

Synchronized Audio indexed Note Taker (SAiNT)

Sponsor: Resource Center for Persons with Disabilities

Stephen Blosser



ECE 480 Design Team 1

Facilitator: Dr. Dean Aslam

Huan Lin

Trieu Nguyen

Christopher Johnson

Li-Shian Chen

Executive Summary

People with disabilities face many challenges while completing their collegiate studies. Reviewing lecture materials poses to be very challenging for visually impaired students. There is a need for students to be able to efficiently review specific topics in a lecture recording. This eliminates the need for the student to repeatedly press rewind and fast forward to find the topic of interest. Stephen Blosser, one of the technical staff from Resource Center for Persons with Disabilities (RCPD), has approached ECE 480 Design Team 1 at Michigan State University to design a device to solve this problem. Specific aims for the project include that it be low cost, accessible, and portable. SAiNT accomplishes these goals, and is the first step towards increasing academic success for people with disabilities. SAiNT is able to record audio to an SD card, synchronize typed notes to the audio, and then upload both the recording and notes to a computer. Once on the computer, SAiNT's graphical user interface takes over. The GUI allows for the files to be imported and then synchronized. From here, the notes can be searched and then playback will begin at the time the searched notes are selected.

Acknowledgements

Design Team One would like to thank our facilitator Dr. Aslam for his advice and guidance throughout the design process. We would also like to thank our sponsor Stephan Blosser. His encouragement and willingness to help proved to be invaluable throughout the semester. We thank the ECE shop staff, Brian Wright and Greg Mulder, for their advice and guidance in part selection as well as PCB design. Finally, we would like to thank our friends and family for their continuous encouragement and support throughout the semester.

Contents

Chapter 1: Introduction and Background	- 5 -
Introduction.....	- 5 -
Background.....	- 5 -
Chapter 2: Exploring the Solution	- 8 -
FAST Diagram	- 8 -
Conceptual Design Criteria.....	- 8 -
Design 1	- 9 -
Design 2	- 9 -
Objectives of Design Specifications.....	- 9 -
Safety	- 9 -
Reliability	- 9 -
Cost	- 10 -
Portability	- 10 -
Power Consumption	- 10 -
Usability	- 10 -
Robustness.....	- 11 -
Decision Matrix	- 11 -
Design Overview.....	- 11 -
Hardware	- 11 -
Software.....	- 12 -
Implementation	- 13 -
Chapter 3: Technical Description	- 14 -
Recording Audio to an SD Card	- 14 -
Signal Pre-amplifier	- 14 -
Analog to Digital Conversion	- 17 -
MSP430 SPI Interface	- 18 -
SD Storage and SPI Communication Protocol	- 21 -
SD Write Function.....	- 24 -
Main Program	- 26 -
Recovering and Storing Audio from SD Card in Playable Format.....	- 27 -
Power Supply.....	- 28 -
3.3V Power Supply for Audio Circuit	- 28 -
Power Layout	- 29 -
5V Power Supply for Key Logger.....	- 29 -
Power Layout.....	- 30 -
SAiNT Final Power Supply	- 31 -
EAGLE	- 31 -
Keylogger	- 33 -
USB Embedded Host.....	- 34 -
Saint Graphical User Interface	- 34 -
Application Design	- 35 -
Audio Playback.....	- 37 -

Text Parsing	- 37 -
Text Searching	- 39 -
Chapter 4: Results and Test Data.....	- 41 -
Final Design	- 41 -
Audio Recording	- 41 -
User Interface.....	- 46 -
Search Functionality	- 46 -
Chapter 5: Conclusion	- 50 -
SAiNT Cost	- 50 -
Appendix 1: Technical Roles	- 52 -
Christopher Johnson	- 52 -
Trieu Nguyen	- 53 -
Huan Lin.....	- 54 -
Li-Shian Chen	- 55 -
Appendix 2: Referances	- 57 -
Appendix 3: Technical Attachments	- 59 -
Gantt Chart.....	- 59 -
Bottom PCB Layer.....	- 61 -
Top PCB Layer.....	- 61 -
Iron on Laser Printer Results	- 62 -
main.c	- 62 -
mmc.h.....	- 63 -
mmc.c	- 65 -
main.cpp.....	- 75 -
button.h.....	- 76 -
button.cpp.....	- 76 -
chooserbox.h	- 77 -
chooserbox.cpp	- 77 -
director.h	- 78 -
director.cpp	- 80 -
easy_browser.h	- 96 -
easy_browser.cpp	- 97 -
listener.h.....	- 98 -
listener.cpp.....	- 98 -
menu_bar.h.....	- 98 -
menu_bar.cpp	- 99 -
slider.h	- 100 -
slider.cpp	- 100 -
text_box.h	- 101 -
text_box.cpp.....	- 101 -
text_box_nonrealtime.h	- 102 -
text_box_nonrealtime.cpp.....	- 102 -
timer.h	- 103 -

Chapter 1: Introduction and Background

Introduction

The Synchronized Audio indexed Note Taker (SAiNT) is being designed for the MSU Resource Center for people with disabilities. The purpose of SAiNT is to assist the disabled with class work and note taking during lecture. Specifically, accessibility to blind students will be considered due to the fact that they have difficulty taking notes during lecture, and then later listening to audio recordings of specific lecture topics. SAiNT will help the end user record entire lectures as well as type notes concurrently. SAiNT will automatically synchronize these notes to the recording. Later the lecture and text will be downloaded to a computer, allowing for the user to search the notes and begin playback of the lecture at the time specific notes were taken. In other words, the device will record audio, record synchronized text, and allow for file transfer to a PC for playback

Background

Various products on the market enable synchronization of audio and text. However, there is a premium price for these devices. Solutions that currently exist include applications for the iPhone (Apple Corporation) and the smart pen (Live Scribe). The iPhone applications work by recording lectures and allowing students to type notes on the phone during lecture. The student can then find these notes later and begin listening to the lecture at the point in time the notes were taken. The smart pen is a similar device. However, the smart pen allows for the user to write notes on a special type of notebook while the pen tracks where notes are taken. The pen will record lecture as well as the location on the page the notes were written. This allows for students to place the pen at a specific point in the notes and begin audio playback from when those notes were taken. Visual depictions of both of these designs are seen below in Figure 1.

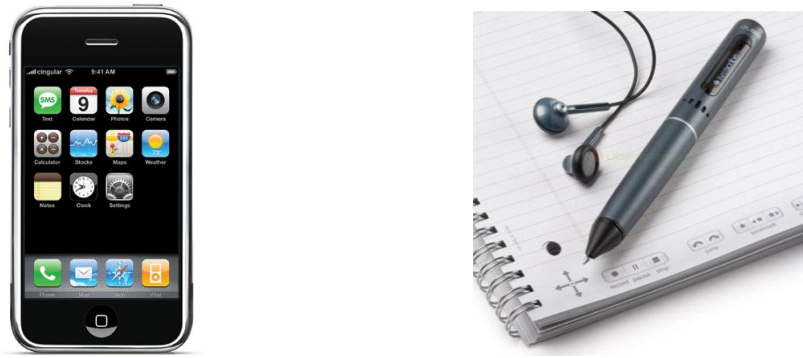


Figure 1: iPhone and Smart Pen, with its required notebook for note taking

There is one common feature that these two products are lacking aside from being priced at over \$200. These devices are not accessible to everyone. This is where there is a need for SAiNT. The keyboard will be a standard sized QWERTY layout, which will be a common standard to almost everyone. The device should also have a buzzer and LED to notify users if the device is actively recording or not. SAiNT will simultaneously record audio and text. The audio and text will be synchronized in time as well.

Various karaoke devices utilize similar technology to the design goals of SAiNT. The karaoke industry has developed a file standard for synchronizing lyrics with music files. It is known as the .lrc file standard, and is very similar to how subtitles are added to videos. SAiNT creates a text file similar to the .lrc standard using a USB hardware keylogger, as the lecture is being recorded. This effectively synchronizes all of the notes with the time in lecture they were typed.

Key components of the system include storing text and a corresponding timestamp, as well as being able to record sound from an on board microphone. The keylogger generated text file and the sound file then need to be exported to a computer. The computer will be able to play the audio file, including the text, and then search the text file for key words that were typed during lecture. The user will select the desired word or phrase and audio playback will begin at that marker.



Figure 2: Picture of the keylogger by keyllama

SAiNT utilizes a keylogger from the company Key Llama as seen above. SAiNT keeps track of the position in lecture by identifying the start of a lecture recording. This is done by pressing the enter key. One of the design's main objectives was to be low cost. Therefore, the ultimate goal is to use a single microcontroller to coordinate all of the components. However, this was not achieved in the limited amount of time for the project.

Chapter 2: Exploring the Solution

FAST Diagram

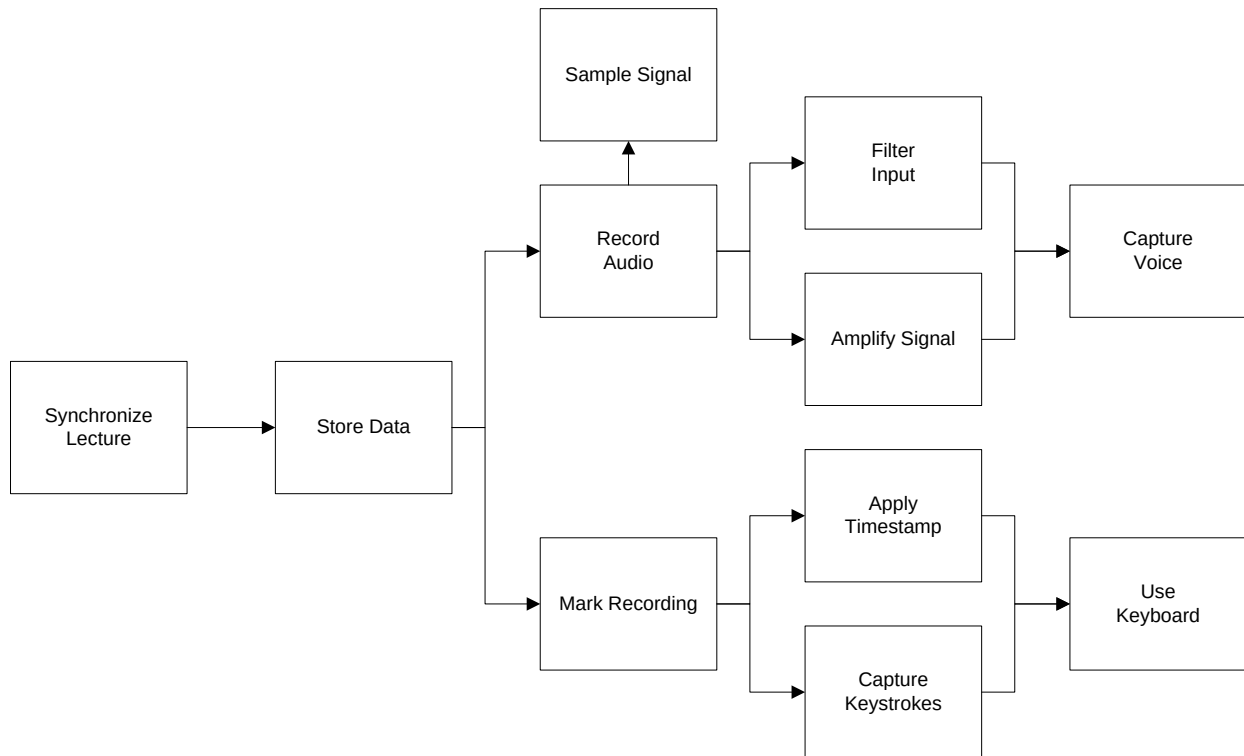


Figure 3: FAST Diagram

The FAST diagram above was created in order to help with identifying the needs and possible ways to design SAiNT. Defining the goals of the project helped to identify a strategy to successfully complete the design project. Each path in the fast diagram was treated as a specific system then the different design components were implemented as one, leading to a completed design.

Conceptual Design Criteria

Two separate hardware design approaches were consider for SAiNT. Both choices were carefully analyzed to determine the best course of action.

Design 1

Design 1 uses a microcontroller interface to write both audio and keyboard data to a source of non-volatile memory, such as an SD card or USB thumb drive. Both the microphone and keyboard will be interfaced with the microcontroller and the data will be simultaneously saved to the memory source. This could pose problems however, since two separate files would have to be saved simultaneously with a single microcontroller. There may not be enough processing power within the microcontroller to accomplish this task.

Design 2

Design 2 consists of utilizing two separate microcontrollers to handle the audio and text acquisition separately. Separating the circuits will allow for each one to be developed in parallel as well as eliminating any problems associated with processing power. This solution is more expensive since it will have two separate microcontrollers. However, due to the low cost of a microcontroller, it will not significantly increase the cost of the final product.

Objectives of Design Specifications

Several important factors were taken into consideration for which design approach was chosen, most importantly: safety, reliability, cost, portability, power consumption, usability, and robustness.

Safety

Safety was the most important aspect of the design. The electrical components of SAiNT must be carefully controlled and designed to prevent any risk of electrical shock or burns to the user. Since the device will be made to operate for long periods of time, temperature monitoring will be a critical task during development. In addition, due to the targeted audience of the device, all the physical aspects of the device must also be designed to prevent against injuries such as piercing of skin or flesh.

Reliability

Reliability was another important design criterion. In order to meet consumer expectation, the device must be able to withstand a reasonable level of usage without error.

The device needs to record both audio and text without failing. The user may not know or notice if the device has failed, making reliability even more important. Components must be selected and tested to withstand several scenarios of usage while maintaining an expected level of functionality.

Cost

SAiNT falls under the category of an accessibility device. As a result, a low cost design is desired to make the device obtainable for any interested consumer. Profit will not be taken into consideration during development. However, meeting other design specifications will take higher priority over financial decisions. Any parts that will be needed to complete the project, such as microcontrollers, must be selected carefully to ensure that unnecessary features do not drive up the cost.

Portability

This device must be small and lightweight, as it is meant to be used in a variety of different environments where space may be a factor. A battery charging system will eliminate the need to be near an electrical wall socket and decrease the amount of necessary cables. Memory will also be an important consideration as this will increase the amount and length of sessions the device can be used for without a need to unload recordings to a computer.

Power Consumption

As a result of the portability design criteria, power consumption was one of the higher priorities during development. The amount of time SAiNT will last between charges will play an important role in consumer satisfaction. The goal of the design is to allow the device to be used several times throughout the day without the need for new batteries. The electrical components must be configured and designed to minimize power consumption while maintaining a satisfactory level of functionality.

Usability

SAiNT must be easy to use for those who are visually impaired. In addition, the design should be accessible regardless of the user's age. In order to meet these criteria, the

interaction between the device and the consumer must be implemented in a simple, intuitive way. The only switch needed for the device will be an On/Off switch. The device will indicate when it has been turned on by using both an audio signal as well as an LED.

Robustness

The design of SAiNT must be very robust. The design will feature the ability to add any USB keyboard. This will allow for easier troubleshooting and cheaper repair. Microphone and audio recording quality will also be important during design to increase robustness

Decision Matrix

Design Criteria	Weight	Design 1	Design 2
Safety	7	5	5
Reliability	6	4	5
Cost	2	3	2
Portability	4	5	5
Power Consumption	3	4	3
Usability	5	4	4
Robustness	1	3	4
Total		120	122

Table 1: Ranking of Design Criteria

Both designs ranked very similar. As a result, the ultimate design came down to simplicity. In this case, the two circuit design was ideal because it allowed for each component to be developed independently. This design also eliminated any issues with processing power since the audio and text capture was completed on separate circuits.

Design Overview

Designing SAiNT was a difficult task, as no other complete solutions existed. Through extensive research, only solutions to individual components required for SAiNT were found. As a result, it was decided that a combination of software and hardware solutions could solve the ultimate problem of audio to text synchronization.

Hardware

Hardware will be used to implement an audio pre-amp. The pre-amp will amplify and apply a band pass filter to the microphone input, producing an optimal signal to be recorded.

At first a 741 op-amp will be used with a 200- 4000 Hz band pass filter. If the audio recording is not satisfactory, a higher quality op-amp will be used to decrease the signal to noise ratio of the recording. Normal speech falls in the frequency range of 200 – 4000 Hz, which determined the parameters for the filter (Don Davis). Since the highest frequency needed for recording is 4000 Hz, application of the Nyquist Theorem leads to a sampling speed of 8000 Hz (Maichen).

A Texas Instruments MSP430 was used as the microcontroller on the audio circuit. TI provides all of the communication protocol necessary to read and write to an SD card. This also led to the design decision to implement an SD card as the source of non-volatile memory. The MSP430 (Texas Instruments) is also very low cost, around \$3.33, and fast, 8 MHz, making it ideal for the project. The microcontroller was programmed to convert the analog signal from the pre-amp to a digital signal using a sampling frequency of 5200 Hz for reasons stated above. After conversion the signal was stored to the SD card.

Early in the project a text circuit was planned, however due to the time and resources required to implement the audio recording circuit, it was decided that buying a USB hardware keylogger was a more viable solution to capture keyboard input. The keylogger records keyboard input and provides a timestamp to allow for synchronization. Figure 5 below is an example of a file provided by the keylogger.

```
[2010/4/13 2:54:35PM] [Pwr]Hello everyone  
[2010/4/13 2:55:15PM]the weather outside is nice  
[2010/4/13 2:56:33PM]this file is provided by the keylogger
```

Figure 4: example .txt file from USB keylogger

SAiNT also requires a battery power supply. SAiNT utilizes a 9V batter source. This allows for easy battery changes and is also low cost. Two voltage regulators were utilized to supply the MSP430 and the keylogger.

Software

There were several software approaches that were considered to interface the SAiNT with a user's personal computer. The final decision was made to implement a C++ based, cross-

platform solution. The graphical interface was developed with Fast Light Toolkit (FLTK) and the audio playback functionality used irrKlang audio engine. Both of these packages were open source. The biggest advantage to this solution was that it did not add any additional cost to SAiNT, since both packages were free to develop with.

Implementation

Once all the components are combined, they are able to record audio, record keyboard input, and finally, synchronize the text to the audio with the software interface. The following diagram, as seen in Figure 5, is a visual representation of the design task. Audio and text were handled separately and then stored to two separate memory devices. Files can then be downloaded to a computer and presented to the user with the SAINT GUI.

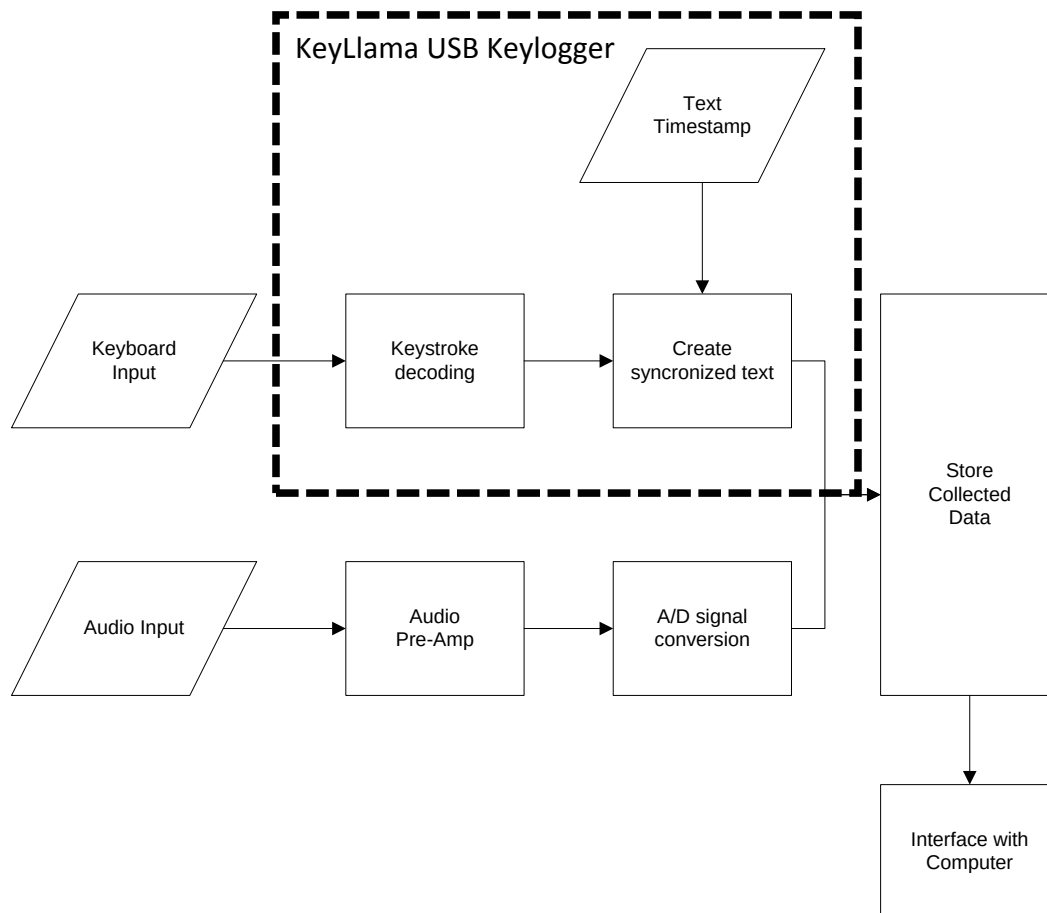


Figure 5: Block diagram of the implemented solution

Chapter 3: Technical Description

SAiNT includes many different components. Since a module design approach was taken, the work was divided up into different sections. The following chapter will outline the methods used to design SAiNT. First the audio recording interface will be discussed, followed by a discussion of the power supply and printed circuit board design. Finally, implementation of the graphical user interface will be outlined.

Recording Audio to an SD Card

Arguably the most important and most challenging aspect of the design process was recording an audio-stream to removable flash memory, and then reconstructing the signal using a computer. In order to complete this task, a cheap and easy to use source of memory was needed. The SD card was selected for various reasons. It is readily available, TI has developed source code for interfacing the MSP430, and it is low cost. The MSP430F2012 was selected as the microcontroller. The reason for selecting it was the development tools were very low cost. Also, the microcontroller itself was inexpensive. The F2012 also did not have a lot of excessive features that would not be utilized. It also has a wide variety of capabilities, including an SPI interface and analog to digital conversions, both necessary for our design. Another advantage to using the MSP430 is the large amount of control over its operation. This adds complexity to programming, but is also very advantageous to have the ability to modify how the microcontroller functions. All of the microcontroller development was completed using IAR Workbench, provided with the MSP430 development kit. However, there were still many different design challenges to overcome. In the end, SAiNT was given the ability to sample an incoming audio stream at 5.2 kHz. This audio-stream was then reconstructed from the SD memory and imported into the SAiNT GUI for synchronization to text.

Signal Pre-amplifier

Before programming the microcontroller begins, the analog input signal needs to be conditioned. An op-amp circuit is used for this. SAiNT uses the TLV2264 from Texas

Instruments. This op-amp was chosen because it is designed for low power applications and was designed to be optimized for low-noise situations. In addition to the op-amp, several filters can be added in order to further condition the input. The circuit below in Figure1 shows the implementation used in SAiNT. The circuit acts as a simple first order low pass filter, serving the purpose of capping the highest frequency willing to be passed. This is important because excessive input noise can be eliminated. This filter should be set with a cut-off frequency of $2 \times f_{\text{sampling}}$ where f_{sampling} is the sampling speed for the intended application. The low pas filter implemented in SAiNT has a cutoff frequency of about 2700Hz. This is in agreement with the Nyquist Theorem which states that to correctly reconstruct a signal; it must be sampled at twice the rate of the highest present frequency. SAiNT samples at 5200 Hz, meaning the highest frequency that can be captured is 2600 Hz. Since the filter is only a low pass, this is an acceptable value to reconstruct the signal. The filter also prevents signal aliasing, thus further enhancing the quality.

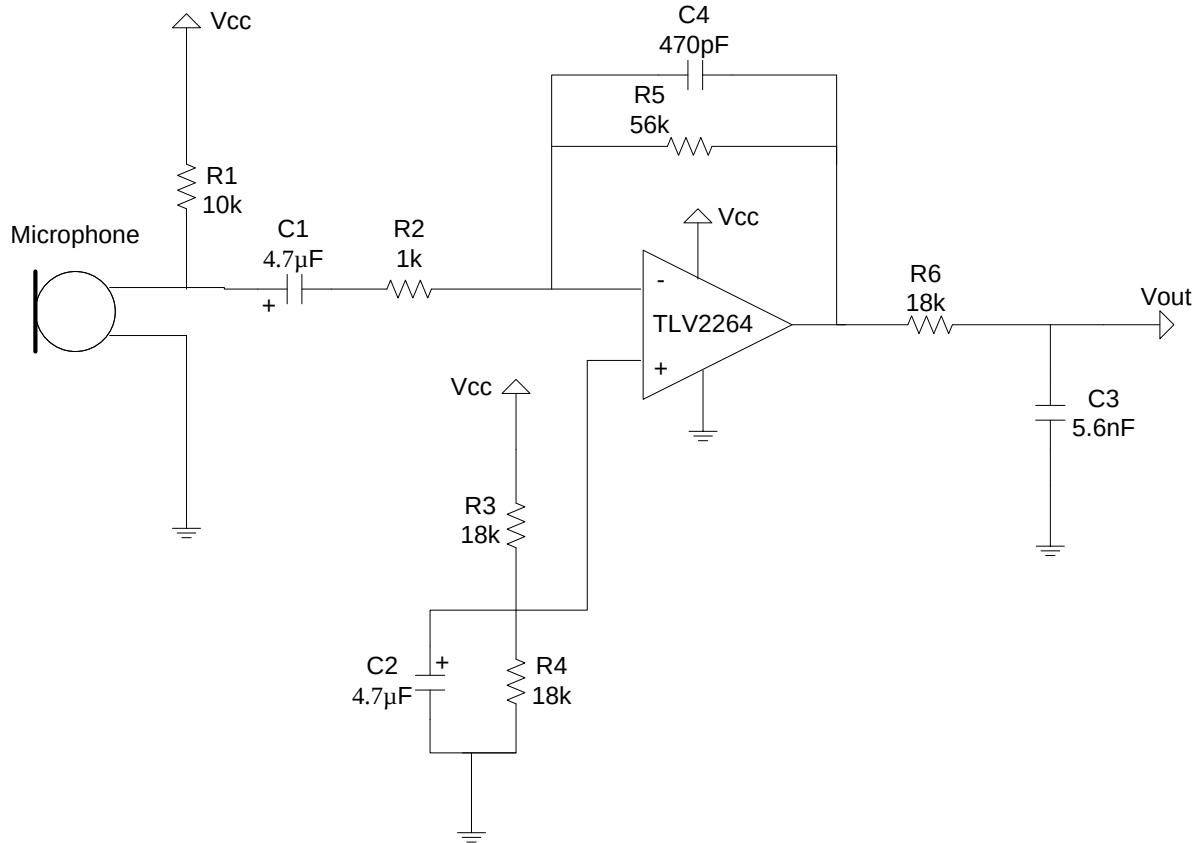


Figure 6: Pre-amp circuit design adapted from, Solid State Voice Recorder Using Flash MSP430. Texas Instruments. Application Note SLA123. January 2001. Raju, Murugavel.

Several other components must be added to the amplifier circuit in order to accurately acquire the analog signal. A pull-up resistor must be added to the positive terminal of the microphone, R1, a value of 10kΩ should suffice. SAiNT uses a 10kΩ pull up resistor which was ideal for our application. C1 acts as a high pass filter to eliminate any DC signal from entering the amplifier. C4 in the feedback loop filters out high frequency noise. C3 can be adjusted to condition the output frequency by the following equation.

$$f = \left(\frac{1}{R6 * C3} \right) * 2\pi$$

The gain of the circuit can also be tuned by adjusting the value of the feedback resistor R5 through the following equation.

$$Gain = -\frac{R5}{R2}$$

The SAiNT design achieves a theoretical gain of 56 times the input level. The maximum output voltage of the circuit will be from 0 to Vcc. Any input can only be amplified to Vcc. Any input after amplification resulting in a signal size higher than this will produce a clipped waveform and will not be able to be reproduced effectively.

Analog to Digital Conversion

The next logical step in the design process was to convert the newly conditioned analog signal to a digital value through the MSP430F2012 on board analog to digital converter (ADC10). The ADC10 provides 10 bit resolution translating to 1024 levels. The ADC was set to sample the data in 2's complement, which provides the signed data type needed to easily store and correctly import to the computer for audio playback. The source code below shows how the AD converter was initialized to make an audio sample.

```
unsigned int sample = 0;

// ADC10ON, interrupt enabled, 2.5V internal reference
ADC10CTL0 = ADC10SHT_2 + ADC10ON + ADC10IE + REF2_5V;
// input Channel is A1, store data in 2's complement
ADC10CTL1 = INCH_1 + ADC10DF;
// PA.1 ADC option select
ADC10AE0 |= 0x02;

// Sampling and conversion start
ADC10CTL0 |= ENC + ADC10SC;
//LPM0, ADC10_ISR will force exit
__bis_SR_register(CPUOFF + GIE);
//Need to save this sample for write function
sample = ADC10MEM;

// Interrupt Vector to wait for conversion end
#pragma vector=ADC10_VECTOR
__interrupt void ADC10_ISR(void)
{
    // Exit Interrupt when AD conversion is finished
    __bic_SR_register_on_exit(CPUOFF);
}
```

The AD converter on the MSP430F2012 requires several different bits to be set in order to function properly. The `AD10CTL0` register controls several general operation settings. For our application, the following bits needed to be activated:

<code>ADC10SHT_2</code>	This bit sets the sample hold time to 16 clock cycles
<code>ADC10ON</code>	This turns on the AD converter
<code>ADC10IE</code>	This enables the interrupt service routine to be used
<code>REF2_5</code>	This sets the AD reference voltage to an internal precision 2.5V source

Much like the `AD10CTL0` register, the `AD10CTL1` register also controls several of the AD converters operating settings. `SAiNT` requires the following bits be set:

<code>INCH_1</code>	This bit enables port A1 for AD conversion
<code>ADC10DF</code>	This stores the acquired data in 2's complement format in the <code>ADC10MEM</code> register

`SAiNT` samples data on pin3 (or IO port A1.1) of the microcontroller. In order to enable this bit as an AD input pin, the `ADC10AE` register is set to 0x02.

The final step to AD conversion in the MSP430F2012 is to actually make the measurement. To do this the `ADC10AE` bits `ENC` and `ADC10SC` are set. Enabling the AD converter and starting the conversion respectfully. Next the interrupt service routine is called which just pauses until the conversion is complete. This ensures that the data in the `ADC10MEM` register is accurate upon exiting the routine. After exit, the AD conversion data can be sent over the SPI bus and written to the SD card.

MSP430 SPI Interface

As stated above, SD card technology was selected as the medium for external memory storage. This was due to the availability of pre-defined communication protocol between the SD card and MSP430 using the Serial Peripheral Interface (SPI). SPI communication can be achieved using both three and four wire modes. In order to interface with an SD card, the four wire mode is required. These four communication lines include, clock, chip select, data in, and

data out. In order to initialize the SPI device on the MSP430, the following function needs to be executed.

```
void MMC_initSPI(void)
{
    // Define chip select pin
    SD_CSn_PxDIR|=SD_CSn_PIN;
    SD_CSn_PxOUT|=SD_CSn_PIN;

    // Enable 16 bit register
    USICNT |= USI16B
    // Enable SDI, SDO, SCLK, SPI master mode, data output enable
    USICTL0 |= USIPE7 + USIPE6 + USIPE5 + USIMST + USIOE;
    // Divide clock by 1 (clock divider), use SMCLK, default clock high
    USICKCTL = USIDIV_0 + USISSEL_2 + USICKPL;
    // USI released for operation
    USICTL0 &= ~USISWRST;
}
```

Just like the AD converter. Several registers need to be set in order to ensure proper function. First the `USICNT` register has the `USI16B` bit set. This allows for the entire 10 bit data to be ported to the SPI register. Once in the register, each 8 bit section can be sent independently to the SD card for storage. The main SPI control register is the `USICTL0`. In order to initialize SPI mode, the following bits need to be set.

<code>USIPE7</code>	This bit enables data to be received through the MSP430 over the SDI port
<code>USIPE8</code>	This bit enables data to be sent through the MSP430 SDO port
<code>USIPE5</code>	This bit enables the SPI clock over the MSP430 SCLK port
<code>USIMST</code>	This enables SPI master mode
<code>USIOE</code>	This enables data output through the SPI interface

The next step is to define the clock settings. To properly interface the SD card with SAI_{NT}, several bits need to be set in the `USICKCTL` register. The bits below need to be set

<code>USIDIV_0</code>	This bit sets the clock division to 1, synchronizing the SPI clock to the microcontroller clock
<code>USISEL_2</code>	This bit selects the SMCLK clock for SPI functionality
<code>USICKPL</code>	This bit sets the clock pin to be held high when not functioning

The final step is to release enable operation of the SPI interface. The final line of code in the `MMC_initSPI()` function completes this step. The microcontroller is now ready to communicate with the SD card.

In order for communication to be successful, the SD card needs to be physically connected to the microcontroller. As stated above, the 4-wire SPI interface was used. The required microcontroller connections needed for SAINTE are seen in the figure below.

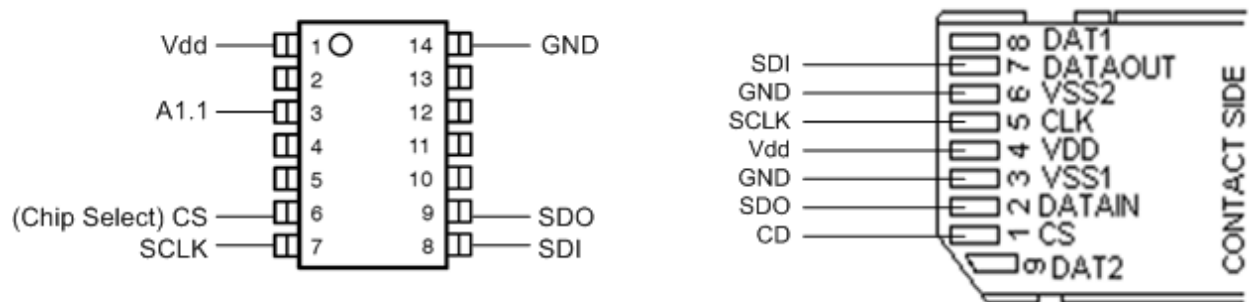


Figure 7: MSP430F2012 SPI interface with the SD card. All pin connections are shown

One point of confusion when connecting the SD card is the SDO pin on the MSP430 connects to the DATAIN pin on the SD card. Conversely, the SDI pin on the MSP430 connects to the DATAOUT pin on the SD card. The DAT1 and DAT2 pins are not utilized in SPI mode operation and are left disconnected. The required operating voltage for both the MSP430F2012 and the SD card is set to 3.3V. The picture below shows the required connections in order to operate an SD card with the MSP430F2013.

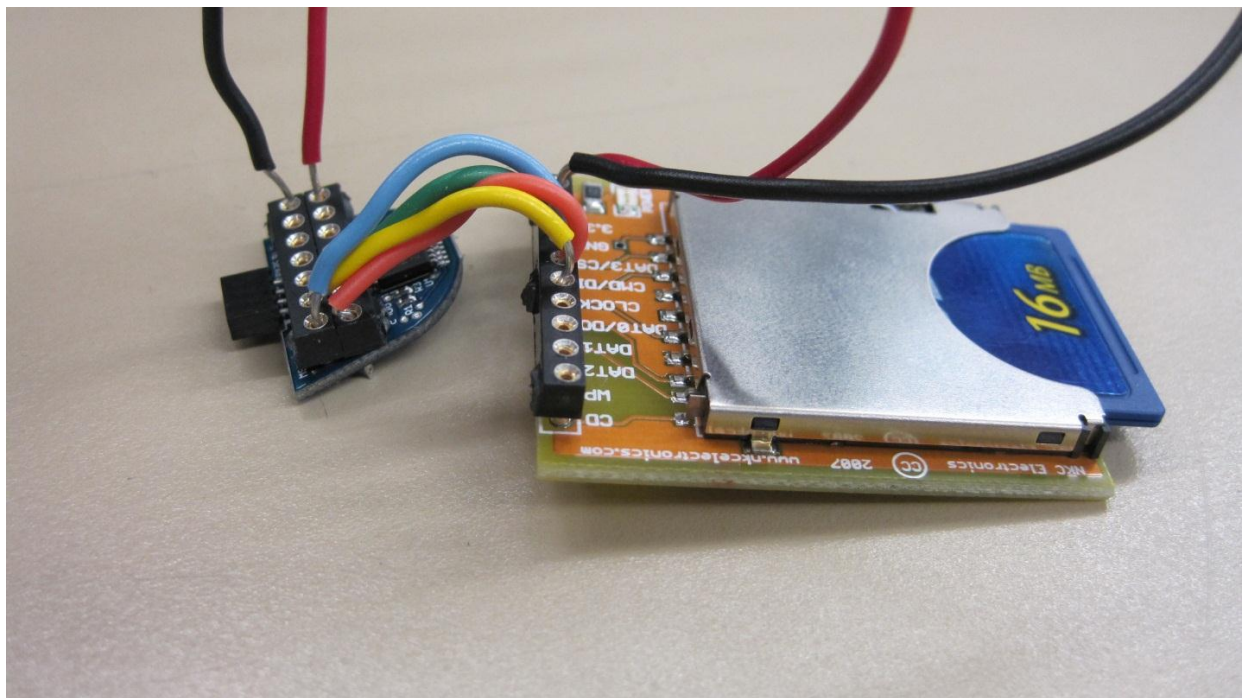


Figure 8: Figure of prototype SPI interface with SD card

SD Storage and SPI Communication Protocol

Understanding the SD card storage system and SPI communication protocol with the card was another very challenging part of the project. In order to begin communication with an SD card, the card needs to be initialized. This is done by sending the CS pin and DATAIN pin high for at least 74 clock cycles. This is done with the `mmcInit()` function. If the card is initialized into SPI mode correctly it sends a success response of 0x00. Assuming a successful initialization, the card is now ready for use. SAiNT uses the SD card to store an audio stream that the text is synchronized to. Therefore, a write function needs to be completed. However, before writing to the SD card, some background about SD data storage is needed.

An SD card has an interesting layout for storing data. The card itself is broken up into individual 512 byte sectors. Each byte in a sector has its own memory address. The figure below depicts the layout of the SD card storage system, where the sectors are numbered from 0 on.

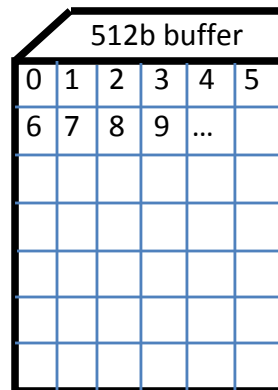


Figure 9: Memory layout of SD card. Each number represents a 512 byte sector in memory

The largest issue regards the SD card specification. The specification states that data can be read or stored in blocks on the card of 2^n bytes. However, once well into the design process, our group found otherwise. Unfortunately, some cards did not behave in this fashion. A majority of SD cards, including all of the ones design team 1 was working with, contain only 512 byte buffers. This proved to be especially problematic considering the MSP430F2012 only has 128 bytes of RAM. This means that an entire sector of the SD card cannot be sent and stored in one command. This will be explained in the next section.

First, we need to be able to send and receive data over the SPI interface. SAI_{NT} utilizes three functions to accomplish this. These functions are all very similar; however, they have very subtle yet important differences.

SAI_{NT} uses three different functions in order to send a byte over the SPI interface. The first of these functions is `spiSendByte`. The function accepts an unsigned char data type limiting it to 8 bits of data transmission. The function is seen below.

```
unsigned char spiSendByte(const unsigned char data)
{
    while (!(USICTL1&USIIFG)); // Wait for RX to finish
    USISR=data;
    USICNT = 8; // send 8 bits of data
    while (!(USICTL1&USIIFG)); // Wait for RX to finish
    return (USISRL); // Return a response
}
```

```
}
```

Basically the function just checks to make sure the SPI ports are not transmitting data, then it sends 8 bytes over the SDO port. The function then waits for transmission to finish and returns the response value in the USISRL register.

One of the major problems encountered during the development of SAiNT had to do with saving different data types. The IAR workbench compiler is very particular that all data types are defined correctly. The spiSendByte function above will only send an unsigned char data type. The AD converter of the MSP430 produces an unsigned short data type. Therefore, the spiSendByte function cannot be used to send the conversion data. This prompted the creation of the spiSendByte2 and spiSendByte3 functions. These functions are almost identical to the spiSendByte function except they take in an unsigned short data type, effectively solving our problem. The reason for the two different functions is that spiSendByte2 sends the 8 most significant bits in the conversion and spiSendByte3 function sends the remaining 2 bits and then 6 zeros to complete the other 8 bit transmission. The functions are below.

```
unsigned char spiSendByte2(const unsigned short data)
{
    while (!(USICTL1&USIIFG));           // Wait for RX to finish
    USISR=data;
    USISRL = USISRH;                      // make sure the 8 MSBs are sent
    USICNT = 8;                          // send 8 bits of data
    while (!(USICTL1&USIIFG));           // Wait for RX to finish
    return (USISRL);                     // Return a response
}

unsigned char spiSendByte3(const unsigned short data)
{
    while (!(USICTL1&USIIFG));           // Wait for RX to finish
    USISR=data;
    USISRL = USISRL;                     // make sure the 8 LSBs are sent
    USICNT = 8;                          // send 8 bits of data
    while (!(USICTL1&USIIFG));           // Wait for RX to finish
    return (USISRL);                     // Return a response
}
```

The USISR register is a 16 bit register. Data in this register is sent to the SD card through the SPI interface. In SAiNT, it is assigned to the value captured by the AD conversion. USISRH and USISRL are the 8 most significant and 8 least significant bits of the USISR register respectively.

spiSendByte2 shifts the most significant bits to the USISRL portion of the register and then sends the data to the SD card. spiSendByte3 then sends the 8 least significant bits of the AD conversion.

SD Write Function

The main workhorse function in SAiNT is the mmcWriteBlock function. This is where AD conversions are made and the data is sent to the SD card buffer, and then written to the disk. This function takes in a memory address, as well as the block length to be written. In the case of SAiNT, the memory address must be a multiple of 512 because a successful write requires an entire sector be written. The block length is fixed to be SD card sector size of 512 bytes. As stated before, the SD specification says the block size is definable; however, many cards do not support this extra level of flexibility. This disparity was one of the main design challenges because the group originally decided to attempt to write 8 bit blocks at first.

The flowchart below summarizes the mmcWriteBlock functionality. Although complex looking, the basic functionality of a block write is straight forward. Data needs to be sampled at a known frequency, and then stored to the SD card. The complexity arises in the required SD communication protocol. The SD card requires several conditions be satisfied in order to accept write data. First, the CS pin needs to be set to low, allowing data input. Next, the microcontroller is required to send the MMC_WRITE_BLOCK code, and the address to begin the write at. The microcontroller acknowledges the command and responds with either an error code or 0x00 if successful. From there the microcontroller checks that it is okay to send data, and then sends 0xFF and 0xFE to the SC card. These two codes tell the microcontroller the data transfer has begun. The function then makes a 10 bit AD conversion in 2's complement and sends the 8 most significant bits (MSBs) to the card. A small delay loop was inserted to achieve a 5200 Hz sampling rate.

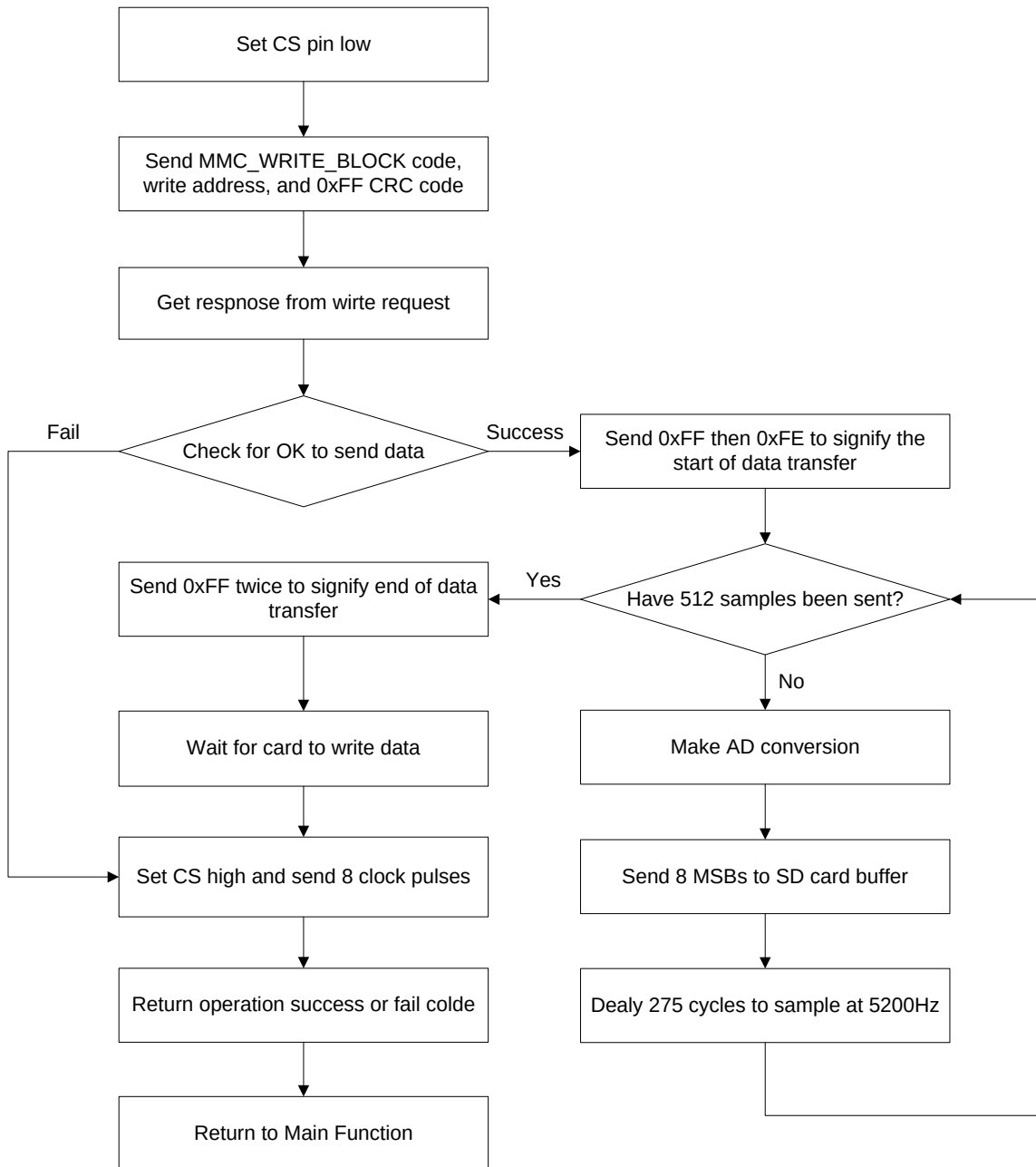


Figure 10: Block diagram of SD write function

The 2 least significant bits (LSBs) of the conversion are discarded. Although this effectively lowers the quality of the signal, it has its advantages as well. By only saving the 8 MSBs of the conversion, 512 samples can be sent to the buffer before writing the data. If the entire 10 bits of resolution were used, only 256 samples can be stored. This may not seem like a problem. However it is. Not only does sending the LSBs of the sample take processor time,

effectively slowing the maximum sampling rate, but so does saving the data to the buffer. Every time the buffer is filled (512 samples); the microcontroller needs to send an end data transfer code, signaling that the data has been sent. Once acknowledged, the SD card moves the buffered data to memory and then requires the write initialization described above to begin accepting data again. This begins to be a problem because the process takes roughly 3ms to complete. SAI_{NT} records at a 5200Hz sampling frequency, meaning a sector is filled every tenth of a second. This implication means that there is roughly a 13ms recording gap every 100ms. This issue can be heard when playing back a recording as a fast clicking sound. Once the data has been saved to the SD memory, the card sends 0x00 back to the microcontroller signaling a successful data transfer. The function has now been completed and returns to the main program. The source code for this function has been omitted, but can be found in Appendix 3 in the mmc.c file.

Main Program

The SAI_{NT} main program is surprisingly simple. It functions by continuously sampling the audio stream until power is removed, or the end of the card is reached. The main program is responsible for initializing the SPI interface, initializing the SD card, and then writing the audio stream to the SD card. The main program is summarized in the flow chart on the next page.

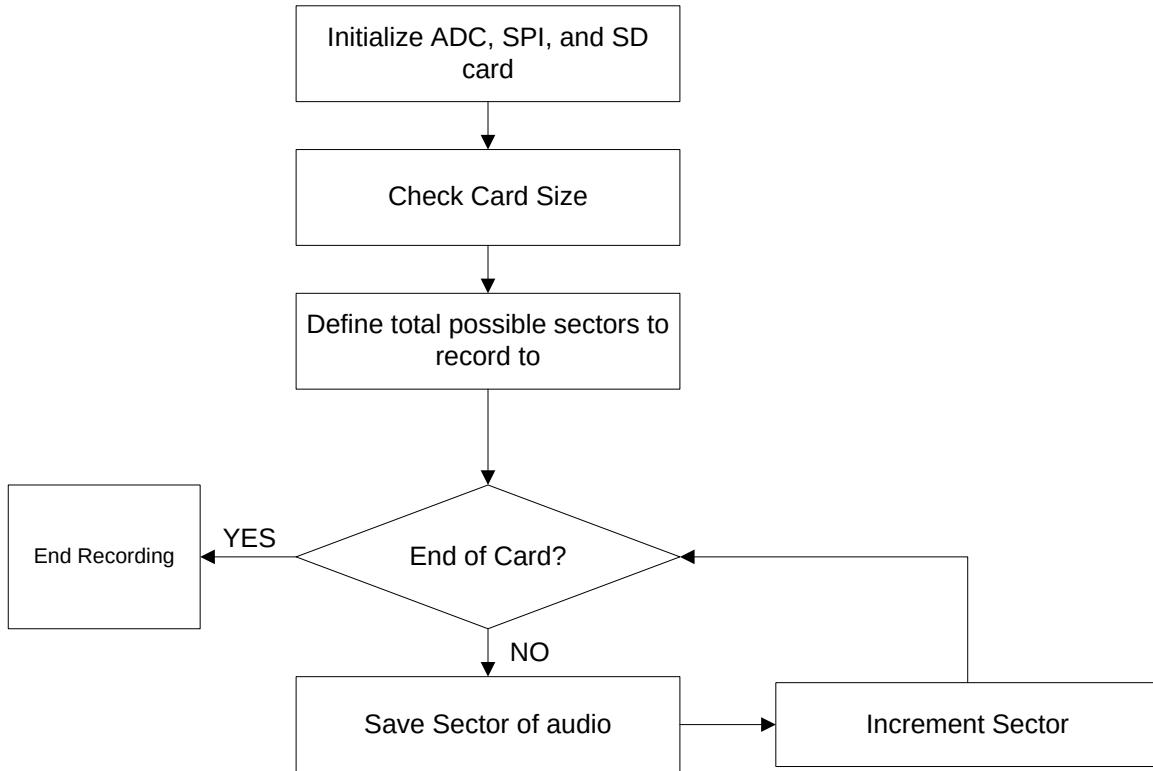


Figure 11: Block diagram of main program

The total audio recording length is dependent on the size of the SD card. An estimate for total recording time in seconds is determined by using the following formula.

$$t = \frac{\text{Card Size}}{5120}$$

Where the card size is in bytes and the time is in seconds. SAiNT has a very high upper limit for recording time. With a 2GB card, 108 hours of audio could be continuously recorded.

Recovering and Storing Audio from SD Card in Playable Format

This portion of the project proved to be the most frustrating. Once the data was stored to the SD card, it needed to be recovered. Since the MSP430F2012 is not capable of supporting an FAT16 or FAT32 file system, the data needed to be retrieved using a hexadecimal editor. HxD was chosen for two reasons. First, it has an intuitive user interface; and second, it is available free of charge. HxD enables the user to view the hexadecimal contents of drives on the computer. When the SD card is inserted, the contents can be viewed and then copied to a

file to import into a sound editor. The sound editor of choice was Audacity. It was chosen because it can import raw audio data and has many different audio processing functions. Audacity is also free and open source, which provides no cost to the end user.

Most of the issues involving with this process occurred while saving the file for upload. The original approach was to import the data into a text editor. Upon upload to audacity, the waveform did not resemble the controlled sine wave input. The reason for this turns out to be that when the hexadecimal data from HxD was pasted into a text editor, it was converted to ASCII characters, effectively eliminating the meaning of the data. In order to fix this problem, the hexadecimal data recovered from HxD was put into a new file within the program and then saved as a .raw file. The .raw file extension indicates uncompressed, raw audio data. This step ensures the hexadecimal representation of the audio signal is conserved.

After saving the .raw file audio stream, the stream needs to be imported into Audacity. SAiNT produces an 8-bit signed audio stream sampled at 5,200 Hz. In order to determine the exact sampling frequency, a 500 Hz sine wav was recorded. The data was then imported to audacity via the process above and the sampling frequency was adjusted until the Fourier transform of the recovered signal showed a peak at 500 Hz. The results of this test are further explained and presented in chapter 4. The sampling frequency and data type are then applied when importing the raw audio stream. Once imported the data is saved as either a .wav or .mp3 file and is ready for use with the user interface.

Power Supply

3.3V Power Supply for Audio Circuit

SAiNT uses the LM1117 is a low dropout voltage regulator. The audio circuit requires a fixed 3.3V power supply. The LM1117 can operate at a fixed level of 3.3V to satisfy this requirement. The designed circuit produces a well regulated 3.3V output. The regulator includes built-in overheating protection that shuts down output when the chip is too hot. The complexity of the circuit is simple and easy to build.

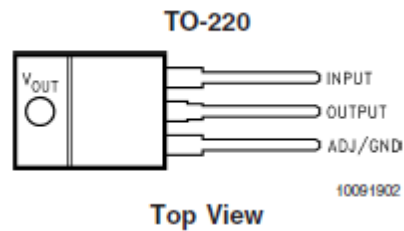


Figure 12: Connection diagram of LM1117 voltage regulator with package model TO-220

Power Layout

The Power layout for the audio circuit consists of one LM1117 voltage regulator that is fed from a 5-12V source. Since SAiNT needs to be portable, batteries are selected as the input voltage. The input voltage of this design utilizes one 9V battery. The circuit outputs 3.3V. Located below is the component layout of the implementation of the voltage regulator that is being used.

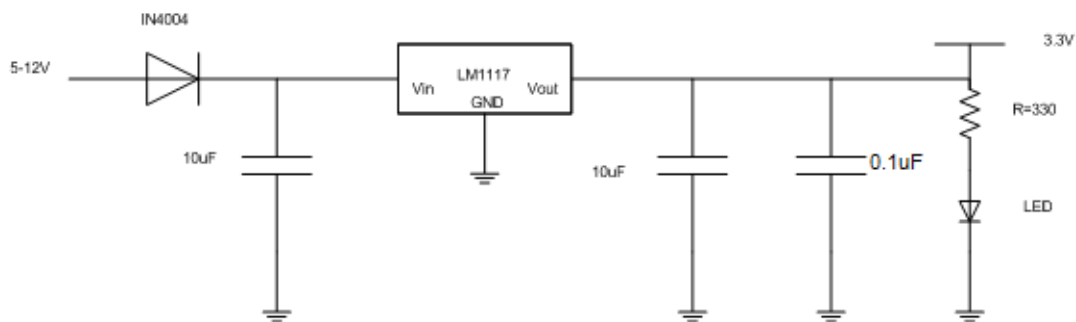


Figure 13: 3.3 V source schematic

The reason for putting a diode at the input is to protect the voltage regulator from input surge. The load resistor prevents inductive power from flowing back. Another reason to use the LED at the input is to show the user that power is on. After testing, this circuit can output a stable 3.3V to the audio circuit. If overheating is an issue, a heat sink can be added.

5V Power Supply for Key Logger

The LM7805 is a 3-Terminal Positive Voltage Regulators that can deliver up to 1A of output current. The Key Logger requires a fixed +5V power supply. LM7805 is commonly used

for +5V digital electronics. Therefore using it as a +5V fixed voltage source can satisfy the Key Logger requirement.

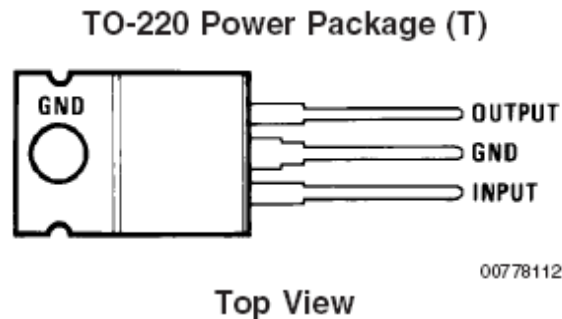


Figure 14: Connection diagram of LM 1117 voltage regulator with the package number TO-220

Power Layout

The Power layout for the Key Logger consists of one LM7805 voltage regulator that is fed from. The input voltage of this design also uses a 9V battery. This circuit produces a well regulated 5V output. This can be achieved by the internal feature of voltage regulator with external components added to the circuit. Below you can see the component layout of the implementation of the voltage regulator that is being used.

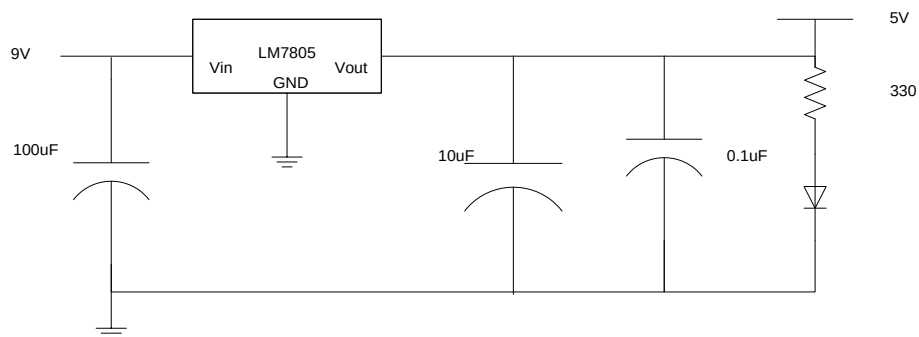


Figure 15: Schematic diagram of 5V source

The capacitors must have a high enough voltage rating to safely handle the input voltage. The reason for putting a resistor load with a LED is to prevent any inductive power flowing back. Another reason of using LED as the load is to show a signal to the user when power is on. After testing, this circuit outputs a stable 5V supply. If overheating is experienced, a heat sink can be added to the LM7805 voltage regulator.

SAiNT Final Power Supply

Based on the 3.3V and 5V power source design, this is the final design. It uses a 9V battery as input voltage. When switch is on, LED will light on to show the power is on.

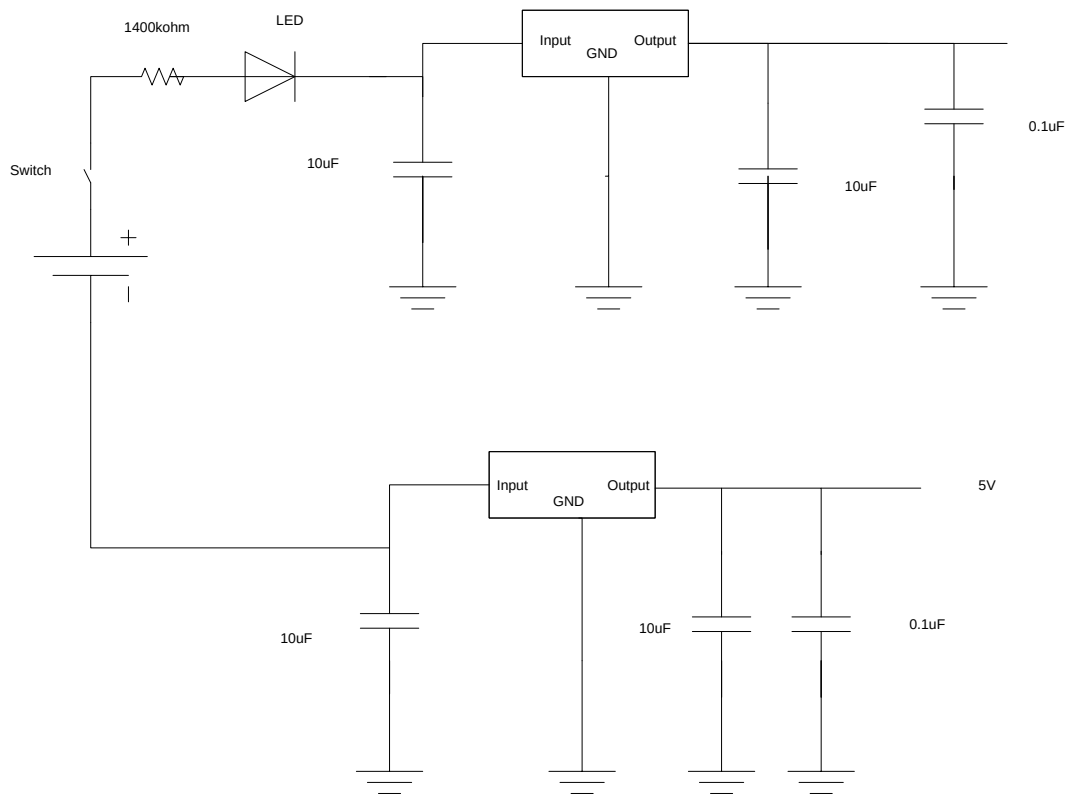


Figure 16: Complete power secematic producing obth a 3.3V and 5V source to power all components

EAGLE

PCB design was a concern of the SAiNT project. For the project, EAGLE software was used for the design. There are two different ways to design the PCB. First was to design directly on the board file. The other way was drawing the schematic and converting it into a board file. The second option was taken. The first step of the process was drawing the schematic. There are numerous libraries provided by EAGLE, which reduce the time of making custom libraries. After connecting all the components on the schematic, EAGLE can convert the schematic into a board file. Generating the board file was the second step for making the PCB. All components

need to be placed in the correct position and the traces need to be routed on the both top and bottom layers. The most important part of the PCB design is to make sure the designer checks that all the traces are routed clearly. The last step was making the PCB. Design Team 1 used a laser printer to print the top and bottom layers to color laser photo paper. Then the photo paper is ironed on the copper board to prevent the copper traces from being etched by the ferric chloride. The third step was to etch the copper board by applying ferric chloride. After the etching is done, pin holes are drilled in the board to finish the design.

The figure below shows the final SAiNT schematic. The schematic shows the integration of the two different voltage sources as well as the audio pre-amp. The power outputs go to the microcontroller, SD card mount, and USB connector for the keylogger. The audio output terminal of the connector feeds into the microcontroller

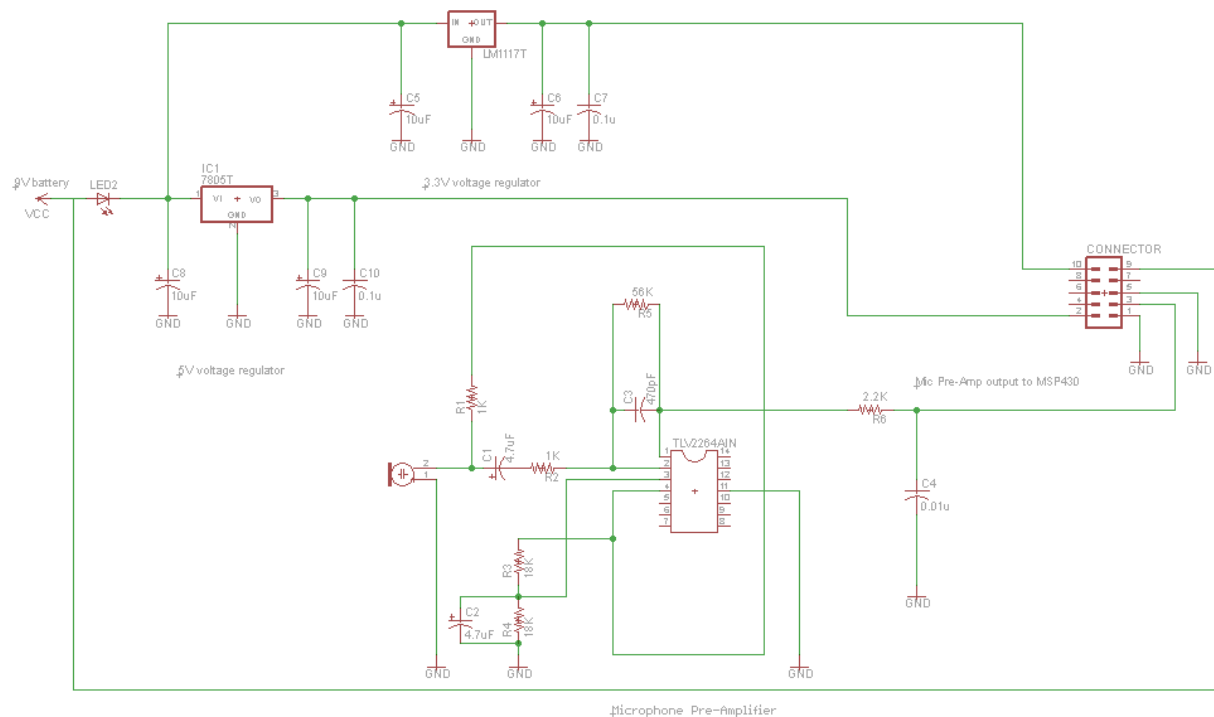


Figure 17: Full design schematic from EAGLE

The two photographs in the figure below show the final EAGLE board. The photo on the left depicts the top of the board and the right shows the bottom.

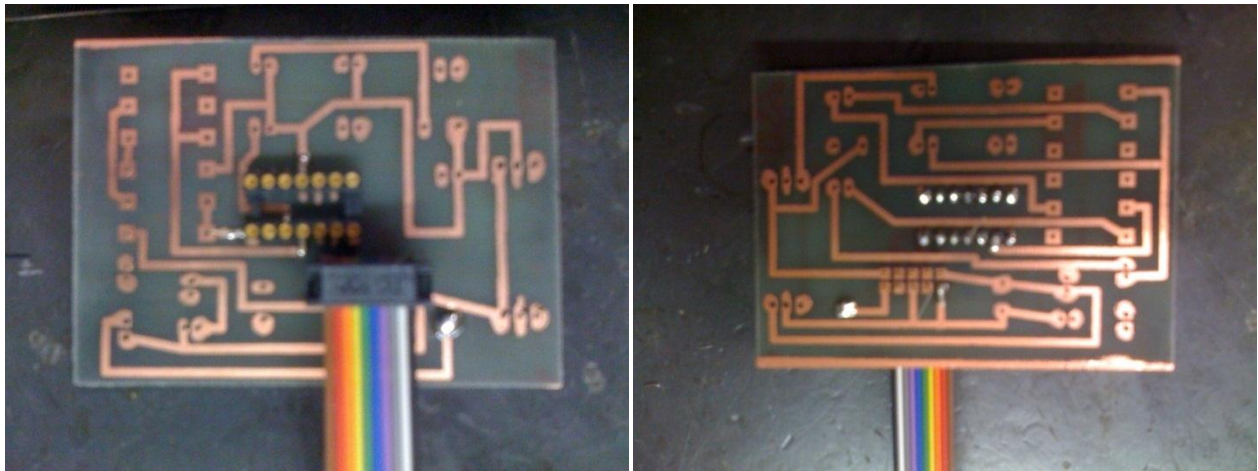


Figure 18: On the left is a top view of the completed PCB layout, the right shows the bottom view

Keylogger

A hardware based keylogger was necessary to record and store keyboard input. For this project, the KeyLlama USB 2 GB model was used. This keylogger does not only capture keystrokes from a USB keyboard, but can also insert timestamps. There are two different operation modes that the keylogger can be set to. The first is capture mode which is set by default when the device is powered on. During this mode, the keylogger works passively to intercept key strokes and stores that information in a text file. The second is retrieval mode in which the device behaves as a flash drive to allow for retrieval of the text file. This mode can be triggered by simultaneously pressing a combination of keys on the keyboard which can be specified in the device configuration. Below is an example of a text file which was created by the keylogger.

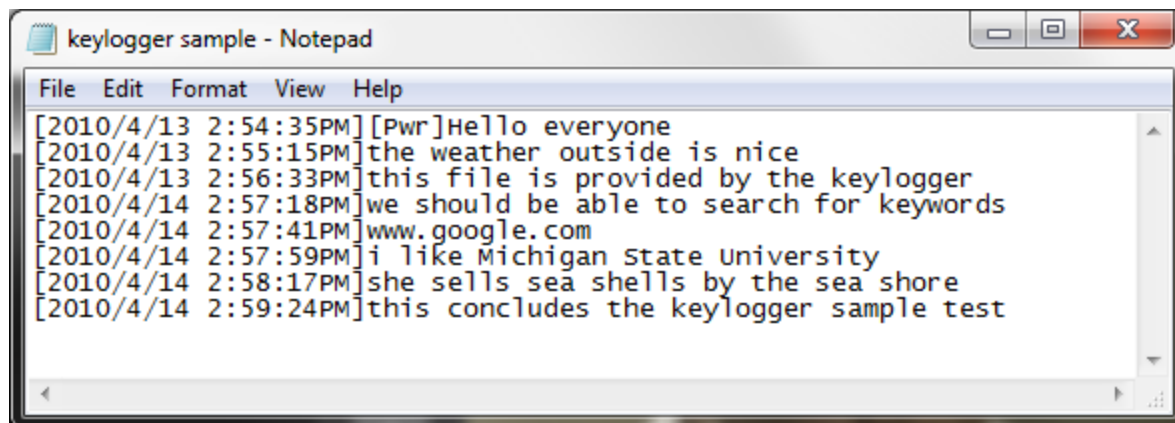


Figure 19: Example of the text file produced by the keylogger

USB Embedded Host

One thing that seemed to be seemingly trivial to the project was powering the USB keyboard and keylogger with SAiNT in order capture text. It turns out this was not the case. Even though the keylogger is a completely independent stand-alone device, meaning it will operate with only power applied, the USB keyboard is not. In order for a USB keyboard to function, it needs to communicate with a controller of some sort to properly initialize. Our team did not think this level of complexity was required to record text. Therefore, with a week left before the project deadline, the team had no way of recording text independent of the computer. In order to do this the PIC 18F4550 was needed to act as a USB embedded host to the USB keyboard, which is in the human interface device class (HID). After many attempts to host the USB keyboard, the team was unsuccessful. Recorded text can be synchronized to audio. The limitation is that the keylogger needs to be plugged into a computer. Hosting the USB keyboard could be another possible point for future work.

Saint Graphical User Interface

In order to allow users to use the data generated by the SAiNT hardware, a graphical user interface was necessary. The GUI was developed from the ground up using C++ and designed specifically to work with the SAiNT hardware.

Application Design

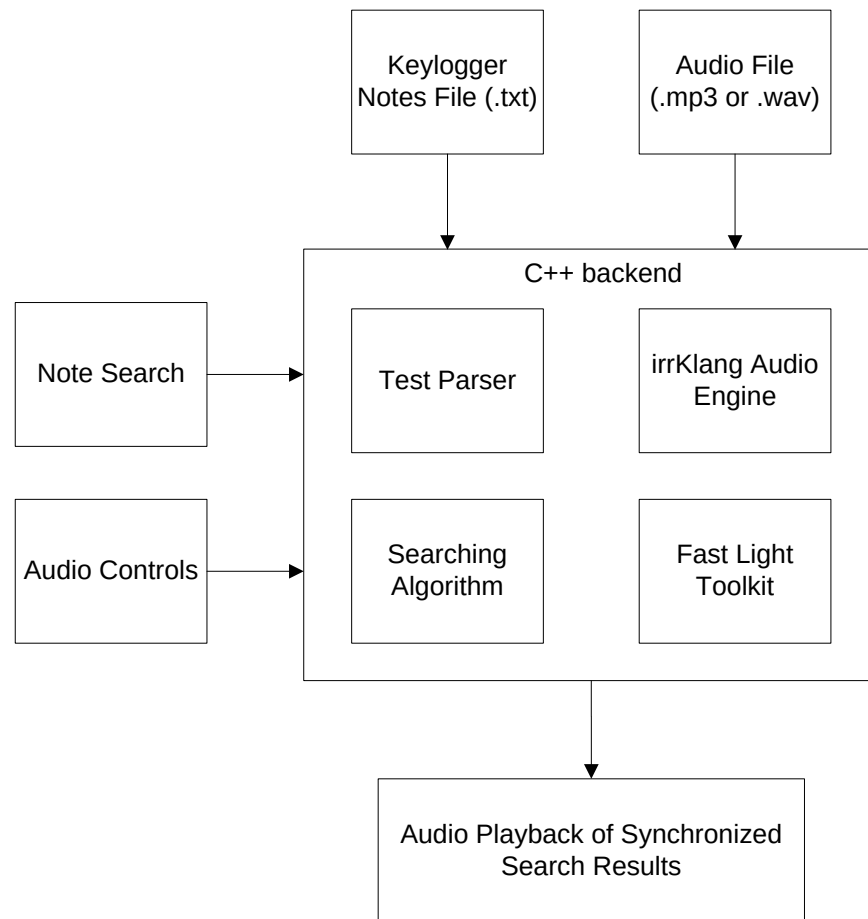


Figure 20: Application flow chart

The figure above demonstrates the functionality of the SAINTE software interface. The SAINTE graphical interface was constructed using Fast Light Toolkit, which is a cross-platform C++ GUI toolkit. An object oriented design pattern was used on the backend to handle actions on the GUI elements. Below is the final design of the user interface:

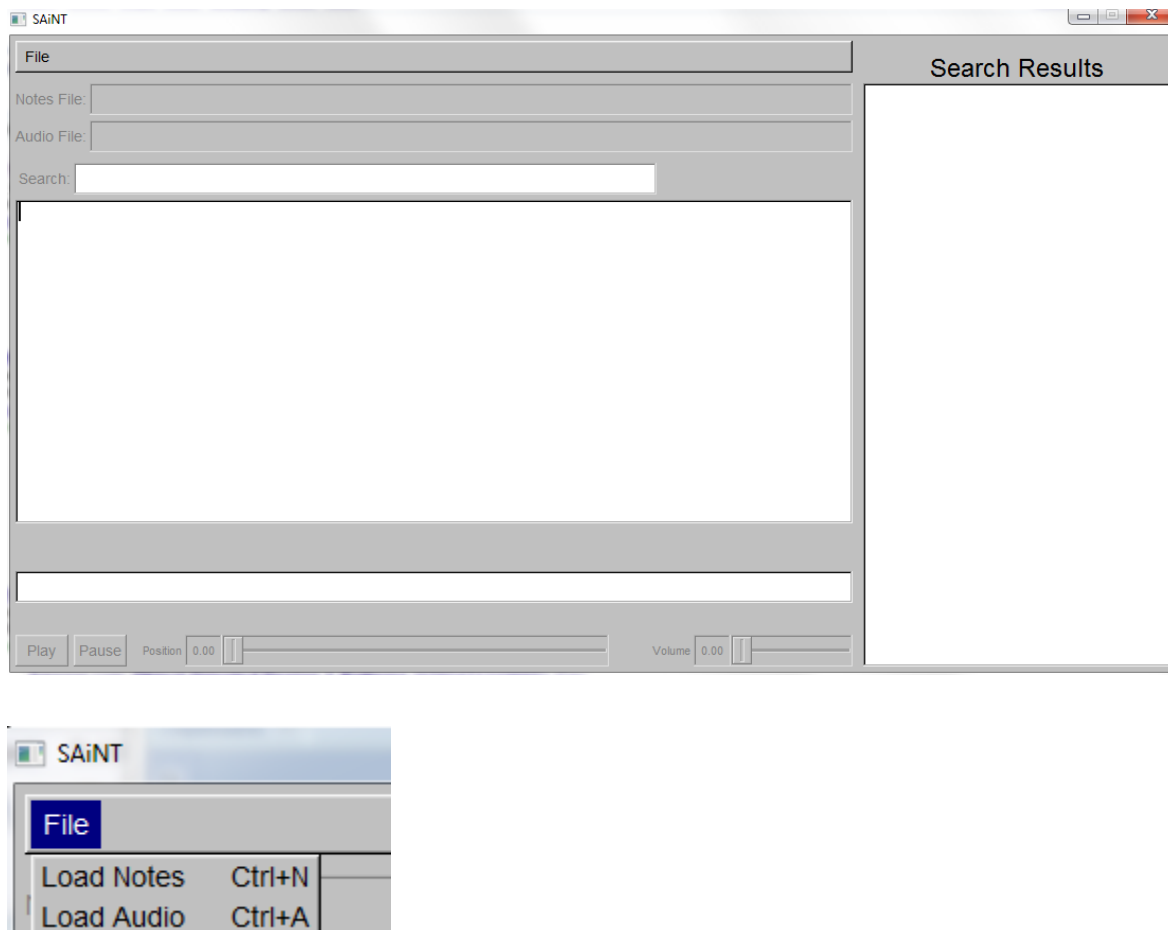


Figure 21: Photo of main SAiNT GUI and File menu

The file menu allows for users to load the lecture recording and notes file. The application will accept .txt files provided by the keylogger. Currently the audio files that can be accepted are .mp3 and .wav files. After extensive testing, both of these audio file types passed all test cases. Other audio files may be loaded into the software; however the software may not function properly as a result. Once the notes file is loaded, the file is displayed in the big center box. The audio controls at the bottom of the application allow users to control the playback of the audio file. The search box allows for users to search for keywords within the notes. Once the user enters a search word, any searching results are displayed in the box labeled as Search Results.

Audio Playback

One of the main functions of the user interface is to allow for playback of recorded lectures. In order to achieve this, the application backend uses the irrKlang audio engine programming interface. This allows the interface to have audio playback functionality without any other program dependencies.

```
//Inititalize sound engine
ISoundEngine* engine = createIrrKlangDevice();

if (!engine)
{
    cout << "The sound engine isn't working" << endl;
}

//Play an audio file
ISound* CurrentPlayingSound = engine->play2D('sample.mp3', true, true,
true);
```

This code is a simple example of how to use irrKlang to play an audio file. The user interface controls were connected to the audio API to allow users to have full control over audio playback. The image below shows the audio controls that are available in the SAiNT application.



Figure 22: SAiNT application audio controls

The controls include two buttons, Play and Pause, and two sliders, Position and Volume. The position slider allows users to scrub through the lecture recording to quickly locate a desired position.

Text Parsing

The user interface can parse text files which are created by the keylogger. When a user loads a notes file, the software will scan every line for both the timestamp and the actual notes typed by the user. This data is then stored in the backend for later use. Below is the text parsing code:

```
ifstream myfile (file_chooser.value());
```

```

//While the file is open
if (myfile.is_open())
{
    while (! myfile.eof() )
    {
        getline (myfile,line);
        char* test = strdup(line.c_str());
        string temp;
        int count = 0;
        double totalTime;

        if(test[count] == '[' )
        {
            count++;
            //While the timestamp is in the date
            while(test[count] != ' ')
            {
                count++;
            }

            //Increment 1 character past the whitespace
            count++;
            while(test[count] != 'P' && test[count] != 'A')
            {
                temp = temp + test[count];
                count++;
            }

            int count2 = 0;
            int amount = 0;
            string hours;
            while(temp[count2] != ':')
            {
                hours = hours + temp[count2];
                count2++;
            }
            count2++;

            double convertedHours = atoi(hours.c_str());
            totalTime = convertedHours * 60 * 60;

            string minutes;
            while(temp[count2] != ':')
            {
                minutes = minutes + temp[count2];
                count2++;
            }
            count2++;

            double convertedMinutes = atoi(minutes.c_str());
            totalTime = totalTime + (convertedMinutes * 60);

            string seconds;
            for(int i = 0; i < 2; i++)
            {
                seconds = seconds + temp[count2];
            }
        }
    }
}

```

```

        count2++;
    }

    double convertedSeconds = atoi(seconds.c_str());
    totalTime = totalTime + convertedSeconds;

    count = count+3;

    string finalText;
    string word = "";
    for(int i = count; i < line.size(); i++)
    {
        finalText = finalText + line[i];
        if(line[i] == ' ')
        {
            //Push each individual word into a vector and
            //the corresponding timestamp
            formattedText.push_back(StringToLower(word));
            textFile.push_back(word);
            timeStamps.push_back(totalTime);
            word = "";
        }
        else
        {
            word = word + line[i];
        }
    }

    //Push the last word into the vector
    formattedText.push_back(StringToLower(word));
    textFile.push_back(word);
    timeStamps.push_back(totalTime);
    result = result + finalText + '\n';
}

}

myfile.close();

```

This code works by, recognizing the common pattern of timestamps to retrieve the hours, minutes and seconds. These values are converted into seconds and added together so that they can be used with the audio engine. Once the time is retrieved, the notes corresponding to the timestamp are separated and stored for searching.

Text Searching

After the notes are loaded into the software, users are allowed to search for keywords. A real-time searching algorithm was implemented to accomplish this functionality. As a user begins typing in the word they are searching for, the software will rerun the searching

algorithm after each letter is typed to look for potential matches. A match is found if the search word is similar to any of the lecture notes. The software also converts all characters to lower case in the backend storage to make the search more versatile. Search function code:

```
//Search Function
void searchLoadedNotes(string temp)
{
    for (int i = 0; i < formattedText.size(); i++)
    {
        const char *ptr = strstr (formattedText[i].c_str(),
            temp.c_str());

        //A possible match was found!
        if(ptr != NULL)
        {
            searchResults.push_back(textFile[i]);
            searchTimes.push_back(timeStamps[i] * 1000);
        }
    }
}
```

This function will find any matches in the notes and then store that match and the corresponding timestamp. It works by using the strstr function which will return a pointer to the first occurrence of one string in another. A NULL pointer is returned otherwise, which allows for an easy check if a match was found.

Chapter 4: Results and Test Data

Final Design

The figure below depicts the final SAiNT design.

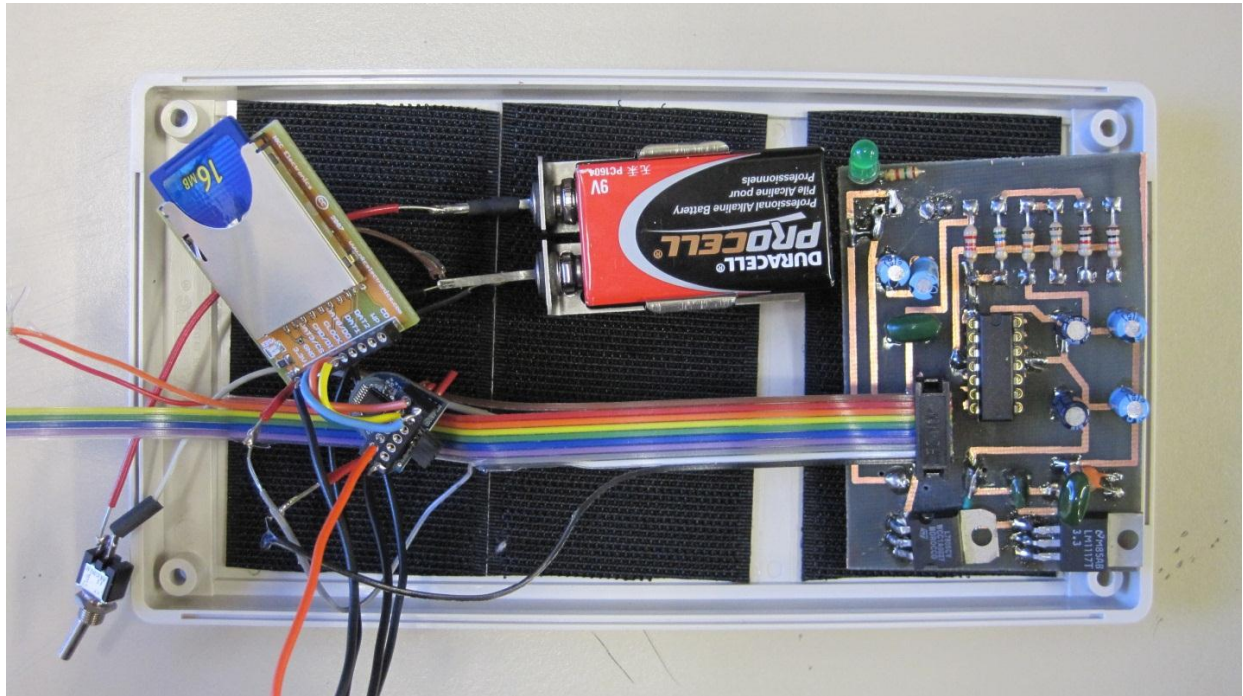


Figure 23: Final SAiNT design

The design features two separate power supplies, the SD interface as well as the audio pre-amp. In order to begin record a lecture, all the user must do is flip the on switch. SAiNT will then record until the end of the SD card is reached.

Audio Recording

As described above, the MSP430 samples an audio stream of 8-bit signed data at 5200 Hz and saves it to an SD card. The raw data is then copied and saved into a .raw file format. Below is an example of a small piece of a .raw file. Section highlighted represents an 8-bit signed AD sample. The samples composing the signal range from 00 to FF (256 levels of resolution). The samples occur sequentially beginning at the 0 location.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	55	7F	77	48	01	B8	89	89	B6	FC	46	78	7F	5A	1A	CC	U.wH. , %qúF x.Z.İ
00000010	93	87	A9	E6	33	72	7F	6C	30	E1	A3	87	99	D3	21	61	`+@æ3r.10á£+™ó!a
00000020	7F	77	40	F5	AD	89	8B	BE	08	4F	79	7D	53	0A	BE	8D	.w@ð-£< % Oy}S. %.
00000030	87	B0	F2	3F	77	7F	63	23	D7	9A	85	A0	DD	2A	6A	7F	+°ò?w.c#×š... Ý*j.
00000040	6E	32	E4	A4	83	91	C6	10	57	7D	75	46	FD	B3	88	8A	n2äxf'Æ.W)uFý²`Š
00000050	B3	FC	47	7B	7F	58	14	C9	93	84	A7	E6	34	6D	7F	6C	²ùG(.X.É`„Sæ4m.l
00000060	2A	EO	AO	85	98	D6	21	63	7F	78	44	FA	B0	8A	91	C1	*à ...“Ó!c.xDú°Š'Á
00000070	12	58	7F	7F	55	0F	C4	93	8D	B2	F7	44	7B	7F	64	21	.X..U.Ä`..²÷D(.d!
00000080	D2	97	83	9E	DD	30	6F	7F	77	3A	EC	A9	87	96	CD	18	Ô-fžÝ0o.w:i@+-Í.
00000090	5C	7F	7A	47	FF	B4	88	89	B5	FA	47	78	7F	59	0E	C7	\.zGý'`%puGx.Y.Ç
000000A0	8D	83	A7	E7	33	70	7F	66	27	DA	9D	83	9A	D6	20	63	.fSç3p.f'Ú.fšÖ c
000000B0	7F	74	40	F1	AD	87	8D	C1	0B	54	7F	7F	55	0D	C4	91	.t@ñ-+.Á.T..U.Ä`
000000C0	8B	B3	F7	45	7C	7F	66	23	D7	9A	8A	EA	E3	31	6F	7F	<²÷E .f#×šš£ä1o.
000000D0	72	38	EB	A4	84	94	CB	18	5C	7F	7A	47	FC	B9	8E	90	r8èx„“È.\.zGù¹ž.
000000E0	BF	02	4C	7C	7F	5C	12	C7	91	86	A6	E7	37	70	7F	67	ç.L . \.Ç`+!ç7p.g
000000F0	26	D8	99	83	9A	D4	1F	62	7F	70	39	EE	AB	85	8B	C1	æ@™fšÖ.b.p9i«...<Á
00000100	0E	54	7D	7D	4E	07	BC	8E	88	B0	F7	42	79	7F	62	1F	.T)}N.¼ž`°÷By.b.
00000110	D1	98	84	A4	E5	33	6E	7F	71	31	EA	AA	89	98	D2	1D	Ñ`„xä3n.q1é²%“Ô.
00000120	63	7F	7B	4A	FE	B5	89	8B	B7	02	4F	7B	7F	57	10	C7	c.{Jpu%< .O(.W.Ç

Figure 24: Sample data output from SAiNT to the SD card

In order to reconstruct the audio file, the .raw data is imported into audacity as raw audio data utilizing the known recording parameters. The figure below shows a computer generated 500 Hz tone sampled at 44100 Hz, which would be considered “perfect”. The waveform below this one is a recording of a 500 Hz wave through the SAiNT interface. As you can see, the waveforms are nearly identical, showing a quality controlled reproduction of audio.

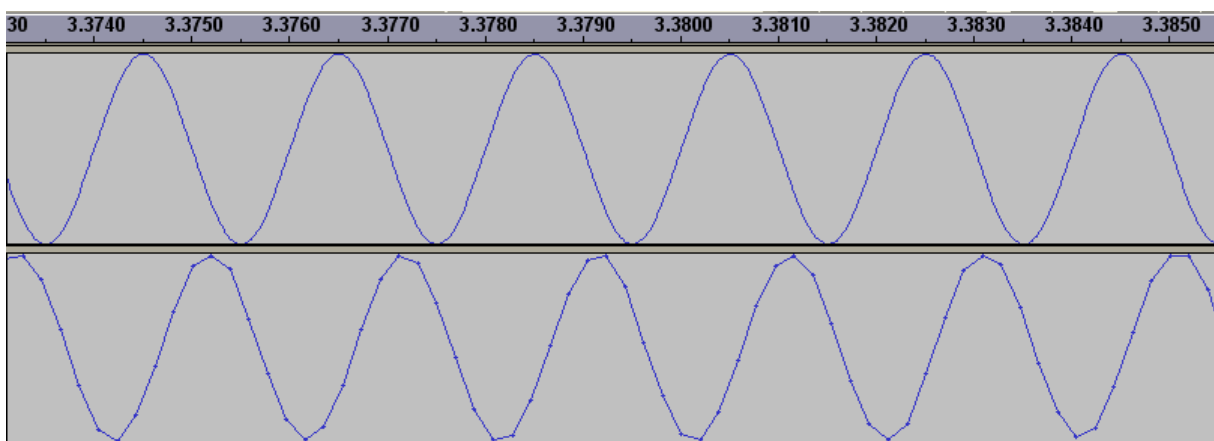


Figure 25: Above is a 500Hz waveform generated by audacity and below is a 500Hz waveform recorded using SAiNT

To further prove that SAiNT has the ability to reproduce a quality audio stream, the waveforms seen above were put through frequency analysis using MATLAB. This was done by importing the .wav file and computing the Fourier Transform. The first figure shows the Fourier transform of the 500 Hz control tone above. The only peak present is at 500 Hz showing it is in fact a 500 Hz tone.

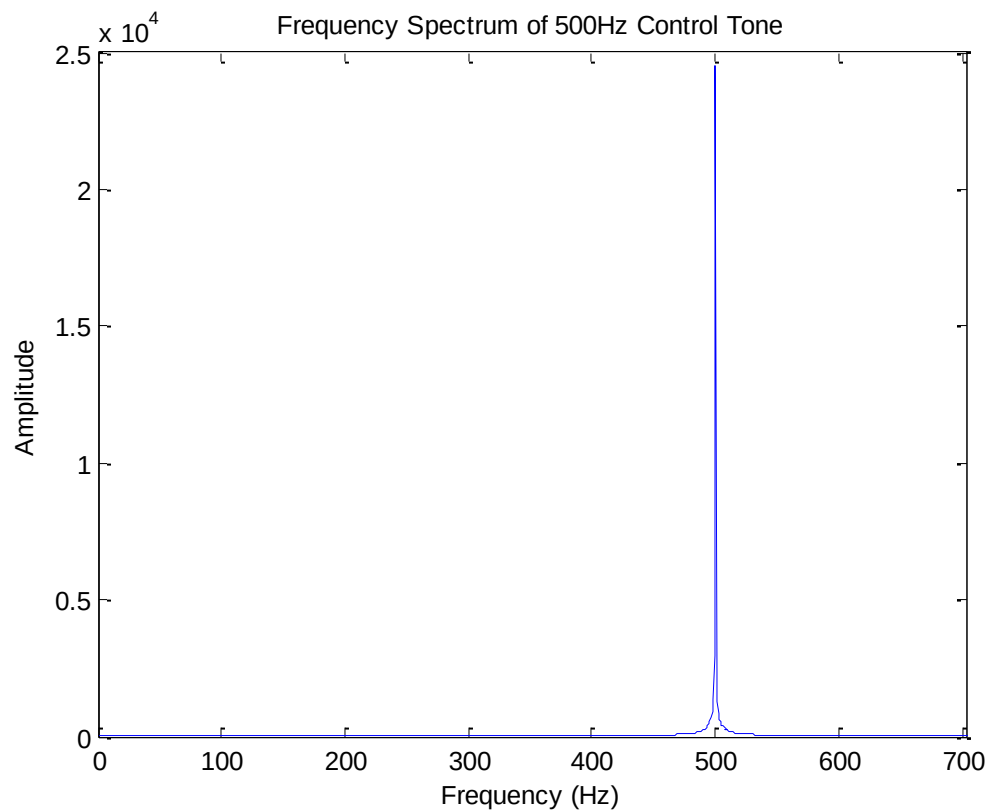


Figure 26: Frequency spectrum of 500Hz control tone generated with MATLAB

Next, the Fourier transform was computed for the SAiNT recording. As you can see, the main peak is roughly at 500 Hz with several minor peaks surrounding it, suggesting the sine wave is not completely uniform. This is expected due to the variations occurring between samples as well as the sector saving issue discussed above. Another possibility for the deviation of the main peak location could be that the sampling frequency is slightly higher than the

predicted 5200 Hz. However, the values are close enough to provide accurate sound reproduction.

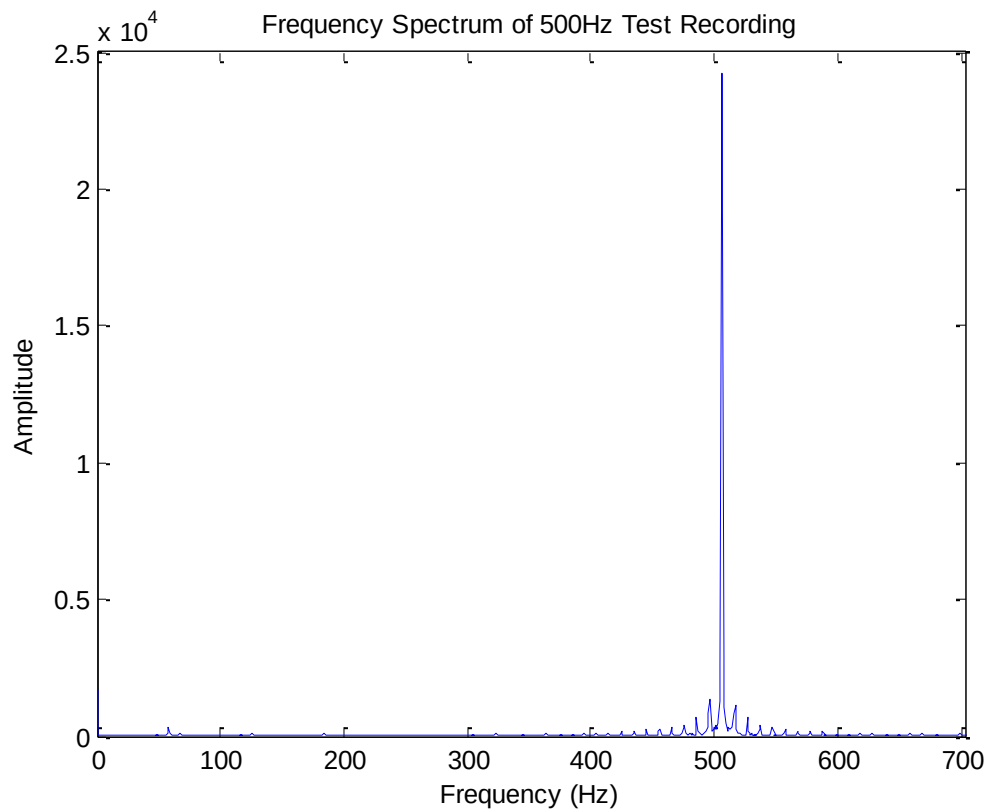


Figure 27: Frequency spectrum of 500Hz recorded tone generated with MATLAB

It was described in chapter 3 that each sector switch inserted a 13 ms gap in the audio stream. Evidence of the gap is seen below.

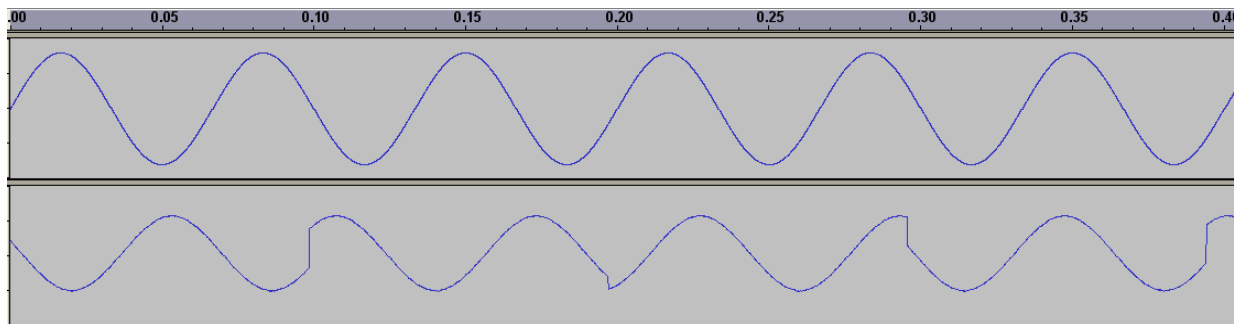


Figure 28: Above is a 15Hz wave produced by Audacity, above is a recording of a 15Hz wave with SAINT. The discontinuities occur when data is saved to a sector.

The controlled waveform is a 15 Hz signal and the waveform at the bottom was a reconstruction of a 15 Hz wave. Notice that every tenth of a second there is a discontinuity in the reconstruction. This discontinuity is evidence of the gap in recording. If there was not gap, then the waveform would be uniform and continuous.

An example of a final audio reconstruction is found below

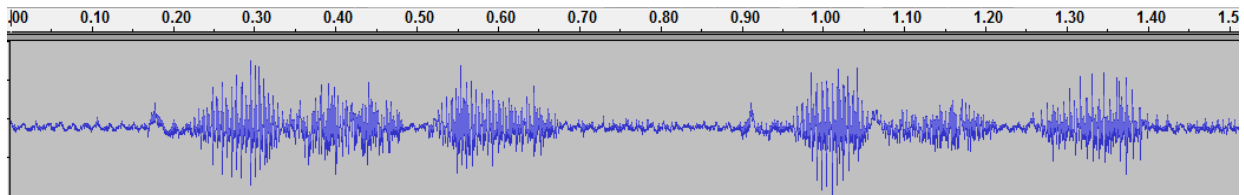


Figure 29: Reconstruction of "Testing, Testing"

This recording was made using the audio pre-amplifier. The recording is of a person saying "Testing, Testing". Notice the two similar repeated waveforms, showing the recording is accurate. When the audio signal is played, the result is a clear representation of what the input was.

User Interface

Below are examples of an actual audio and notes file loaded into the user interface.

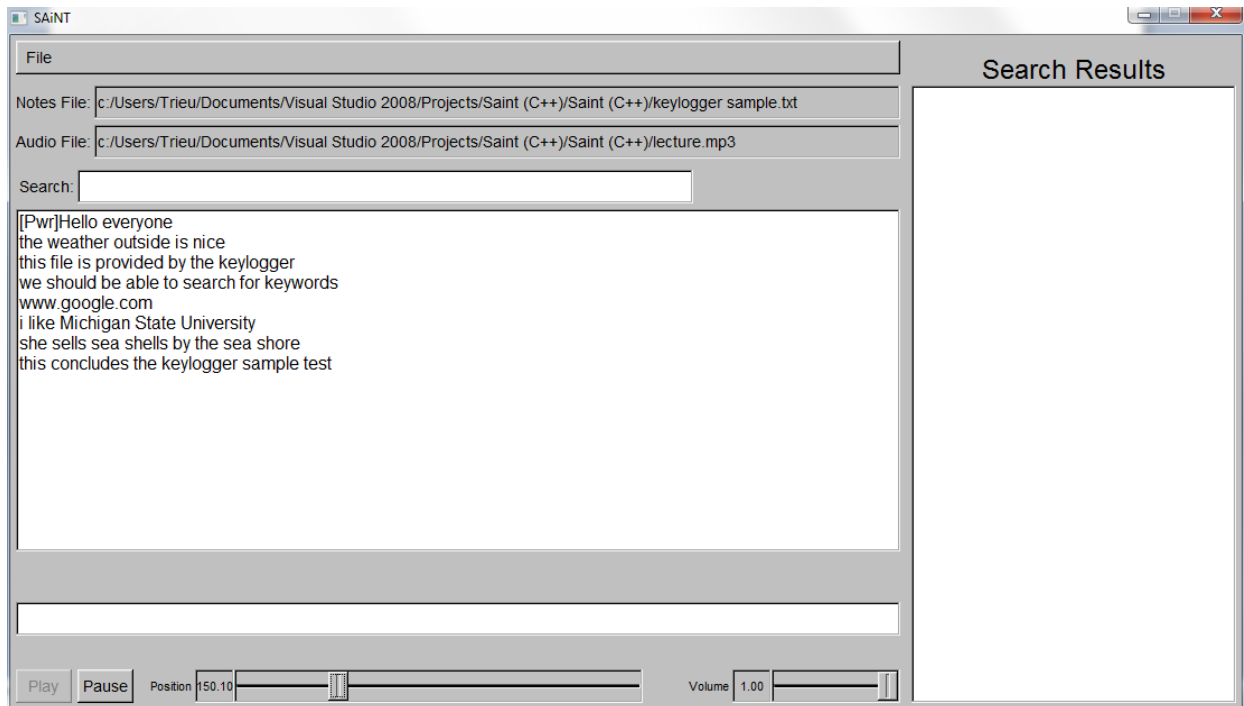


Figure 30: Screen shot of GUI with notes loaded

As you can see in figure 30, the time stamps are separated from the notes and displayed through the center output box. The interface also displays the location of both files at the top. The audio controls are also enabled to allow for control over the lecture recording.

Search Functionality

When the letter 'm' is entered into the Search box, the program finds three potential matches. These matches are then displayed in the right side Search Results box, as seen below.

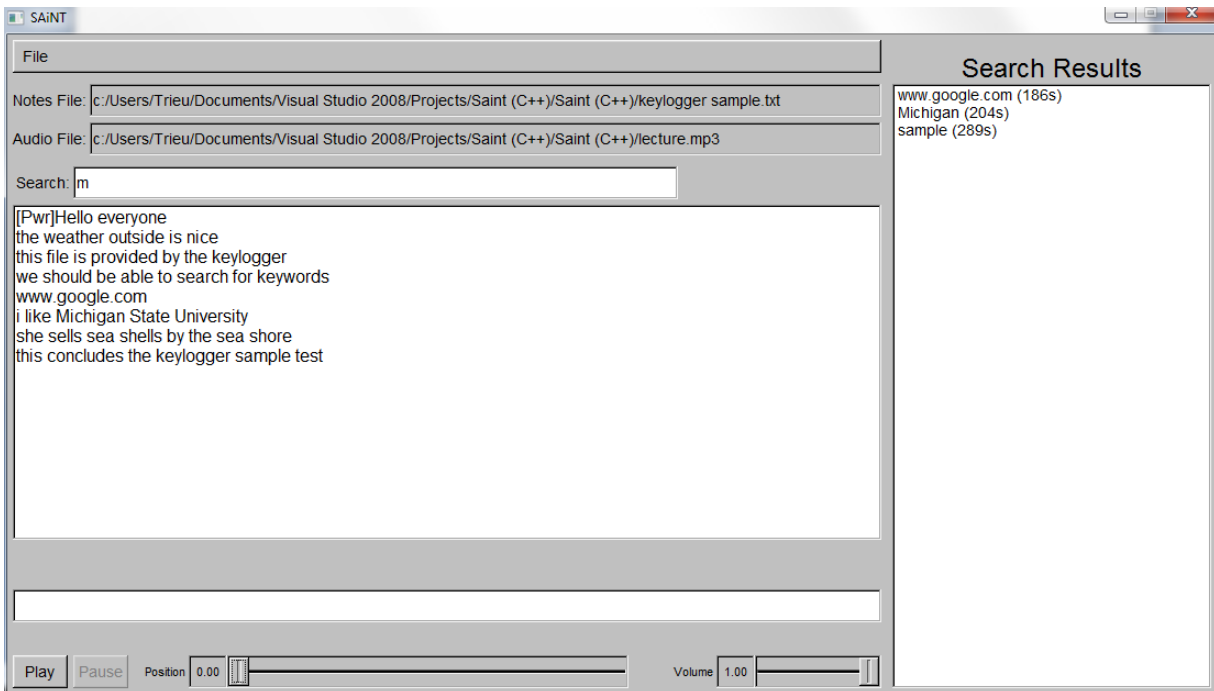


Figure 31: SAiNT Application searching functionality

As you can see from figure 31, the application will find any words in the notes that are similar to the search word. Each of these results also contain the time in which that word was typed. The results may also be selected with the mouse which will automatically move the audio position to same time the word was typed. Figure 31 shows what happens when one of the search results are selected. The result 'Michigan (204s)' is selected with the mouse. This corresponds to the word 'Michigan' in the notes and the time at which that word was typed which was 204 seconds after the recording started. By selecting this result, the audio position will jump to the result time, thus allowing users to begin playback from that spot.

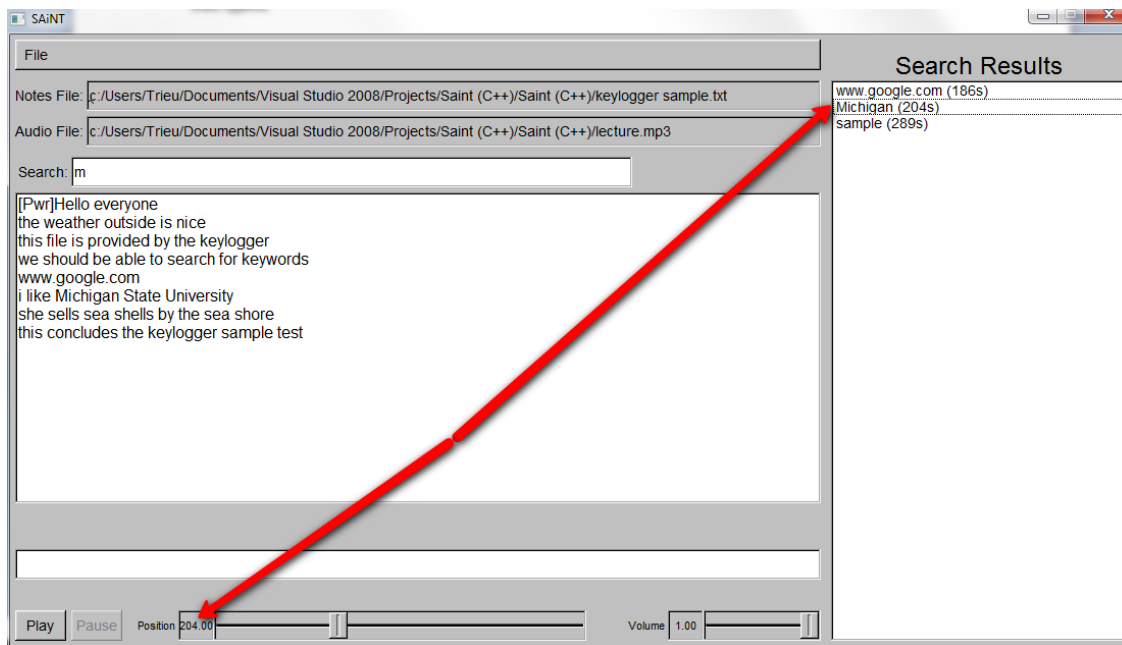


Figure 32: Selecting a search result

The previous example was demonstrated with broad search criteria, just words containing the letter 'm'. Figure 32 shows an example of a more precise search word.

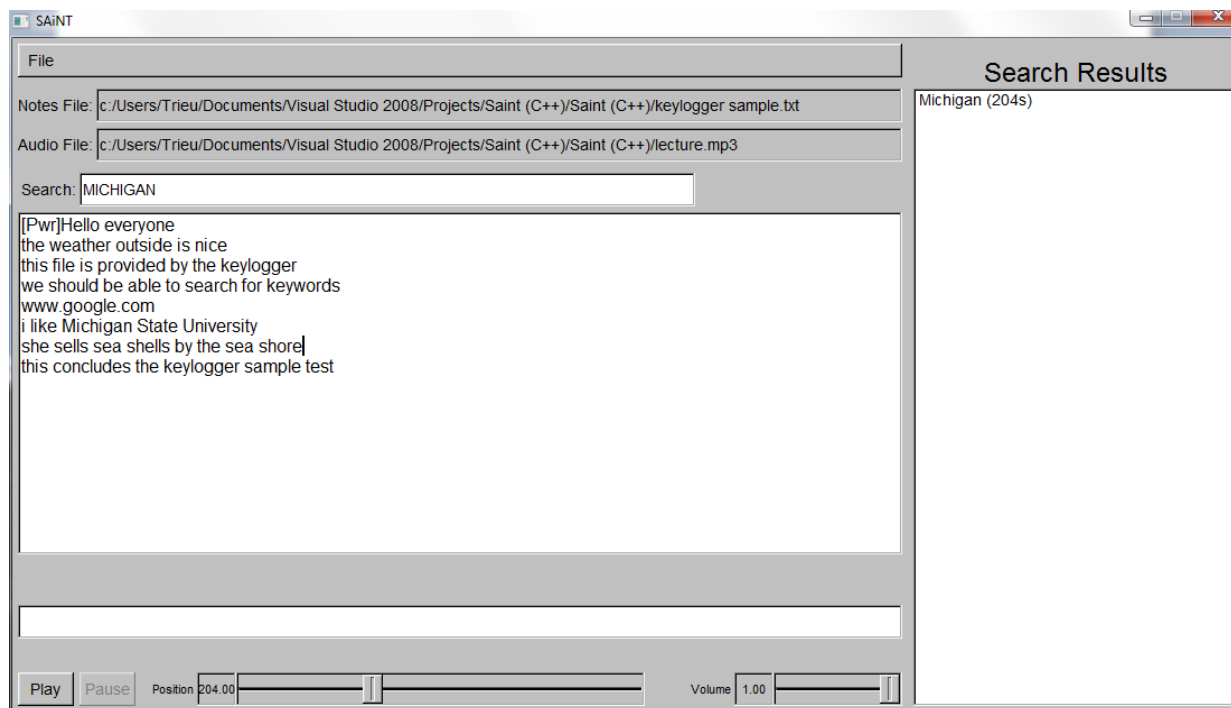


Figure 33: Searching for a complete word

In the above figure, the search word was 'MICHIGAN'. Notice that only one result was returned as there was only one word within the notes that matched every character in the search word. This allows for users to have easy control over the precision of their search criteria.

Chapter 5: Conclusion

Development of SAiNT by ECE 480 Design Team 1 ended with a successful working prototype. The device is able to record an audio stream to external memory. Once recording is complete, the signal can be imported to a computer, and be successfully reconstructed to a working audio file. The text input recorded by the keylogger can also be imported to the computer. The SAiNT interface can actively search the notes and begin playback at the point in lecture when the notes were recorded. Open source development tools were used for audio processing and the user interface. This helps to reduce the cost of SAiNT to the end user. Many of the design objectives were successfully implemented.

One of the major possible areas of improvement includes reducing the cost of the prototype. Overall, SAiNT is low cost, but the major expense in the design is implementing the keylogger. The cost of the device could be reduced by designing a keylogger, instead of purchasing an existing one. The cost of recording audio with SAiNT is \$28.21 and the keylogger is \$163.88 leaving lots of room for cost improvement. Other possible areas for improvement include increasing the audio quality. This may be achieved by sampling at a higher frequency, using a faster AD converter, and sampling with a higher resolution. Another improvement would be to support a full FAT16 or FAT32 file system. This would remove the step of having to manually extract the recording data from the SD card. However, this would require a more powerful microcontroller as well as an understanding of how to create a file system.

SAiNT Cost

qty	components	Cost each
1	TI 4 channel op-amp (TLV2264AIN)	\$2.56
1	107 keys USB standard keyboard	\$6.99
1	Kingston 2GB SD card	\$9.75
1	microphone 2.2kohm x 6mm	\$1.45
1	SD card adapter to microcontroller	\$11.99
1	KeyLlama 2GB USB keylogger	\$163.88
1	MSP430F2012 target board	\$3.33

1	3.3V IC Regulator	\$2.14
	Various Resistors, capacitors, and PCBs, enclosures, battery harness, and battery were obtained at no cost from the ECE shop and our sponsor	
	Total	\$202.09

Additional, problems with powering on the USB keyboard with the SAiNT power supply did not become known until near the end of development. The keyboard will receive power; however it will not begin data transmission without proper initialization using USB protocol. A future improvement would be to interface the keyboard to a microcontroller capable of USB device interaction.

Overall, Design Team 1 is extremely pleased with the final design results. We hope that this project will continue on in future semesters and improve on the design criteria as discussed above. We also hope that SAiNT will help to influence more research and development in the field of assistive technologies.

Appendix 1: Technical Roles

Christopher Johnson



Throughout the design process I was responsible for the design, testing, and microcontroller programming of the audio recording circuit. First I build an audio pre-amp. Next I worked on AD sampling. To do this I had to define several initialization parameters and then write the proper AD interrupt service routine allowing for the signal to be properly sampled. After the sample was taken, the result could be used for storage.

Next, I designed the SD card interface hardware. This included initializing the microcontrollers SPI communication interface. Once the proper connections were made, the SD card interface code was uploaded to the microcontroller and tested. Testing included connecting the oscilloscope to the circuit and monitoring the sent and received signals between the two devices, as well as calling a function that returned the card size and comparing the results to the known card size. Both of these results proved to be successful.

The next step in the design process was to save data to the SD card. This proved to be by far the most challenging aspect of the project. The main reason for this was that not all SD cards obey the SD card specification exactly. Also, the MSP420F2012 only has 128 bytes of memory, meaning that the entire sector of data could not be transferred at once. This problem was bypassed by making an AD conversion, and then sending the result to the SD card. The SD card would then write the data to its inherent 512 byte buffer. Once the buffer is filled, the end of data transmission is signaled, and the SD card stores the data to a sector. This process is

repeated throughout the recording time. The only problem is that it takes roughly 13 ms to write a sector of the SD card. This means that there is going to be a 13 ms gap in the audio signal every 512 bytes. The on F2012s board AD converter has 10 bit resolution requiring 2 bytes to properly store the sample leading to 256 samples per sector. To increase the samples per sector, I elected to only store the 8 most significant bits. This was advantageous because there would be a longer time interval between sampling interruptions caused by the SD card write operation. Once I was able to successfully record audio I needed to reconstruct the signal on the computer. I completed all of the testing required for successful reconstruction included using basic signal analysis to identify the correct sampling frequency

Trieu Nguyen

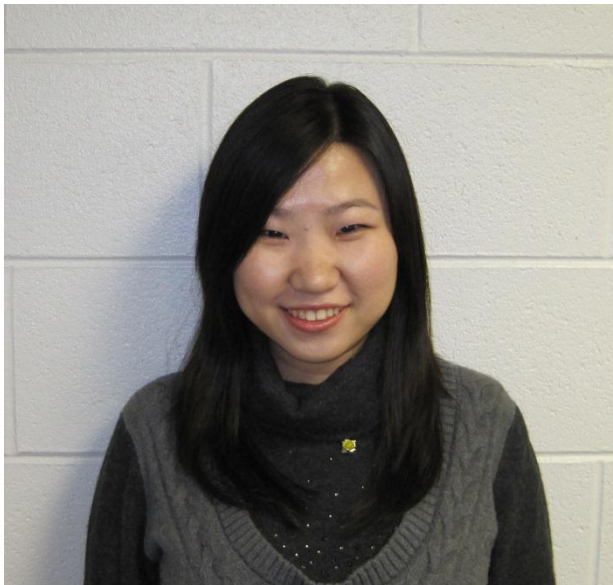


My overall portion of the SAiNT project was the graphical user interface. The goal of the user interface was handle the data produced by both the audio recording and key logging circuits and present this information to the end user. During the beginning of the design process, it was made clear by our group that a user interface would be necessary to meet our project goals. As the only computer engineer in the group, I felt that this was a good area for me to focus on. I presented

several ideas for a software interface solution. I started out with some high level ideas of some of the basic functionality that would be required. After discussing with the team, we came up with a concrete list of functionality that the software needed to have. The first functionality was that the software needed to be able to playback audio files. The second important functionality was that the software needed to have the ability to parse text files from the keylogger. The last major functionality was the ability to search through the text file to allow audio playback from certain times.

My first task was building the user interface. In order to accomplish this, I familiarized myself with Fast Light Toolkit. I put together all the elements that I thought were necessary. Once this was completed, I programmed the audio functionality and tested the GUI with several simple audio files. Next, I programmed the text parsing functionality. This required retrieving both the times and the keystroke data from the keylogger. I performed several test cases using the hardware keylogger to understand the pattern of the text files and wrote the parser accordingly. I then loaded several different text files into the software to verify that the data was being parsed and stored properly. Once this was completed I wrote the searching algorithm. I started with very basic search functionality and expanded the searching capabilities as time permitted. Once all the functionality was completed, I began end to end testing to verify that everything worked together and fixed all the loose ends.

Huan Lin



The specific area Huan was responsible for in the project was the design and testing the Power Supply of the audio circuit and the Key Logger. The design process includes researching different approaches of building power supply, understanding the features and usability of voltage regulators and DC/DC converters, and designing the appropriate circuits to satisfy the needs of the audio circuit and the Key Logger.

The audio circuit requires a 3.3V power supply. To do this, Huan chose to accomplish this from two approaches. One is using voltage regulators; the other is to use DC/DC converters. In order to select the right series of voltage regulators and DC/DC converters, the time spending on understanding the datasheets is very important. Before building the appropriate 3.3V power supply with LM1117 voltage regulator which accomplishes the need of audio circuit, Huan has satisfied building two other 3.3V power supply circuits with LM317 and LM3940. Through these

learning processes, she finds out that having a fixed voltage regulator versus an adjustable voltage regulator is better satisfying the need to the audio circuit. In addition, she has better understanding of voltage regulators' characteristic. She also satisfied designed an alternative solution with DC/DC converter TPS60100. After the completion of 3.3V power supply, she took the time of designing and testing the 5V power supply of the Key Logger with LM7805 voltage regulator and completed this target.

In addition to her technical role, she also contributed to an alternative solution of interfacing SD card with PIC18F4520 when her team mates faced a problem of using MSP430F2012. At the end of this project, the team faced a problem of communicating the Key Logger with the portable SAiNT device. This could be done by interfacing a USB Host with microcontroller. Even though there was only one week left, she took the initiative of researching possible microcontrollers and programming PIC18F4550.

Li-Shian Chen



My technical portion of the SAiNT project is the PCB design. I break my portion into four steps. The first is picking the software for the PCB design and learn how to use it. In order to design the PCB correctly, I have read several tutorials and understood how to use the CAD software. After tried out couple different software, I decided to use EAGLE. EAGLE is easy to use for beginner and it includes huge amount of libraries which is really useful for designing all kinds of schematic. Second step

was to design the schematic on EAGLE. I was able to design the SAiNT schematic after reading the tutorial. While I was drawing the schematic, I have to confirm with my teammate and make sure the design includes all components they are using and whatever they change from the prototype. Soon as I finish the schematic, I met our sponsor Stephen Blosser and showed him

the schematic of the design. After the discussion and final edit, I began the third step which was the board design. I placed all the components as compact as I could to make it small. Then I route all the traces on top and bottom layer of the board and show the design to Stephen. Soon after we check the routing and make sure there are no short traces, we began to make the PCB at in his office. We use laser printer to print out the top and bottom layer on two films and iron the film on both sides of the copper board to make sure the toner on the board will protect the copper etched by ferric chloride. When the board was done, I went to Stephen's lab and drilled all the pin holes and soldered all the components on.

Appendix 2: References

Amiera. irrKlang. n.d. <<http://www.ambiera.com/irrklang/>>.

Apple Corporation. iPhone 3G. n.d. <www.apple.com>.

C++ Timer Class <http://sites.google.com/site/jivsoft/Home/timer>

Don Davis, Carolyn Davis. Sound System Engineering Edition 2. Elsevier, 1997.

Fast Light Toolkit <http://www.fltk.org/>

Interfacing the MSP430 with MMC / SD Flash Memory Cards. Stefan Schauer and Christian Speck. Texas Instruments. SLAA281B. November 2005. Revised March 2008.

LM1117/LM1117I 800mA Low-Dropout Linear Regulator. National Semiconductor. April 2006<
<http://www.national.com/ds/LM/LM1117.pdf>>

LM340/LM78XX Series 3-Terminal Positive Voltage Regulators. National Semiconductor. July 2006 <http://www.datasheetcatalog.org/datasheet/fairchild/LM7805.pdf>

Keyllama USB keylogger <http://keyllama.com/>

KeeLog. KeyDemon TimeKeeper. n.d.
<http://www.keelog.com/hardware_keylogger_timestamp.html>.

Live Scribe. Smart Pen. n.d. <<http://www.livescribe.com/Smartpen/index.html>>.

Maichen, Wolfgang. Digital timing measurements: from scopes and probes to timing and jitter. Springer, 2006.

MSP430X2XXX Family User Guide. Texas Instruments. SLAU144E. 2008.

MSP430x20x1, MSP430x20x2, MSP430x20x3 MIXED SIGNAL MICROCONTROLLER. Texas Instruments. SLAS491E. August 2005. Revised April 2010.

Record Audio with the MSP430. Ian Lesnet. May 2008.

<http://www.diyliife.com/2008/05/16/record-audio-with-the-msp430-microcontroller/>

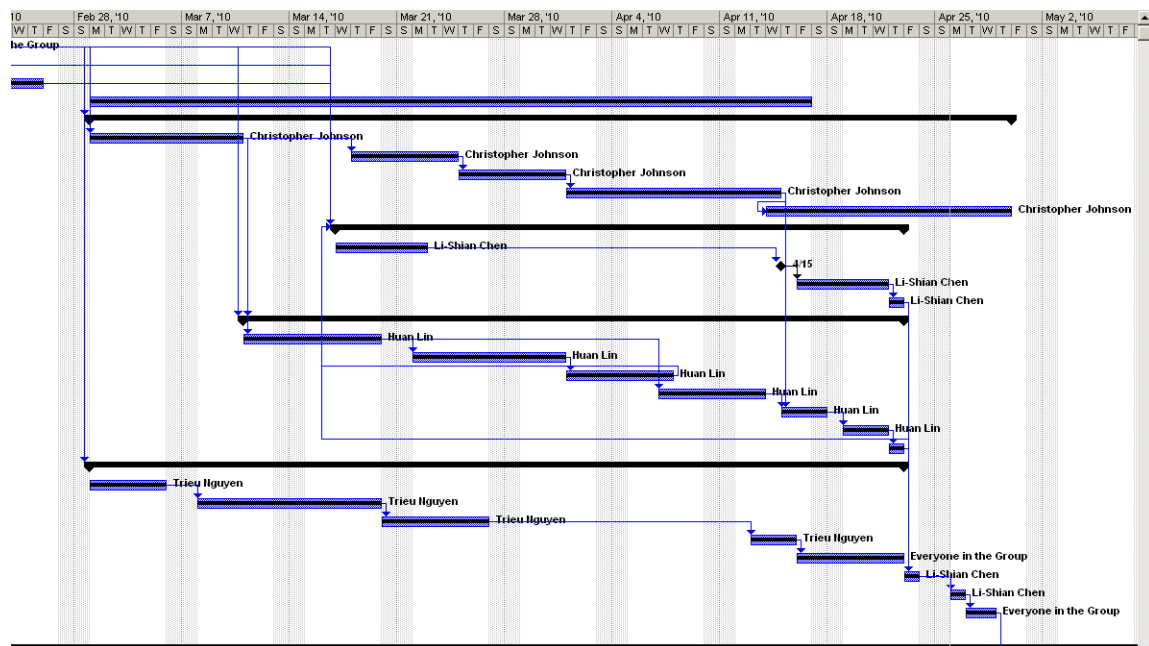
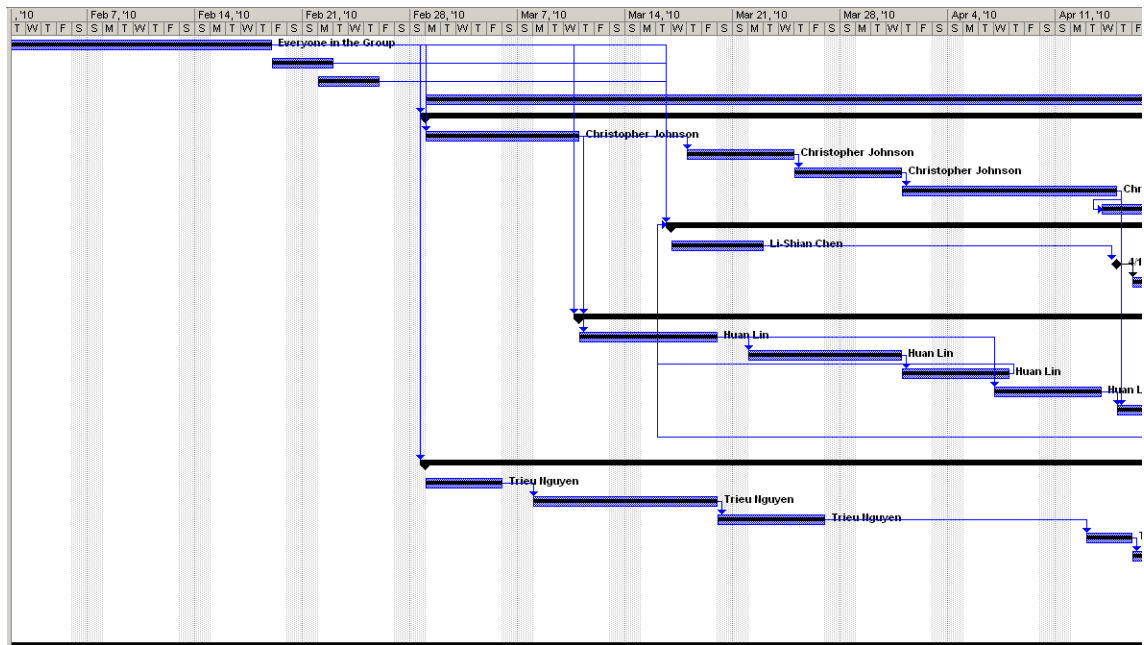
Solid State Voice Recorder Using Flash MSP430. Texas Instruments. Application Note SLA123. January 2001. Raju, Murugavel.

SD Card Specification, Simplified Version. SD Association. October 2001.

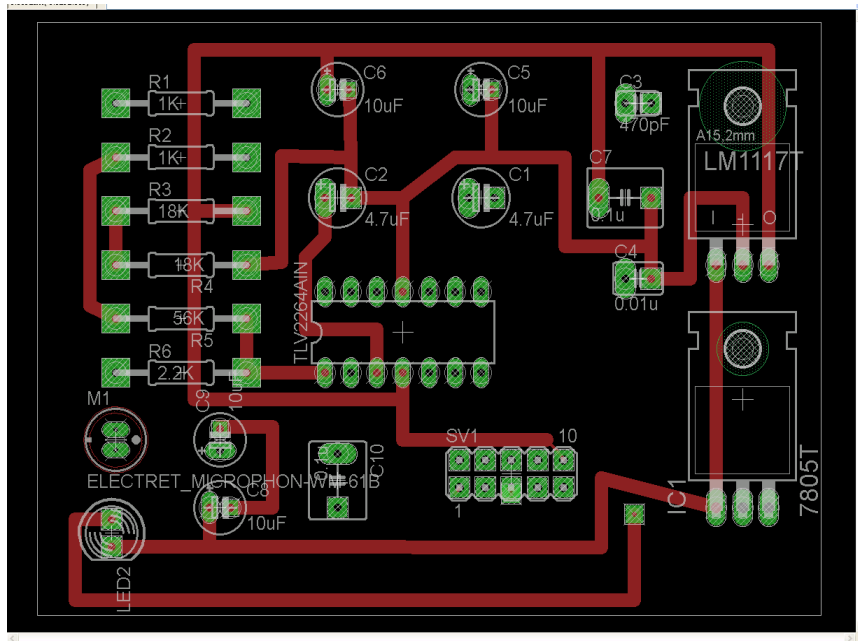
Appendix 3: Technical Attachments

Gantt Chart

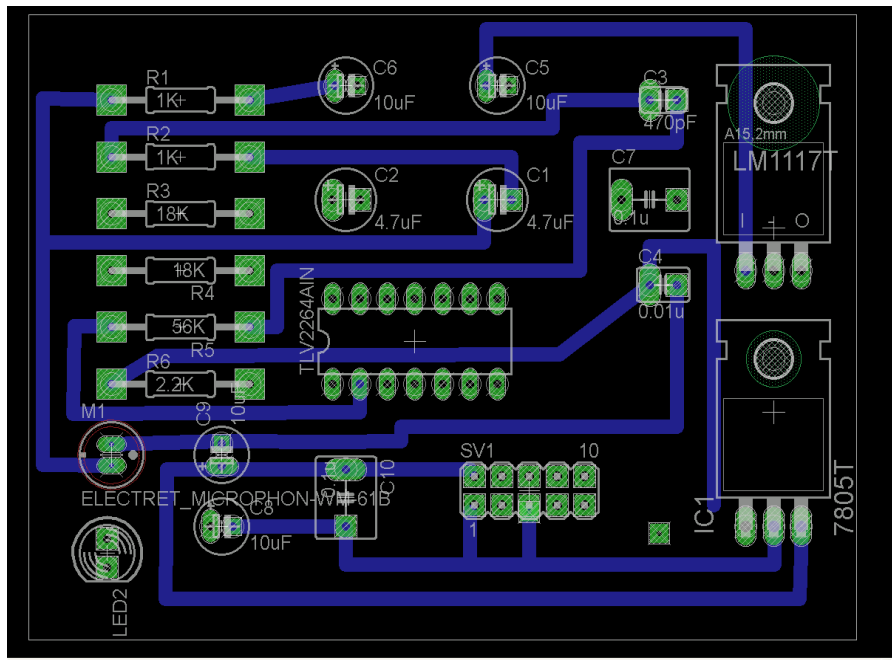
	Task Name	Duration	Start	Finish	Predecessors	Resource Names
✓	Research	20 days	Fri 1/22/10	Thu 2/18/10		Everyone in the Group
✓	Order Parts	2 days	Fri 2/19/10	Mon 2/22/10		
✓	Parts delivered	4 days	Mon 2/22/10	Thu 2/25/10		
✓	Protoboard Design	35 days	Mon 3/1/10	Fri 4/16/10		
✓	Audio Circuit and audio input microcontroller programming	44 days	Mon 3/1/10	Thu 4/29/10	1,2,3	Christopher Johnson
✓	Build the Audio Recorder Circuit	8 days	Mon 3/1/10	Wed 3/10/10	1,2,3	Christopher Johnson
✓	Microcontroller Initialization	5 days	Thu 3/18/10	Wed 3/24/10	6	Christopher Johnson
✓	ADC conversion	5 days	Thu 3/25/10	Wed 3/31/10	7	Christopher Johnson
✓	Store data to SD card	10 days	Thu 4/1/10	Wed 4/14/10	8	Christopher Johnson
✓	Audio Circuit Testing	12 days	Wed 4/14/10	Thu 4/29/10	9	Christopher Johnson
✓	Build circuit with USB keyboard input	27 days	Wed 3/17/10	Thu 4/22/10	1,2,3,6,19,23	Li-Shian Chen
✓	PCB design software learning	4 days	Wed 3/17/10	Mon 3/22/10		Li-Shian Chen
✓	PCB design for Audio Circuit	0 days	Thu 4/15/10	Thu 4/15/10	12	Li-Shian Chen
✓	PCB design for Power Supply Circuits	4 days	Fri 4/16/10	Wed 4/21/10	13	Li-Shian Chen
✓	Final PCB Design	1 day	Thu 4/22/10	Thu 4/22/10	14	Li-Shian Chen
✓	Power Source design	31 days	Thu 3/11/10	Thu 4/22/10	6,1,2,3	Huan Lin
✓	Researching appropriate approaches	7 days	Thu 3/11/10	Fri 3/19/10	6	Huan Lin
✓	Audio Circuit Power Source Design	8 days	Mon 3/22/10	Wed 3/31/10	17	Huan Lin
✓	Audio Power Source Circuit Testing	5 days	Thu 4/1/10	Wed 4/7/10	18	Huan Lin
✓	Power Source with DC/DC converter	5 days	Wed 4/7/10	Tue 4/13/10	17	Huan Lin
✓	Testing with Audio Circuit	2 days	Thu 4/15/10	Sat 4/17/10	20,9	Huan Lin
✓	Keylogger Power Source Design	3 days	Mon 4/19/10	Wed 4/21/10	21	Huan Lin
✓	Keylogger Power Source Testing	1 day	Thu 4/22/10	Thu 4/22/10	22	
✓	Computer Interface and microcontroller programming	39 days	Mon 3/1/10	Thu 4/22/10	1,2,3	Trieu Nguyen
✓	Graphical Interface	5 days	Mon 3/1/10	Fri 3/5/10		Trieu Nguyen
✓	Audio Playback Search Functionality	10 days	Mon 3/8/10	Fri 3/19/10	25	Trieu Nguyen
✓	Text Search Functionality	5 days	Sat 3/20/10	Fri 3/26/10	26	Trieu Nguyen
✓	Keylogger Interfacing	3 days	Tue 4/13/10	Thu 4/15/10	27	Trieu Nguyen
✓	Testing Interface with data input	5 days	Fri 4/16/10	Thu 4/22/10	28	Everyone in the Group
✓	Take to do PCB BoardDesign	1 day	Fri 4/23/10	Fri 4/23/10	15	Li-Shian Chen
✓	Soldering the final PCB BoardDesign	1 day	Mon 4/26/10	Mon 4/26/10	30	Li-Shian Chen
✓	Final Testing and Packaging	2 days	Tue 4/27/10	Wed 4/28/10	31	Everyone in the Group



Bottom PCB Layer

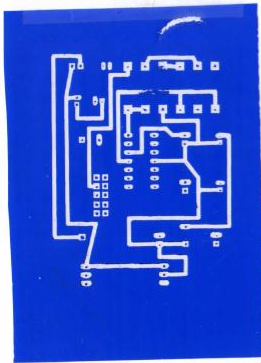


Top PCB Layer

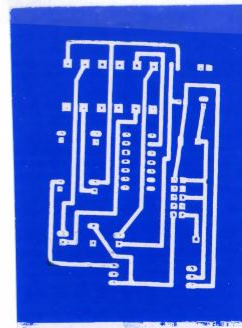


Iron on Laser Printer Results

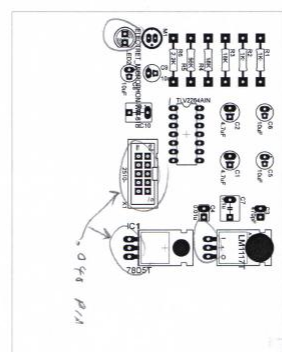
Top film (Mirror Image)



Bottom film



Components layout



Use laser printer to print the top and bottom on the plastic film which the laser toner could prevent the copper etching by ferric chloride

main.c

```
#include <msp430x20x2.h>
#include "mmc.h"

char sdstatus;
unsigned long cardSize = 0;
unsigned char status = 1;
unsigned int i = 0;
unsigned long j=0;
unsigned int sample = 0;
unsigned int totalSectors = 0;
unsigned int startSector = 1024;

int main(void)
{
    WDTCTL = WDTPW + WDTHOLD;           // Stop watchdog timer
    BCSCTL1 = CALBC1_8MHZ;               // Load DCO constants
    DCOCTL = CALDCO_8MHZ;

    ADC10CTL0 = ADC10SHT_2 + ADC10ON + ADC10IE + REF2_5V;
    ADC10CTL1 = INCH_1 + ADC10DF;        // input A1
    ADC10AE0 |= 0x02;                    // PA1.1 ADC option select
    P1DIR |= 0x01;

    // Sampling and conversion start :F2012
    ADC10CTL0 |= ENC + ADC10SC;
    // LPM0, ADC10_ISR will force exit
    __bis_SR_register(CPUOFF + GIE);
```

```

// Need to save this sample inside of a write function
sample = ADC10MEM;

// Activate SPI Mode
MMC_initSPI();
// Initialize SD card
sdstatus=initMMC();
// Read card size
cardSize = MMC_ReadCardSize();

// Calculate total writeable sectors
totalSectors = cardSize / 512 - 3;

// Begin sector by sector recording
for(unsigned long k = 0; k < totalSectors; k++)
{
    // Record sector of audio
    sdstatus = mmcWriteBlock(startSector, 512);
    // Increment beginning sector memory address
    sector = sector + 512;
}
}

```

mmc.h

```

//
*****
//
// Filename:  mmc.h:
// Declarations for Communication with the MMC (see mmc.c) in unprotected SPI
mode.
//
// Version 1.1
//   added ul declaration in macros mmcWriteSector and mmcReadSector
//
*****

// macro defines
#define HIGH(a) ((a>>8)&0xFF)           // high byte from word
#define LOW(a)  (a&0xFF)                // low byte from word

//sd card hardware configuration

//chip select
#define SD_CSn_PxOUT    P1OUT//1.4
#define SD_CSn_PxDIR    P1DIR
#define SD_CSn_PIN      0x10

#define CS_LOW()        SD_CSn_PxOUT &= ~SD_CSn_PIN           // Card Select
#define CS_HIGH()       SD_CSn_PxOUT |= SD_CSn_PIN            // Card
Deselect

#define DUMMY 0xff

```

```

// Tokens (necessary because at NPO/IDLE (and CS active) only 0xff is on the
// data/command line)
#define MMC_START_DATA_BLOCK_TOKEN          0xfe    // Data token start byte,
Start Single Block Read
#define MMC_START_DATA_MULTIPLE_BLOCK_READ  0xfe    // Data token start byte,
Start Multiple Block Read
#define MMC_START_DATA_BLOCK_WRITE          0xfe    // Data token start byte,
Start Single Block Write
#define MMC_START_DATA_MULTIPLE_BLOCK_WRITE 0xfc    // Data token start byte,
Start Multiple Block Write
#define MMC_STOP_DATA_MULTIPLE_BLOCK_WRITE  0xfd    // Data token stop byte,
Stop Multiple Block Write

// an affirmative R1 response (no errors)
#define MMC_R1_RESPONSE                     0x00

// this variable will be used to track the current block length
// this allows the block length to be set only when needed
// unsigned long _BlockLength = 0;

// error/success codes
#define MMC_SUCCESS                         0x00
#define MMC_BLOCK_SET_ERROR                 0x01
#define MMC_RESPONSE_ERROR                  0x02
#define MMC_DATA_TOKEN_ERROR                0x03
#define MMC_INIT_ERROR                      0x04
#define MMC_CRC_ERROR                       0x10
#define MMC_WRITE_ERROR                     0x11
#define MMC_OTHER_ERROR                     0x12
#define MMC_TIMEOUT_ERROR                   0xFF

// commands: first bit 0 (start bit), second 1 (transmission bit); CMD-number
+ Offset 0x40
#define MMC_GO_IDLE_STATE                   0x40    //CMD0
#define MMC_SEND_OP_COND                    0x41    //CMD1
#define MMC_READ_CSD                        0x49    //CMD9
#define MMC_SEND_CID                        0x4a    //CMD10
#define MMC_STOP_TRANSMISSION               0x4c    //CMD12
#define MMC_SEND_STATUS                     0x4d    //CMD13
#define MMC_SET_BLOCKLEN                    0x50    //CMD16
#define MMC_READ_SINGLE_BLOCK                0x51    //CMD17 Read block from memory
#define MMC_READ_MULTIPLE_BLOCK              0x52    //CMD18
#define MMC_CMD_WRITE_BLOCK                  0x54    //CMD20 Write block to memory
#define MMC_WRITE_BLOCK                      0x58    //CMD24
#define MMC_WRITE_MULTIPLE_BLOCK             0x59    //CMD25
#define MMC_WRITE_CSD                       0x5b    //CMD27 PROGRAM_CSD
#define MMC_SET_WRITE_PROT                  0x5c    //CMD28
#define MMC_CLR_WRITE_PROT                  0x5d    //CMD29
#define MMC_SEND_WRITE_PROT                 0x5e    //CMD30
#define MMC_TAG_SECTOR_START                 0x60    //CMD32
#define MMC_TAG_SECTOR_END                  0x61    //CMD33
#define MMC_UNTAG_SECTOR                     0x62    //CMD34

```

```

#define MMC_TAG_ERASE_GROUP_START 0x63    //CMD35
#define MMC_TAG_ERASE_GROUP_END   0x64    //CMD36
#define MMC_UNTAG_ERASE_GROUP     0x65    //CMD37
#define MMC_ERASE                  0x66    //CMD38
#define MMC_READ_OCR               0x67    //CMD39
#define MMC_CRC_ON_OFF            0x68    //CMD40

char mmcMountBlock(unsigned long address);
char mmcReadBytes(const unsigned char count, unsigned char *pBuffer);
char mmcReadByte(void);
void mmcUnmountBlock(void);

// mmc init
char initMMC (void);

// check if MMC card is present
char mmc_ping(void);

// send command to MMC
void mmcSendCmd (const char cmd, unsigned long data, const char crc);

// set MMC in Idle mode
char mmc_GoIdle();

// set MMC block length of count=2^n Byte
char mmcSetBlockLength (const unsigned long);

// read a size Byte big block beginning at the address.
char mmcReadBlock(const unsigned long address, const unsigned long count,
unsigned char *pBuffer);
#define mmcReadSector(sector, pBuffer) mmcReadBlock(sector*512ul, 512,
pBuffer)

// write a 512 Byte big block beginning at the (aligned) address
char mmcWriteBlock (const unsigned long address, const unsigned long count);
#define mmcWriteSector(sector, pBuffer) mmcWriteBlock(sector*512ul, 512,
pBuffer)

// Read Register arg1 with Length arg2 (into the buffer)
char mmcReadRegister(const char, const unsigned char, unsigned char
*pBuffer);

// Read the Card Size from the CSD Register
unsigned long MMC_ReadCardSize(void);

void MMC_initSPI (void);

```

mmc.c

```

// *****
// File: mmc.c
// Description: Library to access a MultiMediaCard
//              functions: init, read, write ...
// C. Speck / S. Schauer
// Texas Instruments, Inc

```

```

// June 2005
//
// Version 1.1
// corrected comments about connection the MMC to the MSP430
// increased timeout in mmcGetXXResponse
//
// Modified by
// Christopher Johnson and Trieu Nguyen
//
// *****
// MMC Lib
// *****
#include <msp430x20x2.h>
#include "mmc.h"

// Function Prototypes
char mmcGetResponse(void);
char mmcGetXXResponse(const char resp);
char mmcCheckBusy(void);
unsigned char spiSendByte(const unsigned char data);
char mmc_GoIdle();

//-----
unsigned char spiSendByte(const unsigned char data)
{
    while (!(USICTL1&USIIFG));           // Wait for RX to finish
    USISR=data;
    USICNT = 8;                          // send 8 bits of data
    while (!(USICTL1&USIIFG));           // Wait for RX to finish
    return (USISRL);                     // Return a response
}

unsigned char spiSendByte2(const unsigned short data)
{
    while (!(USICTL1&USIIFG));           // Wait for RX to finish
    USISR=data; //data;
    USISRL = USISRH;
    USICNT = 8; //changed this
    while (!(USICTL1&USIIFG));           // Wait for RX to finish
    return (USISRL);                     // Return a response
}

unsigned char spiSendByte3(const unsigned short data)
{
    while (!(USICTL1&USIIFG));           // Wait for RX to finish
    USISR=data; //data;
    USISRL = USISRL;
    USICNT = 8; //changed this
    while (!(USICTL1&USIIFG));           // Wait for RX to finish
    return (USISRL);                     // Return a response
}

void MMC_initSPI(void)
{
    // Define chip select pin
    SD_CS_n_PxDIR|=SD_CS_n_PIN;
    SD_CS_n_PxOUT|=SD_CS_n_PIN;
}

```

```

    // Enable 16 bit register
    USICNT |= USI16B
    // Enable SDI, SDO, SCLK, SPI master mode, data output enable
    USICTL0 |= USIPE7 + USIPE6 + USIPE5 + USIMST + USIOE;
    // Divide clock by 1 (clock divider), use SMCLK, default clock high
    USICKCTL = USIDIV_0 + USISSEL_2 + USICKPL;
    // USI released for operation
    USICTL0 &= ~USISWRST;
}

// Initialize MMC card
// set DI and CS high and apply more than 74 pulses to SCLK
char initMMC (void)
{
    int i;

    CS_HIGH(); //set CS high
    for(i=0;i<11;i++)
        spiSendByte(0xff); //set DI high with ff (10 times)

    return (mmc_GoIdle());
}

char mmc_GoIdle()
{
    char response=0x01;
    CS_LOW();

    //Send Command 0 to put MMC in SPI mode
    mmcSendCmd(MMC_GO_IDLE_STATE,0,0x95);
    //Now wait for READY RESPONSE
    if(mmcGetResponse()!=0x01)
        return MMC_INIT_ERROR;

    while(response==0x01)
    {
        CS_HIGH();
        spiSendByte(0xff);
        CS_LOW();
        mmcSendCmd(MMC_SEND_OP_COND,0x00,0xff);
        response=mmcGetResponse();
    }
    CS_HIGH();
    spiSendByte(0xff);
    return (MMC_SUCCESS);
}

// mmc Get Response
char mmcGetResponse(void)
{
    //Response comes 1-8bytes after command
    //the first bit will be a 0
    //followed by an error code
    //data will be 0xff until response
    int i=0;

```

```

    char response;

    while(i<=64)
    {
        response=spiSendByte(0xff);
        if(response==0x00)break;
        if(response==0x01)break;
        i++;
    }
    return response;
}

char mmcGetXXResponse(const char resp)
{
    //Response comes 1-8bytes after command
    //the first bit will be a 0
    //followed by an error code
    //data will be 0xff until response
    int i=0;

    char response;

    while(i<=1000)
    {
        response=spiSendByte(0xff);
        if(response==resp)break;
        i++;
    }
    return response;
}

char mmcCheckBusy(void)
{
    //Response comes 1-8bytes after command
    //the first bit will be a 0
    //followed by an error code
    //data will be 0xff until response
    int i=0;

    char response;
    char rvalue;
    while(i<=64)
    {
        response=spiSendByte(0xff);
        response &= 0x1f;
        switch(response)
        {
            case 0x05: rvalue=MMC_SUCCESS;break;
            case 0x0b: return(MMC_CRC_ERROR);
            case 0x0d: return(MMC_WRITE_ERROR);
            default:
                rvalue = MMC_OTHER_ERROR;
                break;
        }
        if(rvalue==MMC_SUCCESS)break;
        i++;
    }
}

```

```

    i=0;
    do
    {
        response=spiSendByte(0xff);
        i++;
    }
    while(response==0);
    return response;
}

// The card will respond with a standard response token followed by a data
// block suffixed with a 16 bit CRC.

char mmcReadBlock(const unsigned long address, const unsigned long count,
unsigned char *pBuffer)
{
    unsigned long i = 0;
    char rvalue = MMC_RESPONSE_ERROR;

    // Set the block length to read
    // block length could be set
    if (mmcSetBlockLength (count) == MMC_SUCCESS)
    {
        // SS = LOW (on)
        CS_LOW ();
        // send read command MMC_READ_SINGLE_BLOCK=CMD17
        mmcSendCmd (MMC_READ_SINGLE_BLOCK,address, 0xFF);
        // Send 8 Clock pulses of delay, check if the MMC acknowledged
        //the read block command
        // it will do this by sending an affirmative response
        // in the R1 format (0x00 is no errors)
        if (mmcGetResponse() == 0x00)
        {
            // now look for the data token to signify the start of
            // the data
            if (mmcGetXXResponse(MMC_START_DATA_BLOCK_TOKEN) ==
MMC_START_DATA_BLOCK_TOKEN)
            {
                // clock the actual data transfer and receive the
                //bytes; spi_read automatically finds the Data Block
                for (i = 0; i < count; i++)
                {
                    // is executed with card inserted
                    pBuffer[i] = spiSendByte(0xff);

                    // get CRC bytes (not really needed by us, but
                    //required by MMC)
                    spiSendByte(0xff);
                    spiSendByte(0xff);
                    rvalue = MMC_SUCCESS;
                }
            }
            else
            {
                // the data token was never received
                rvalue = MMC_DATA_TOKEN_ERROR;        // 3
            }
        }
        else

```

```

        {
            // the MMC never acknowledge the read command
            rvalue = MMC_RESPONSE_ERROR;          // 2
        }
    }
else
{
    rvalue = MMC_BLOCK_SET_ERROR;                // 1
}
CS_HIGH ();
spiSendByte(0xff);
return rvalue;
} // mmc_read_block

void mmcUnmountBlock(void)
{
    // get CRC bytes (not really needed by us, but required by MMC)
    spiSendByte(0xff);
    spiSendByte(0xff);
    CS_HIGH ();
    spiSendByte(0xff);
}

char mmcMountBlock(unsigned long address)
{
    char rvalue = MMC_RESPONSE_ERROR;

    // Set the block length to read
    if (mmcSetBlockLength (512) == MMC_SUCCESS)    // block length could be
set
    {
        // SS = LOW (on)
        CS_LOW ();
        // send read command MMC_READ_SINGLE_BLOCK=CMD17
        mmcSendCmd (MMC_READ_SINGLE_BLOCK,address, 0xFF);
        // Send 8 Clock pulses of delay, check if the MMC acknowledged
        //the read block command
        // it will do this by sending an affirmative response
        // in the R1 format (0x00 is no errors)
        if (mmcGetResponse() == 0x00)
        {
            // now look for the data token to signify the start of
            // the data
            if (mmcGetXXResponse(MMC_START_DATA_BLOCK_TOKEN) ==
MMC_START_DATA_BLOCK_TOKEN)
            {
                //success, data ready to read
                rvalue = MMC_SUCCESS;
            }
            else
            {
                // the data token was never received
                rvalue = MMC_DATA_TOKEN_ERROR;    // 3
                CS_HIGH ();
                spiSendByte(0xff);
            }
        }
    }
}

```

```

    }
    else
    {
        // the MMC never acknowledge the read command
        rvalue = MMC_RESPONSE_ERROR;          // 2
        CS_HIGH ();
        spiSendByte(0xff);
    }
}
else
{
    rvalue = MMC_BLOCK_SET_ERROR;             // 1
    CS_HIGH ();
    spiSendByte(0xff);
}
return rvalue;
} // mmc_read_block

//-----
//char mmcWriteBlock (const unsigned long address)
char mmcWriteBlock (const unsigned long address, const unsigned long count)
{
    unsigned short sample = 0;
    unsigned long i = 0;
    unsigned long j = 0;
    char rvalue = MMC_RESPONSE_ERROR;         // MMC_SUCCESS;

    CS_LOW ();
    // send write command
    mmcSendCmd (MMC_WRITE_BLOCK,address, 0xFF);

    // check if the MMC acknowledged the write block command
    // it will do this by sending an affirmative response
    // in the R1 format (0x00 is no errors)

    if (mmcGetXXResponse(MMC_R1_RESPONSE) == MMC_R1_RESPONSE)
    {
        spiSendByte(0xff);
        // send the data token to signify the start of the data
        spiSendByte(0xfe); //fc for continuous write
        // clock the actual data transfer and transmitt the bytes (One
        //full sector)
        for (i = 0; i < count; i++)
        {
            // Sampling and conversion start :F2012
            ADC10CTL0 |= ENC + ADC10SC;
            //LPMO, ADC10_ISR will force exit
            __bis_SR_register(CPUOFF + GIE);
            //Need to save this sample inside of a write function
            sample = ADC10MEM
            //Send 8 most significant 2's compliment bits
            spiSendByte2(sample);
            if (i < count-1)
            {
                // Between sample delay providing 5200 Hz sampling
                for (j = 0; j < 275; j++){

```

```

    }
    // put CRC bytes (not really needed by us, but required by MMC)
    spiSendByte(0xff);
    spiSendByte(0xff);
    // read the data response xxx0<status>1 : status 010: Data
    // accepted, status 101: Data
    // rejected due to a crc error, status 110: Data rejected due to
    //a Write error.
    mmcCheckBusy();
    rvalue = MMC_SUCCESS;
}
else
{
    // the MMC never acknowledge the write command
    rvalue = MMC_RESPONSE_ERROR;    // 2
}

CS_HIGH ();
// Send 8 Clock pulses of delay.
spiSendByte(0xff);
return rvalue;
} // mmc_write_block

```

```

//-----
void mmcSendCmd (const char cmd, unsigned long data, const char crc)
{
    char frame[6];
    char temp;
    int i;
    frame[0]=(cmd|0x40);
    for(i=3;i>=0;i--)
    {
        temp=(char) (data>>(8*i));
        frame[4-i]=(temp);
    }
    frame[5]=(crc);
    for(i=0;i<6;i++)
        spiSendByte(frame[i]);
}

```

```

//----- set blocklength 2^n -----
char mmcSetBlockLength (const unsigned long blocklength)
{
    // char rValue = MMC_TIMEOUT_ERROR;
    // char i = 0;
    // SS = LOW (on)
    CS_LOW ();
    // Set the block length to read
    // MMC_SET_BLOCKLEN =CMD16
    mmcSendCmd(MMC_SET_BLOCKLEN, blocklength, 0xFF);

    // get response from MMC - make sure that its 0x00 (R1 ok response
    // format)
    if(mmcGetResponse()!=0x00)

```

```

    {
        initMMC();
        mmcSendCmd(MMC_SET_BLOCKLEN, blocklength, 0xFF);
        mmcGetResponse();
    }

    CS_HIGH ();

    // Send 8 Clock pulses of delay.
    spiSendByte(0xff);

    return MMC_SUCCESS;
} // Set block_length

// Reading the contents of the CSD and CID registers in SPI mode is a simple
// read-block transaction.
char mmcReadRegister (const char cmd_register, const unsigned char length,
unsigned char *pBuffer)
{
    char uc = 0;
    char rvalue = MMC_TIMEOUT_ERROR;

    if (mmcSetBlockLength (1024) == MMC_SUCCESS)
    {
        CS_LOW ();
        // CRC not used: 0xff as last byte
        mmcSendCmd(cmd_register, 0x000000, 0xff);

        // wait for response
        // in the R1 format (0x00 is no errors)
        if (mmcGetResponse() == 0x00)
        {
            if (mmcGetXXResponse(0xfe) == 0xfe)
                for (uc = 0; uc < length; uc++)
                    pBuffer[uc] = spiSendByte(0xff);
            //mmc_buffer[uc] = spiSendByte(0xff);
            // get CRC bytes (not really needed by us, but required by MMC)
            spiSendByte(0xff);
            spiSendByte(0xff);
            rvalue = MMC_SUCCESS;
        }
        else
            rvalue = MMC_RESPONSE_ERROR;
        // CS = HIGH (off)
        CS_HIGH ();

        // Send 8 Clock pulses of delay.
        spiSendByte(0xff);
    }
    CS_HIGH ();
    return rvalue;
} // mmc_read_register

#include "math.h"
unsigned long MMC_ReadCardSize(void)

```

```

{
    // Read contents of Card Specific Data (CSD)

    unsigned long MMC_CardSize;
    unsigned short i,          // index
                  j,          // index
                  b,          // temporary variable
                  response,    // MMC response to command
                  mmc_C_SIZE;

    unsigned char mmc_READ_BL_LEN, // Read block length
                  mmc_C_SIZE_MULT;

    CS_LOW ();

    spiSendByte(MMC_READ_CSD); // CMD 9
    for(i=4; i>0; i--)         // Send four dummy bytes
        spiSendByte(0);
    spiSendByte(0xFF); // Send CRC byte

    response = mmcGetResponse();

    // data transmission always starts with 0xFE
    b = spiSendByte(0xFF);

    if( !response )
    {
        while (b != 0xFE) b = spiSendByte(0xFF);
        // bits 127:87
        // Host must keep the clock running for at
        for(j=5; j>0; j--)
            b = spiSendByte(0xff);

        // 4 bits of READ_BL_LEN
        // bits 84:80
        b = spiSendByte(0xff); // lower 4 bits of CCC and
        mmc_READ_BL_LEN = b & 0x0F;

        b = spiSendByte(0xff);

        // bits 73:62 C_Size
        // xxCC CCCC CCCC CC
        mmc_C_SIZE = (b & 0x03) << 10;
        b = spiSendByte(0xff);
        mmc_C_SIZE += b << 2;
        b = spiSendByte(0xff);
        mmc_C_SIZE += b >> 6;

        // bits 55:53
        b = spiSendByte(0xff);

        // bits 49:47
        mmc_C_SIZE_MULT = (b & 0x03) << 1;
        b = spiSendByte(0xff);
        mmc_C_SIZE_MULT += b >> 7;
    }
}

```

```

        // bits 41:37
        b = spiSendByte(0xff);

        b = spiSendByte(0xff);

        b = spiSendByte(0xff);

        b = spiSendByte(0xff);

        b = spiSendByte(0xff);
    }

    for(j=4; j>0; j--)          // Host must keep the clock running for at
        b = spiSendByte(0xff); // least Ncr (max = 4 bytes) cycles after
    // the card response is received
    b = spiSendByte(0xff);
    CS_LOW ();

    MMC_CardSize = (mmc_C_SIZE + 1);
    // power function with base 2 is better with a loop
    // i = (pow(2,mmc_C_SIZE_MULT+2)+0.5);
    for(i = 2, j=mmc_C_SIZE_MULT+2; j>1; j--)
        i <<= 1;
    MMC_CardSize *= i;
    // power function with base 2 is better with a loop
    // i = (pow(2,mmc_READ_BL_LEN)+0.5);
    for(i = 2, j=mmc_READ_BL_LEN; j>1; j--)
        i <<= 1;
    MMC_CardSize *= i;

    return (MMC_CardSize);
}

char mmc_ping(void)
{
    // if ((SD_CD_PxIN & SD_CD_PIN))
    return (MMC_SUCCESS);
    // else
    // return (MMC_INIT_ERROR);
}

#pragma vector=ADC10_VECTOR
__interrupt void ADC10_ISR(void)
{
    __bic_SR_register_on_exit(CPUOFF);
}

```

main.cpp

```

//Main driver for GUI.  Initiates the application.
#include <iostream>
#include <FL/Fl.H>
#include "director.h"

```

```

int main()
{
    Director app;
    return Fl::run();
}

```

button.h

```

//Interface for button object wrapper
#ifndef _BUTTON_H_
#define _BUTTON_H_

#include <FL/Fl_Button.H>
#include <vector>

class Listener;

class Button : public Fl_Button
{
public:
    Button(int x, int y, int width, int height, const char* label = 0);
    virtual ~Button();
    virtual void Add_listener(Listener* listener);
    virtual void Notify();

private:
    Button(const Button& src);
    const Button& operator=(const Button& src);
    std::vector<Listener*> listeners;
};

#endif //ndef __BUTTON_H_

```

button.cpp

```

//Implementation for button object wrapper
#include "button.h"
#include "listener.h"

using namespace std;

void Button_callback(Fl_Widget* widget)
{
    Button* button = dynamic_cast<Button*>(widget);
    if (button != NULL)
        button->Notify();
}

Button::Button(int x, int y, int width, int height, const char *label)
    : Fl_Button(x, y, width, height, label)
{
}

```

```

        callback(Button_callback);
    }

Button::~~Button()
{
}

void Button::Add_listener(Listener* listener)
{
    listeners.push_back(listener);
}

void Button::Notify()
{
    for (vector<Listener*>::iterator it = listeners.begin();
         it != listeners.end(); it++)
    {
        (*it)->Widget_changed(this);
    }
}

```

chooserbox.h

```

//Interface for selection box object wrapper
#ifndef _CHOOSERBOX_H_
#define _CHOOSERBOX_H_

#include <FL/Fl_Select_Browser.H>
#include <vector>

class Listener;

class ChooserBox : public Fl_Select_Browser
{
public:
    ChooserBox(int x, int y, int width, int height, const char* label = 0);
    virtual ~ChooserBox();
    virtual void Add_listener(Listener* listener);
    virtual void Notify();

private:
    ChooserBox(const ChooserBox& src);
    const ChooserBox& operator=(const ChooserBox& src);
    std::vector<Listener*> listeners;
};

#endif //ndef __ChooserBox_H_

```

chooserbox.cpp

```

//Implementation for selection box object wrapper
#include "chooserbox.h"
#include "listener.h"

```

```

using namespace std;

void ChooserBox_callback(Fl_Widget* widget)
{
    ChooserBox* box = dynamic_cast<ChooserBox*>(widget);
    if (box != NULL)
        box->Notify();
}

ChooserBox::ChooserBox(int x, int y, int width, int height, const char
*label): Fl_Select_Browser(x, y, width, height, label)
{
    callback(ChooserBox_callback);
}

ChooserBox::~ChooserBox()
{
}

void ChooserBox::Add_listener(Listener* listener)
{
    listeners.push_back(listener);
}

void ChooserBox::Notify()
{
    for (vector<Listener*>::iterator it = listeners.begin();
        it != listeners.end(); it++)
    {
        (*it)->Widget_changed(this);
    }
}

```

director.h

```

//Interface file for a "manager" type object which handles all user
//interaction with the SAiNT Interface
#ifndef _DIRECTOR_H_
#define _DIRECTOR_H_

#include "listener.h"
#include <string>

//Audio Engine overhead
#include <irrKlang.h>

#ifdef WIN32
#include <conio.h>
#else
#include "../common/conio.h"
#endif

using namespace irrklang;

#pragma comment(lib, "irrKlang.lib") // link with irrKlang.dll

```

```

class Easy_Browser;
class Fl_Widget;
class Fl_Window;
class Menu_Bar;
class Text_Box;
class Text_Box_Nonrealtime;
class Button;
class Fl_Value_Slider;
class Slider;
class Timer;
class ChooserBox;
class Fl_Output;
class Fl_Multiline_Output;

class Director : public Listener
{

public:

    Director();
    virtual ~Director();
    virtual void Widget_changed(Fl_Widget* widget);

protected:

    Fl_Window* window;
    Fl_Window* recordWindow;
    Menu_Bar* menu;
    int recordMode, loadnotes, loadaudio, quit, context;
    Fl_Multiline_Output* output_box;
    Text_Box* input_box;
    Easy_Browser* chooser_box;
    Easy_Browser* status_box;

    Fl_Output* notes_box;
    Fl_Output* audio_box;

    char* unformattedNotes;

    Button * leftarrow;
    Button * rightarrow;
    Button * doubleleftarrow;
    Button * doublerightarrow;

    Button * play;
    Button * pause;
    Fl_Value_Slider* slider;
    Fl_Value_Slider* volumeSlider;
    ChooserBox* resultBox;
    ISoundEngine* engine;
    ISound* CurrentPlayingSound;
    IAudioRecorder* recorder;

    //Timer
    Timer *timer;

    //Recorder Window

```

```

        Button * startRecorder;
        Button * stopRecorder;

        Easy_Browser* noteBox;
        Text_Box* noteTakerBox;

        int notePosition;
        int spacePosition;
        int startPosition;
        int endPosition;
        int noteSize;

private:

        Director(const Director& src);
        const Director& operator=(const Director& src);
};

#endif //ndef _DIRECTOR_H_

```

director.cpp

```

//Implementation for director object
#include "director.h"
#include <algorithm>
#include <FL/Fl_Window.H>
#include <FL/Fl_File_Chooser.H>
#include <FL/Fl_Value_Slider.H>
#include <FL/Fl_Select_Browser.H>
#include <FL/Fl_Multiline_Output.H>
#include <FL/Fl_Output.H>
#include "menu_bar.h"
#include "chooserbox.h"
#include "easy_browser.h"
#include "text_box.h"
#include "text_box_nonrealtime.h"
#include "button.h"
#include <iostream>
#include <fstream>
#include <vector>
#include <map>
#include <sstream>
#include <iostream>
#include "timer.h"
#include <time.h>
using namespace std;

//Slider
void UpdateTimerCallback(void*);
void OnSliderMoved(Fl_Value_Slider*, void*);
void OnVolumeSliderMoved(Fl_Value_Slider*, void*);
Fl_Value_Slider * currentSlider;
Fl_Value_Slider * currentVolumeSlider;

ISound* CurrentPlayingSound2;
double currentVolume = 1;

```

```

int resultCount;
string originalNotes;

vector<string> searchResults;
vector<double> searchTimes;
vector<double> timeStamps;
vector<string> textFile;
vector<string> formattedText;
bool textLoaded;
map<double,string> final;
ostringstream oss;
ostringstream oss2;

//Function declarations
void writeWaveFile(const char*, SAudioStreamFormat, void*);
void searchNotes(string);
void searchLoadedNotes(string);
void multiWordSearch(string);
string StringToLower(string);
string StringToUpper(string);
void Tokenize(string ,vector<string>& );

namespace
{
    const int WINDOW_WIDTH = 800;
    const int WINDOW_HEIGHT = 600;
    const int PADDING = 5;
    const int BAR_HEIGHT = 30;
    const int BUTTON_WIDTH = 50;
    const double updateTimeSpeed = 1.0 / 20; // x times a second
    const double timeDisplayFactor = 1 / 1000.0;
}

/*-----
Name:   Director

Purpose:  This handles the create of all GUI elements.
-----*/
Director::Director()
{
    //Initilize sound engine
    engine = createIrrKlangDevice();
    recorder = createIrrKlangAudioRecorder(engine);

    if (!engine|| !recorder)
    {
        cout << "The sound engine isn't working" << endl;
    }

    // Make the main window
    window = new Fl_Window(WINDOW_WIDTH +300, WINDOW_HEIGHT, "SAiNT");

    // Create the menu bar
    menu = new Menu_Bar(PADDING, PADDING, WINDOW_WIDTH - (2 * PADDING),
        BAR_HEIGHT);

    //Add Menu Items

```

```

loadnotes = menu->add("File/Load Notes", "^n" , 0, 0);
loadaudio = menu->add("File/Load Audio", "^a" , 0, 0);
recordMode = menu->add("File/Record Mode", "^n" , 0, 0);
quit = menu->add("File/Quit", "^q" , 0, 0);

menu->Add_listener(this);

notes_box = new Fl_Output(PADDING*15, (3 * PADDING) + BAR_HEIGHT,
WINDOW_WIDTH - (16*PADDING) , BAR_HEIGHT, "Notes File:");
audio_box = new Fl_Output(PADDING*15, (4 * PADDING) + BAR_HEIGHT*2,
WINDOW_WIDTH - (16*PADDING) , BAR_HEIGHT, "Audio File:");
notes_box->deactivate();
audio_box->deactivate();
notes_box -> color(FL_GRAY);
audio_box -> color(FL_GRAY);

//Make the result selection box
resultBox = new ChooserBox(805, 45, 290, 550, "Search Results");
resultBox->Add_listener(this);
resultBox->align(FL_ALIGN_TOP);
resultBox->labelsize(24);

// Make the search box
input_box = new Text_Box(60, (6 * PADDING) + BAR_HEIGHT* 3,
WINDOW_WIDTH - (2 * PADDING) - 240 , BAR_HEIGHT, "Search:");
input_box->Add_listener(this);
input_box->deactivate();

//Make the output box
output_box = new Fl_Multiline_Output(PADDING, (7 * PADDING) + (4 *
BAR_HEIGHT), WINDOW_WIDTH - (2 * PADDING), WINDOW_HEIGHT - (5 *
PADDING) - (9 * BAR_HEIGHT));
output_box->textsize(16);
output_box -> type(4);
output_box -> wrap(1);

// Make the results status box
status_box = new Easy_Browser(PADDING, (4 * PADDING) + (2 * BAR_HEIGHT)
+ WINDOW_HEIGHT - (5 * PADDING) - (5 * BAR_HEIGHT) , WINDOW_WIDTH - (2
* PADDING), BAR_HEIGHT );

//Audio player controls
play = new Button( PADDING, (4 * PADDING) + (2 * BAR_HEIGHT) +
WINDOW_HEIGHT - (5 * PADDING) - (3 * BAR_HEIGHT) , BUTTON_WIDTH,
BAR_HEIGHT, "Play");
play -> Add_listener(this);
play -> deactivate();
pause = new Button( 2* PADDING + BUTTON_WIDTH, (4 * PADDING) + (2 *
BAR_HEIGHT) + WINDOW_HEIGHT - (5 * PADDING) - (3 * BAR_HEIGHT) ,
BUTTON_WIDTH, BAR_HEIGHT, "Pause");
pause -> Add_listener(this);
pause -> deactivate();

//Slider
slider = new Fl_Value_Slider( 3* PADDING + 3* BUTTON_WIDTH, (4 *
PADDING) + (2 * BAR_HEIGHT) + WINDOW_HEIGHT - (5 * PADDING) - (3 *
BAR_HEIGHT) , BUTTON_WIDTH*8, BAR_HEIGHT, "Position");

```

```

slider-> callback((Fl_Callback*)OnSliderMoved);
slider -> type(1);
slider->labelsize(11);
slider->minimum(0);
slider->textsize(11);
slider->align(FL_ALIGN_LEFT);
slider->type(FL_HOR_NICE_SLIDER);
slider -> deactivate();
currentSlider = slider;

//Volume slider
volumeSlider = new Fl_Value_Slider( 9* PADDING + 12* BUTTON_WIDTH, (4 *
PADDING) + (2 * BAR_HEIGHT) + WINDOW_HEIGHT - (5 * PADDING) - (3 *
BAR_HEIGHT) , BUTTON_WIDTH*3, BAR_HEIGHT, "Volume");
volumeSlider -> callback((Fl_Callback*)OnVolumeSliderMoved);
volumeSlider -> type(1);
volumeSlider->labelsize(11);
volumeSlider->minimum(0);
volumeSlider->textsize(11);
volumeSlider->align(FL_ALIGN_LEFT);
volumeSlider->type(FL_HOR_NICE_SLIDER);
volumeSlider -> deactivate();
currentVolumeSlider = volumeSlider;

window->end();
window->show();

}

/*-----
Name:   ~Director

Purpose:   Destructor for Director class
-----*/
Director::~~Director()
{
}

/*-----
Name:   ~Widget_changed

Purpose:   This function is called when the user interacts with a GUI
element
-----*/

void Director::Widget_changed(Fl_Widget* widget)
{
    if (widget == menu)
    {
        //If Load Notes was selected
        if (menu -> value() == 1)
        {
            textLoaded = true;
            timeStamps.clear();
            searchTimes.clear();
            searchResults.clear();
        }
    }
}

```

```

textFile.clear();
formattedText.clear();

Fl_File_Chooser file_chooser(".", "Text Files (*.{txt})",
Fl_File_Chooser::SINGLE , "Open File");
file_chooser.show();

while (file_chooser.shown())
{
    Fl::wait();
}

if(file_chooser.count() == 1)
{
    //Update file output box
    notes_box->value(file_chooser.value());
    notes_box->activate();

    //Enable the search box
    input_box -> activate();

    //Read in the file input and store it to a string
    string line;
    string result;
    ifstream myfile (file_chooser.value());

    //While the file is open
    if (myfile.is_open())
    {
        while (! myfile.eof() )
        {
            getline (myfile,line);
            char* test = strdup(line.c_str());
            string temp;
            int count = 0;
            double totalTime;

            if(test[count] == '[' )
            {
                count++;
                //While the timestamp is in the
                date
                while(test[count] != ' ')
                {
                    count++;
                }

                //Increment 1 character past the
                whitespace
                count++;

                while(test[count] != 'P' && test[count] != 'A')
                {
                    temp = temp + test[count];
                    count++;
                }
            }
        }
    }
}

```

```

    int count2 = 0;
    int amount = 0;
    string hours;
    while(temp[count2] != ':')
    {
        hours = hours + temp[count2];
        count2++;
    }
    count2++;

    double convertedHours = atoi(hours.c_str());
    totalTime = convertedHours * 60 * 60;

    string minutes;
    while(temp[count2] != ':')
    {
        minutes = minutes + temp[count2];
        count2++;
    }
    count2++;

    double convertedMinutes = atoi(minutes.c_str());
    totalTime = totalTime + (convertedMinutes * 60);

    string seconds;
    for(int i = 0; i < 2; i++)
    {
        seconds = seconds + temp[count2];
        count2++;
    }

    double convertedSeconds = atoi(seconds.c_str());
    totalTime = totalTime + convertedSeconds;
    count = count+3;
    string finalText;
    string word = "";
    for(int i = count; i < line.size(); i++)
    {
        finalText = finalText + line[i];
        if(line[i] == ' ')
        {
            //Push each individual word into a vector
            //and the corresponding timestamp

            formattedText.push_back(StringToLower(word));
            textFile.push_back(word);
            timeStamps.push_back(totalTime);
            word = "";
        }
        else
        {
            word = word + line[i];
        }
    }

    //Push the last word into the vector

```

```

formattedText.push_back(StringToLower(word));
textFile.push_back(word);
timeStamps.push_back(totalTime);
result = result + finalText + '\n';

    }
    myfile.close();
    output_box->value(result.c_str());

}
else
{
    cout << "Unable to open file";
}

//Fix the times and reference the first time to 0 seconds
double referenceTime = timeStamps[0];
for(int i = 0; i < timeStamps.size(); i++)
{
    timeStamps[i] = timeStamps[i] - referenceTime;
}

}

}

//If Load Audio was selected
if (menu -> value() == 2)
{

    Fl_File_Chooser file_chooser(".", "MP3 Files (*.mp3,wav)",
    Fl_File_Chooser::SINGLE , "Open File");
    file_chooser.show();

    while (file_chooser.shown())
    {
        Fl::wait();
    }

    if(file_chooser.count() == 1)
    {

        //Stop all sounds first before loading a new one
        if (CurrentPlayingSound2)
        {
            CurrentPlayingSound2->drop();
        }

        engine->stopAllSounds();

        CurrentPlayingSound = engine->play2D(file_chooser.value(),
        true , true, true);

        CurrentPlayingSound2 = CurrentPlayingSound;

        //Update file output box
        audio_box->value(file_chooser.value());
    }
}

```

```

        audio_box->activate();

        //Enable controls
        slider -> bounds(0, CurrentPlayingSound->getPlayLength() *
timeDisplayFactor);
        play -> activate();
        pause -> deactivate();
        slider -> activate();
        volumeSlider->value(currentVolume);
        CurrentPlayingSound2->setVolume(currentVolume);
        volumeSlider->redraw();
        volumeSlider -> activate();

        // initialize timer
        Fl::add_timeout(updateTimerSpeed, UpdateTimerCallback);
    }

}

//If Quit was selected
if (menu -> value() == 4)
{
    window -> hide();
}

//If Audio Mode was selected
if (menu -> value() == 3)
{
    recordWindow = new Fl_Window(400, 400, "Record Lecture");

    startRecorder = new Button(35, 10, 150, 50, "Start
Recording");
    startRecorder->Add_listener(this);

    stopRecorder = new Button(215, 10, 150, 50, "Stop
Recording");
    stopRecorder->Add_listener(this);
    stopRecorder -> deactivate();

    noteTakerBox = new Text_Box(5, 65, 390, 380);
    noteTakerBox->Add_listener(this);
    noteTakerBox->deactivate();
    noteTakerBox->textsize(16);

    //This is the wordwrap type of the input widget
    noteTakerBox -> type(4);
    noteTakerBox -> wrap(1);
    recordWindow->end();
    recordWindow->show();

}

}

//If user enters Play button

```

```

if (widget == play)
{
    CurrentPlayingSound->setIsPaused(0);
    play -> deactivate();
    pause -> activate();
}

//If user enters Pause button
if (widget == pause)
{
    CurrentPlayingSound->setIsPaused(1);
    pause -> deactivate();
    play -> activate();
}

//If user enters a search
if (widget == input_box)
{
    //Clear our vectors
    searchTimes.clear();
    searchResults.clear();
    resultCount = 0;

    //Clear the results box
    resultBox -> clear();

    if(input_box -> size() > 0)
    {
        //Search for results and store the result and time in two
        //vectors that have matching indexes
        if( textLoaded == true)
        {
            searchLoadedNotes(StringToLower(input_box->value()));
        }
        else
        {
            searchNotes(StringToLower(input_box->value()));
        }

        //Update the result chooser box
        for (int i = 0; i < searchResults.size(); i++)
        {
            stringstream result;
            double time = (searchTimes[i] / 1000);
            result << searchResults[i] << " " << "(" << time <<
            "s" << ")";
            string temp = result.str();
            resultBox -> add(temp.c_str());
            resultCount++;
        }
    }

    //If user selects one of the results of the search
    if (widget == resultBox)

```

```

{
    //Check bounds
    if(resultBox -> value() > 0 && resultBox -> value() <=
resultCount)
    {

        if (CurrentPlayingSound2)
        {
            //The vector starts at 0, but the result box starts
            at one
            double pos = (searchTimes[(resultBox -> value()-1)]);

            //Update position slider and play position
            currentSlider ->value(pos);
            CurrentPlayingSound2->setPlayPosition(pos);
        }
    }

    //If user enters notes
    //Record notes in real time
    if (widget == noteTakerBox)
    {
        //This is a hack to handle backspaces
        if( noteTakerBox->size() < (notePosition + 1) )
        {
            //Update our positions
            notePosition--;
        }
        else
        {

            char* temp = strdup(noteTakerBox->value());
            unformattedNotes = temp;
            string temp2 = "";
            string temp3 = "";
            double temp4 = 0;

            //Parse by spaces and newlines
            if(temp[notePosition] == ' ' || temp[notePosition] == '/n')
            {
                temp4 = timer->getCurrent();
                timeStamps.push_back(timer->getCurrent());
                endPosition = notePosition;

                for(int i = startPosition; i < endPosition; i++)
                {
                    temp2 = temp2 + temp[i];
                }

                textFile.push_back(temp2);
                formattedText.push_back(StringToLower(temp2));
                startPosition = notePosition + 1;
            }

            notePosition++;
        }
    }
}

```

```

    }

    originalNotes = strdup(noteTakerBox->value());
}

//If user starts an audio recording
if (widget == startRecorder)
{
    //Start the timer and initilize values
    timer = new Timer();
    textLoaded = false;

    timeStamps.clear();
    searchTimes.clear();
    searchResults.clear();
    textFile.clear();
    formattedText.clear();

    notePosition = 0;
    startPosition = 0;
    endPosition = 0;
    noteSize = 0;
    noteTakerBox->activate();

    //Initilize audio engine recording
    recorder->startRecordingBufferedAudio();
    startRecorder -> deactivate();
    stopRecorder -> activate();
}

//If user stops an audio recording
if (widget == stopRecorder)
{
    //Lets loop through the vector and clean up newlines
    for(int i = 0; i < textFile.size(); i++)
    {
        textFile[i].erase(std::remove(textFile[i].begin(),
        textFile[i].end(), '\n'), textFile[i].end());
    }

    //Stop the engine
    recorder->stopRecordingAudio();
    writeWaveFile("recorded.wav", recorder->getAudioFormat(),
    recorder->getRecordedAudioData());
    recorder->addSoundSourceFromRecordedAudio("MyRecording");

    CurrentPlayingSound = engine->play2D("MyRecording", true , true,
    true);
    CurrentPlayingSound2 = CurrentPlayingSound;
    audio_box->value("MyRecording");

    //Enable controls
    currentSlider -> bounds(0, CurrentPlayingSound2->getPlayLength()
    * timeDisplayFactor);
    play -> activate();
    pause -> deactivate();
}

```

```

        slider -> activate();
        volumeSlider->value(currentVolume);
        CurrentPlayingSound2->setVolume(currentVolume);
        volumeSlider->redraw();
        volumeSlider -> activate();

        //initialize slider timer
        Fl::add_timeout(updateTimerSpeed, UpdateTimerCallback);

        stringstream str;
        input_box -> activate();

        output_box -> value(unformattedNotes);
        recordWindow -> hide();
    }

}

//This function moves the slider and keeps it in sync with the play position
void UpdateTimerCallback(void*)
{
    //Update play position
    currentSlider -> value(CurrentPlayingSound2->getPlayPosition() *
        timeDisplayFactor);

    Fl::repeat_timeout(updateTimerSpeed, UpdateTimerCallback);
}

//This function handles user interaction with the position slider
void OnSliderMoved(Fl_Value_Slider*, void*){
    if (CurrentPlayingSound2)
    {
        double pos = (currentSlider->value() / timeDisplayFactor);
        CurrentPlayingSound2->setPlayPosition(pos);
    }
}

//This function handles user interaction with the volume slider
void OnVolumeSliderMoved(Fl_Value_Slider*, void*){
    if (CurrentPlayingSound2)
    {
        double pos = currentVolumeSlider->value();
        currentVolume = pos;
        CurrentPlayingSound2->setVolume(pos);
    }
}

//Search the notes and store results and times in vectors
void searchNotes(string temp)
{
    for (int i = 0; i < formattedText.size(); i++)
    {
        const char *ptr = strstr (formattedText[i].c_str(),
            temp.c_str());

        //A possible match was found!
        if(ptr != NULL)

```

```

{

    if(formattedText.size() > 6)
    {

        if((i > 2) && (i < formattedText.size() - 3))
        {

            string resultString = textFile[i];
            string::size_type pos = 0;
            while ( (pos = resultString.find('\n', pos)) !=
string::npos )
            {
                resultString.replace( pos, 1, " " );
                pos++;
            }
            searchResults.push_back(resultString);
            searchTimes.push_back(timeStamps[i]);

        }
        else if ( i > formattedText.size() - 4)
        {
            string resultString;
            string resultString = textFile[i];
            string::size_type pos = 0;
            while ( (pos = resultString.find('\n', pos)) !=
string::npos )
            {
                resultString.replace( pos, 1, " " );
                pos++;
            }

            searchResults.push_back(resultString);
            searchTimes.push_back(timeStamps[i]);

        }

    }

    else if ( i < 3)
    {

        string resultString = textFile[i];
        string::size_type pos = 0;
        while ( (pos = resultString.find('\n', pos)) !=
string::npos )
        {
            resultString.replace( pos, 1, " " );
            pos++;
        }

        searchResults.push_back(resultString);
        searchTimes.push_back(timeStamps[i]);

    }

    else

```

```

        {
            searchResults.push_back(textFile[i]);
            searchTimes.push_back(timeStamps[i]);
        }

    }

}

//This is for when notes are loaded
void searchLoadedNotes(string temp)
{
    for (int i = 0; i < formattedText.size(); i++)
    {
        const char *ptr = strstr (formattedText[i].c_str(),
            temp.c_str());

        //A possible match was found!
        if(ptr != NULL)
        {
            searchResults.push_back(textFile[i]);
            searchTimes.push_back(timeStamps[i] * 1000);
        }
    }
}

//This function handles searches which span across words
void multiWordSearch(string temp)
{
    const char *ptr = strstr(originalNotes.c_str(), temp.c_str());

    //cout << ptr << endl;

    //Found a match
    if( ptr != NULL)
    {
        string str = strdup(ptr);
        string searchString = temp;
        int start = 0;
        string final;
        bool finished = false;

        //This is not a good idea, this may come back to haunt me later
        while(start < searchString.size()-1 || finished == false &&
            start<str.size())
        {
            if(start > searchString.size()-1)
            {
                if(str[start] != ' ')

```

```

        {
            final = final + str[start];
            start++;
        }
        else
        {
            finished = true;
        }
    }
    else
    {
        final = final + str[start];
        start++;
    }
}

cout << final << endl;

//Lets seperate using a space delimiter and put them into the
search result
vector<string> tokens;
Tokenize(final, tokens);

int foundPosition;
bool winner = true;

for (int j = 0; j < formattedText.size(); j++)
{
    const char *ptr = strstr(formattedText[j].c_str(),
tokens[0].c_str());

    //Got a match
    if(ptr != NULL)
    {
        foundPosition = j;
        int count = 0;
        for (int i = 0; i < tokens.size(); i++)
        {
            if(tokens[i] != formattedText[foundPosition])
            {
                break;
            }

            foundPosition++;
            count++;
        }

        if( count == tokens.size() )
        {
            searchResults.push_back(final);
            searchTimes.push_back(timeStamps[j]);
        }
    }
}
}

```

```

    }

}

//Output recorded audio to a wave file
void writeWaveFile(const char* filename, SAudioStreamFormat format, void*
data)
{
    if (!data)
    {
        cout << "Could not save recorded data to %s, nothing recorded\n";
        return;
    }

    FILE* file = fopen(filename, "wb");

    if (file)
    {
        // write wave header
        unsigned short formatType = 1;
        unsigned short numChannels = format.ChannelCount;
        unsigned long sampleRate = format.SampleRate;
        unsigned short bitsPerChannel = format.getSampleSize() * 8;
        unsigned short bytesPerSample = format.getFrameSize() ;
        unsigned long bytesPerSecond = format.getBytesPerSecond();
        unsigned long dataLen = format.getSampleDataSize();

        const int fmtChunkLen = 16;
        const int waveHeaderLen = 4 + 8 + fmtChunkLen + 8;

        unsigned long totalLen = waveHeaderLen + dataLen;

        fwrite("RIFF", 4, 1, file);
        fwrite(&totalLen, 4, 1, file);
        fwrite("WAVE", 4, 1, file);
        fwrite("fmt ", 4, 1, file);
        fwrite(&fmtChunkLen, 4, 1, file);
        fwrite(&formatType, 2, 1, file);
        fwrite(&numChannels, 2, 1, file);
        fwrite(&sampleRate, 4, 1, file);
        fwrite(&bytesPerSecond, 4, 1, file);
        fwrite(&bytesPerSample, 2, 1, file);
        fwrite(&bitsPerChannel, 2, 1, file);

        // write data
        fwrite("data", 4, 1, file);
        fwrite(&dataLen, 4, 1, file);
        fwrite(data, dataLen, 1, file);

        //finish

        printf("Saved audio as %s\n", filename);
        fclose(file);
    }
}

```

```

//Function to convert all letters in a string to upper case
string StringToUpper(string strToConvert)
{
    //change each element of the string to upper case
    for(unsigned int i=0;i<strToConvert.length();i++)
    {
        if(strToConvert[i] != ' ')
        {
            strToConvert[i] = toupper(strToConvert[i]);
        }
    }
    return strToConvert;//return the converted string
}

// Function to convert all letters in a string to lower case
string StringToLower(string strToConvert)
{
    //change each element of the string to lower case
    for(unsigned int i=0;i<strToConvert.length();i++)
    {
        if(strToConvert[i] != ' ')
        {
            strToConvert[i] = tolower(strToConvert[i]);
        }
    }
    return strToConvert;//return the converted string
}

//String tokenizer, right now its set to tokenize by spaces
void Tokenize(string str,vector<string>& tokens)
{
    // Skip delimiters at beginning.
    string::size_type lastPos = str.find_first_not_of(' ', 0);
    // Find first "non-delimiter".
    string::size_type pos = str.find_first_of(' ', lastPos);

    while (string::npos != pos || string::npos != lastPos)
    {
        // Found a token, add it to the vector.
        tokens.push_back(str.substr(lastPos, pos - lastPos));
        // Skip delimiters. Note the "not_of"
        lastPos = str.find_first_not_of(' ', pos);
        // Find next "non-delimiter"
        pos = str.find_first_of(' ', lastPos);
    }
}

```

easy_browser.h

```

//Interface for notes output object
#ifndef _EASY_BROWSER_H_
#define _EASY_BROWSER_H_

#include <FL/Fl_Browser.H>

```

```

#include <string>

class Easy_Browser : public Fl_Browser
{
public:
    Easy_Browser(int x, int y, int width, int height);
    virtual ~Easy_Browser();
    virtual void value(const std::string& text);

private:
    Easy_Browser(const Easy_Browser& src);
    const Easy_Browser& operator=(const Easy_Browser& src);
};

#endif //ndef _EASY_BROWSER_H_

```

easy_browser.cpp

```

//Implementation for notes output object
#include "easy_browser.h"

using namespace std;

Easy_Browser::Easy_Browser(int x, int y, int width, int height)
    : Fl_Browser(x, y, width, height)
{
}

Easy_Browser::~~Easy_Browser()
{
}

void Easy_Browser::value(const string& text)
{
    clear();
    if (text.length() == 0)
        return;
    string current_line;
    size_t line_start = 0, line_end;
    do
    {
        line_end = text.find('\n', line_start);

        current_line = text.substr(line_start, line_end - line_start);
        add(current_line.c_str());

        line_start = line_end + 1;
    } while (line_end != string::npos);
}

```

listener.h

```
//Interface for listener object. This behaves as a base class for //manager
objects to inherit.
#ifndef _LISTENER_H_
#define _LISTENER_H_

class Fl_Widget;

class Listener
{
public:
    virtual ~Listener();
    virtual void Widget_changed(Fl_Widget* widget) = 0;

protected:
    Listener();

private:
    Listener(const Listener& src);
    const Listener& operator=(const Listener& src);
};

#endif //ndef _LISTENER_H_
```

listener.cpp

```
//Implementation for listener object
#include "listener.h"

Listener::Listener()
{
}

Listener::~~Listener()
{
}
```

menu_bar.h

```
//Interface for menu bar object
#ifndef _MENU_BAR_H_
#define _MENU_BAR_H_

#include <FL/Fl_Menu_Bar.H>
#include <FL/Fl_Menu_Item.H>

#include <vector>

class Listener;

class Menu_Bar : public Fl_Menu_Bar
{
public:
```

```

    Menu_Bar(int x, int y, int width, int height);
    virtual ~Menu_Bar();
    virtual void Add_listener(Listener* listener);
    virtual void Notify();

private:
    Menu_Bar(const Menu_Bar& src);
    const Menu_Bar& operator=(const Menu_Bar& src);
    std::vector<Listener*> listeners;
};

#endif //ndef _MENU_BAR_H_

```

menu_bar.cpp

```

//Implementation for menu bar object
#include "menu_bar.h"
#include "listener.h"

using namespace std;

void Menu_callback(Fl_Widget* widget)
{
    Menu_Bar* menu = dynamic_cast<Menu_Bar*>(widget);
    if (menu != NULL)
        menu->Notify();
}

Menu_Bar::Menu_Bar(int x, int y, int width, int height)
    : Fl_Menu_Bar(x, y, width, height)
{
    callback(Menu_callback);
}

Menu_Bar::~~Menu_Bar()
{
}

void Menu_Bar::Add_listener(Listener* listener)
{
    listeners.push_back(listener);
}

void Menu_Bar::Notify()
{
    for (vector<Listener*>::iterator it = listeners.begin();
         it != listeners.end(); it++)
    {
        (*it)->Widget_changed(this);
    }
}

```

slider.h

```
//Interface for slider object
#ifndef __SLIDDER_H_
#define __SLIDDER_H_

#include <FL/Fl_Value_Slider.H>
#include <vector>

class Listener;

class Slider : public Fl_Value_Slider
{
public:

    Slider(int x, int y, int width, int height, const char* label = 0);
    virtual ~Slider();
    virtual void Add_listener(Listener* listener);

    virtual void Notify();

private:
    Slider(const Slider& src);
    const Slider& operator=(const Slider& src);
    std::vector<Listener*> listeners;
};

#endif //ndef __Slider_H_
```

slider.cpp

```
//Implementation for slider object
#include "slider.h"
#include "listener.h"

using namespace std;

void Slider_callback(Fl_Widget* widget)
{
    Slider* slider = dynamic_cast<Slider*>(widget);
    if (slider != NULL)
        slider->Notify();
}

Slider::Slider(int x, int y, int width, int height, const char *label)
    : Fl_Value_Slider(x, y, width, height, label)
{
    callback(Slider_callback);
}

Slider::~Slider()
{
}

void Slider::Add_listener(Listener* listener)
{
}
```

```

        listeners.push_back(listener);
    }

void Slider::Notify()
{
    for (vector<Listener*>::iterator it = listeners.begin();
         it != listeners.end(); it++)
    {
        (*it)->Widget_changed(this);
    }
}

```

text_box.h

```

//Interface for non-buffer text output object
#ifndef _TEXT_BOX_H_
#define _TEXT_BOX_H_

#include <FL/Fl_Input.H>
#include <vector>

class Listener;

class Text_Box : public Fl_Input
{
public:
    Text_Box(int x, int y, int width, int height, const char* label =
    NULL);
    virtual ~Text_Box();
    virtual void Add_listener(Listener* listener);
    virtual void Notify();

private:
    Text_Box(const Text_Box& src);
    const Text_Box& operator=(const Text_Box& src);
    std::vector<Listener*> listeners;
};

#endif //ndef _TEXT_BOX_H_

```

text_box.cpp

```

//Implementation for non-buffer text output object
#include "text_box.h"
#include "listener.h"

using namespace std;

void Text_callback(Fl_Widget* widget)
{
    Text_Box* box = dynamic_cast<Text_Box*>(widget);
    if (box != NULL)
    {
        box->Notify();
    }
}

```

```

    }
}

Text_Box::Text_Box(int x, int y, int width, int height, const char* label)
    : Fl_Input(x, y, width, height, label)
{
    callback(Text_callback);
    when(FL_WHEN_CHANGED);
}

Text_Box::~~Text_Box()
{
}

```

text_box_nonrealtime.h

```

//Interface for buffer based text object
#ifndef _TEXT_BOX_NONREALTIME_H_
#define _TEXT_BOX_NONREALTIME_H_

#include <FL/Fl_Input.H>
#include <vector>

class Listener;

class Text_Box_Nonrealtime : public Fl_Input
{
public:
    Text_Box_Nonrealtime(int x, int y, int width, int height, const char*
        label = NULL);
    virtual ~Text_Box_Nonrealtime();
    virtual void Add_listener(Listener* listener);
    virtual void Notify();
private:
    Text_Box_Nonrealtime(const Text_Box_Nonrealtime& src);
    const Text_Box_Nonrealtime& operator=(const Text_Box_Nonrealtime& src);
    std::vector<Listener*> listeners;
};

#endif //ndef _TEXT_BOX_H_

```

text_box_nonrealtime.cpp

```

//Implementation for buffer based text object
#include "text_box_nonrealtime.h"
#include "listener.h"

using namespace std;

void Text_nonrealtime_callback(Fl_Widget* widget)
{
    Text_Box_Nonrealtime* box =
        dynamic_cast<Text_Box_Nonrealtime*>(widget);
    if (box != NULL)

```

```

        {
            box->Notify();
        }
    }

Text_Box_Nonrealtime::Text_Box_Nonrealtime(int x, int y, int width, int
height, const char* label): Fl_Input(x, y, width, height, label)
{
    callback(Text_nonrealtime_callback);
    when(FL_WHEN_ENTER_KEY | FL_WHEN_NOT_CHANGED);
}

Text_Box_Nonrealtime::~~Text_Box_Nonrealtime()
{
}

void Text_Box_Nonrealtime::Add_listener(Listener* listener)
{
    listeners.push_back(listener);
}

void Text_Box_Nonrealtime::Notify()
{
    for (vector<Listener*>::iterator it = listeners.begin();
        it != listeners.end(); it++)
    {
        (*it)->Widget_changed(this);
    }
}

```

timer.h

```

//Software based timer class
#include <ctime>

class Timer {
public:
    typedef double diff_type;
    Timer(): start(std::clock()), elapsed(0) {}
    diff_type last() const
    {
        return elapsed;
    }
    void begin()
    {
        start = std::clock();
        elapsed = 0;
    }
    diff_type end();
    diff_type getCurrent();

private:
    std::clock_t start;
    diff_type elapsed;
};

```

```
Timer::diff_type Timer::end()
{
    elapsed = (diff_type)std::clock() - start;
    elapsed /= CLOCKS_PER_SEC;
    return elapsed;
}

Timer::diff_type Timer::getCurrent()
{
    elapsed = (diff_type)std::clock() - start;
    return elapsed;
}
```