      startAddress[0] = 0x72;
      return 66; //returns known value of numBlocks based on note size
      break; }
    case(Anote): {
      startAddress[3] = 0x00;
      startAddress[2] = 0x00;
      startAddress[1] = 0x02;
      startAddress[0] = 0xB4;
      return 67; //returns known value of numBlocks based on note size
      break; }
    case(Bnote): {
      startAddress[3] = 0x00;
      startAddress[2] = 0x00;
      startAddress[1] = 0x02;
      startAddress[0] = 0xF7;
      return 65; //returns known value of numBlocks based on note size
      break; }
    case(highCnote): {
      startAddress[3] = 0x00;
      startAddress[2] = 0x00;
      startAddress[1] = 0x03;
      startAddress[0] = 0x38;
      return 66; //returns known value of numBlocks based on note size
      break; }
  }
}
case(Animals):
{
  switch (noteID) {
    case(lowCnote): {
      startAddress[3] = 0x00; //cat
      startAddress[2] = 0x00;
      startAddress[1] = 0x03;
      startAddress[0] = 0x7A;
      return 25; //returns known value of numBlocks based on note size
      break;}
    case(Dnote): {
      startAddress[3] = 0x00; //dog
      startAddress[2] = 0x00;
      startAddress[1] = 0x03;
  startAddress[0] = 0x93;
  return 19; //returns known value of numBlocks based on note size
  break; }
case(Enote): {
  startAddress[3] = 0x00; //dolphin
  startAddress[2] = 0x00;
  startAddress[1] = 0x03;
  startAddress[0] = 0xA6;
  return 34; //returns known value of numBlocks based on note size
  break; }
case(Fnote): {
  startAddress[3] = 0x00; //donkey
  startAddress[2] = 0x00;
  startAddress[1] = 0x03;
  startAddress[0] = 0xC8;
  return 22; //returns known value of numBlocks based on note size
  break; }
case(Gnote): {
  startAddress[3] = 0x00; //duck
  startAddress[2] = 0x00;
  startAddress[1] = 0x03;
  startAddress[0] = 0xDE;
  return 5; //returns known value of numBlocks based on note size
  break; }
case(Anote): {
  startAddress[3] = 0x00; //frog
  startAddress[2] = 0x00;
  startAddress[1] = 0x03;
  startAddress[0] = 0xE3;
  return 12; //returns known value of numBlocks based on note size
  break; }
case(Bnote): {
  startAddress[3] = 0x00; //jaguar
  startAddress[2] = 0x00;
  startAddress[1] = 0x03;
  startAddress[0] = 0xEF;
  return 42; //returns known value of numBlocks based on note size
  break; }
case(highCnote): {
  startAddress[3] = 0x00; //lion
  startAddress[2] = 0x00;
```

```
                startAddress[1] = 0x04;
                startAddress[0] = 0x19;
                return 31; //returns known value of numBlocks based on note size
                break; }
            }
        }
    }
}
```

### 8.2.4 adc.c file:

```
#include "adc.h"
#include <stdio.h>
#include <stdlib.h>
#include <xc.h>
#include <string.h>
#include <plib.h>
#include <p32xxxx.h>
#include "delays.h"
#include "lcd.h"
#include "sd.h"
#include <sys/attribs.h>

void init_adc()
{
   DDPCONbits.JTAGEN = 0;
   IFS1CLR = 2; //clear ADC conversion interrupt
   IEC1SET = 2; //enable ADC interrupt
   AD1PCFG = 0b1101110100000001; //Configure all input pins to Analog
   AD1CON1 = 0b00000000000000001000000011100110; //Configure Sample clock
source
   AD1CON2 = 0b0000010000100000; //Configure ADC voltage reference
   AD1CON3 = 0x0000; //Configure ADC conversion clock
   AD1CON3bits.SAMC = 0b00010;    //auto sample at 2TAD
   AD1CON3bits.ADCS = 0b00000001; //TAD = 4TPB
   AD1CHS = 0x00000000; //Configure input channels- CH0+ input,
   AD1CON2bits.CSCNA=1;
   AD1CSSL = 0b0010001011111100; //set inputs to be scanned
   AD1CON1SET = 0x8000; //Turn on the ADC module
}
```

### 8.2.5 pwm.c file:

```
#include "pwm.h"
#include <stdio.h>
#include <stdlib.h>
#include <xc.h>
#include <string.h>
#include <plib.h>
#include <p32xxxx.h>
#include "delays.h"
#include "lcd.h"
#include "sd.h"
#include <sys/attribs.h>

void init_pwm()
{
   INTEnableSystemMultiVectoredInt();     // Enable system wide interrupt to
                  // multivectored mode.
   OC1CON = 0x0000;       // Turn off the OC1 when performing the setup
   OC1R = 0x0001;         // Initialize primary Compare register (duty cycle=100)
   OC1RS = 0x0001;        // Initialize secondary Compare register (value that gets loaded
into duty cycle for next period)
   OC1CON = 0x0006;       // Configure for PWM mode without Fault pin enabled
   PR2 = 5000;            // Set period

   IFS0CLR = 0x00000100;   // Clear the T2 interrupt flag
   IEC0SET = 0x00000100;   // Enable T2 interrupt
   IPC2SET = 0x0000001C;   // Set T2 interrupt priority to 7
   T2CONSET = 0x8000;      // Enable Timer2
   OC1CONSET = 0x8000;
}
```

### 8.2.6 delays.c file:

```
#include "delays.h"
#include <stdint.h>
#include <xc.h>

#define GetSystemClock() 80000000UL


uint64_t FCY;

void set_sys_clock(uint64_t val)
```

```
{
   FCY = val;
}
void delay_ms(unsigned int delayms)
{
   unsigned int tWait, tStart;

tWait=(FCY/2000)*delayms;
tStart=ReadCoreTimer();
do
{
   // do something
   _nop();
}while((ReadCoreTimer()-tStart)<tWait);  // wait auto negotiation start

}

void delay_us(unsigned int delayus)
{
   unsigned int tWait, tStart;
tWait=(FCY/2000000)*delayus;
tStart=ReadCoreTimer();
do
{
   // do something
   _nop();
}while((ReadCoreTimer()-tStart)<tWait);  // wait auto negotiation start

}
```

### 8.2.7 lcd.h file

```
/*
 * File:   lcd.h
 * Author: Chris
 *
 * Created on May 2, 2013, 12:49 PM
 */

#ifndef LCD_H
#define          LCD_H
```

```
#ifdef __cplusplus
extern "C" {
#endif
#define clear_lcd 0x51
#define set_cursor 0x45
#define underline_cursor_on 0x47
#define underline_cursor_off 0x48
#define display_on 0x41
#define display_off 0x42
#define shift_cursor_left 0x49
#define shift_cursor_right 0x4A
#define blinking_cursor_on 0x4B
#define blinking_cursor_off 0x4C

#define MainMenu 0
#define InstrumentSelect 1
#define Options 2
#define About 3

#define _line1 0x00
#define _line2 0x40
#define _line3 0x14
#define _line4 0x54

  void init_lcd_spi(void);
  unsigned char display_char(unsigned char dataOut);
  void display_string(unsigned char phrase[]);
  void send_command(unsigned char cmd, unsigned char param);
  int button_debounce(int currentCursor);
  int screen_select(int currentScreen, int currentCursor);
  void display_mainmenu(void);
  void display_instrmenu(void);
  void display_options(void);
  void display_about();



#ifdef __cplusplus
}
#endif
```

```
#endif/* LCD_H */
```

### 8.2.8 sd.h file:

```
/*
 * File:   sd.h
 * Author: Chris
 *
 * Created on May 2, 2013, 1:05 PM
 */
#define Piano 1
#define Clarinet 2
#define Animals 3

#define lowCnote 0
#define Dnote 1
#define Enote 2
#define Fnote 3
#define Gnote 4
#define Anote 5
#define Bnote 6
#define highCnote 7

#ifndef SD_H
#define         SD_H

#ifdef __cplusplus
extern "C" {
#endif


unsigned char xchg_spi(unsigned char dataOut);

   void sd_command(unsigned char command [], unsigned char response[]);

   void spi_init(void);

   void sd_init(void);

   void put_mblocks(unsigned char outdata [], int numBlocks, unsigned char
startAddress []);
```

```
    void get_mblocks(unsigned char indata [], int numBlocks, unsigned char startAddress []);

    int find_address(unsigned char startAddress [], int instrumentID, int noteID);


#ifdef __cplusplus
}
#endif

#endif/* SD_H */
```

### 8.2.9 adc.h file:

```
/*
 * File:   adc.h
 * Author: Chris
 *
 * Created on May 3, 2013, 3:31 AM
 */

#ifndef ADC_H
#define	ADC_H

#ifdef __cplusplus
extern "C" {
#endif

    void init_adc(void);



#ifdef __cplusplus
}
#endif

#endif/* ADC_H */
```

### 8.2.10 pwm.h file:

```
/*
 * File:   pwm.h
 * Author: Chris
 *
 * Created on May 3, 2013, 3:31 AM
 */

#ifndef PWM_H
#define          PWM_H

#ifdef __cplusplus
extern "C" {
#endif

    void init_pwm(void);


#ifdef __cplusplus
}
#endif

#endif/* PWM_H */
```

### 8.2.11 delays.h file:

```
/*
 * File:   delays.h
 * Author: Chris
 *
 * Created on May 1, 2013, 9:42 PM
 */
#include <stdint.h>

#ifndef DELAYS_H
#define          DELAYS_H

#ifdef __cplusplus
extern "C" {
#endif

void set_sys_clock(uint64_t val);
void delay_ms(unsigned int delayms);
void delay_us(unsigned int delayus);

#ifdef __cplusplus
}
#endif

#endif/* DELAYS_H */
```