Spring 2017

Stringless Guitar

Kue Z. Yang
University of Akron, kzy1@zips.uakron.edu

Anthony Batey
University of Akron, arb125@zips.uakron.edu

Dominic Mercorelli
University of Akron, dlm158@zips.uakron.edu

Nathaniel Hawk
University of Akron, nah38@zips.uakron.edu

Recommended Citation
Yang, Kue Z.; Batey, Anthony; Mercorelli, Dominic; and Hawk, Nathaniel, "Stringless Guitar" (2017). *Honors Research Projects*. 508.
http://ideaexchange.uakron.edu/honors_research_projects/508

Kue Yang

Computer Engineering

May 1st, 2017

Honors Project

Senior Design Contribution

My main contribution to the design project was mainly designing, testing, and maintaining the software used for the project. My individual contribution to the project are listed as follows:

- I had selected the microprocessor that our project used. The microprocessor that was used is the PIC32MX270F512L from Microchip.
- I maintained, documented, and updated the source code throughout the semester. The project was hosted on GitHub for ease of access both at home and at school. Doxygen was used to generate the documentation pulled from the source code.
- I research and designed algorithms used in the project. For example, how audio data was read and written from memory to a digital to analog converter via the use of interrupts.
- I created debugging tools to test the functionality of the guitar. The debugging tool utilized UART to serial communications via a command terminal to send commands to the microprocessor.
- I assisted my team members with designing the hardware portions of the project. An excel sheet was created listing the pinouts of the microprocessor and how it was to be connected to other components.

Stringless Guitar

Senior Project Final Report

Design Team #08

Batey, Anthony

Hawk, Nathaniel

Mercorelli, Dominic

Yang, Kue

Dr. Robert J. Veillette

Monday, May 1st, 2017

Abstract

The aim of this project is to improve the design of a typical guitar by designing a digital stringless guitar. Due to the nonlinearities inherent in a guitar, it would be difficult to reproduce guitar tones by summing harmonic components; therefore, digital samples of guitar tones were taken in order to preserve these unique and wonderfully sounding tones. These digital samples were stored in the guitar and used to produce its tones when played. The digital guitar includes fingerboard position sensors as well as strum sensors for each string. The fingerboard position sensors detect the player's fingers at discrete positions. The strum sensors detect which string is being strummed. All the sensors are durable and may be replaced as the instrument ages. The guitar produces sound similar to a typical guitar without requiring strings or tuning. The stringless guitar has many of the standard features of a typical guitar including a standard guitar audio jack, volume control and tone control. The hope of this project is to produce a novel digital guitar that may be admired by professional and beginner musicians alike.

[NAH, KZY]

Table of Contents

List of Figures

List of Tables

1. Problem Statement

1.1 Need

A guitar requires the use of strings to create its sound. Strings frequently need to be replaced after the sound becomes dull. The guitar must be tuned for it to produce proper pitch of the tones. Furthermore, the pitch and quality of the guitar sound are sensitive to temperature changes. Therefore, tuning of the guitar is required before each usage.

[ARB, NAH, DLM, KZY]

1.2 Objective

The objective of this project is to design a guitar that does not require the use of strings. The neck of the guitar has six arrays of sensors for detecting finger placement and chord arrangements. The body of the guitar has six sensors for detecting the strumming of each string. The guitar is battery-powered and can be connected to a standard guitar amplifier. Other similar features of an electric guitar such as volume and tone controls are included in the overall design.

[ARB, NAH, DLM, KZY]

1.3 Background

1.3.1 Patent Search

Similar concepts of the stringless guitar have been implemented in various applications. In [1], an actuator blade on the body of the guitar was strummed or picked. This strumming action would flex each actuator blade back and forth. Flexing each of the actuator blades controlled the amplified output of an oscillator, which in return produced the frequency of the note that should play based on the finger position detected on the fret board [1]. Another implementation involved the use of push buttons on the neck of the digital guitar. These buttons would allow a microcontroller to detect the note that should be played [2]. A sensor matrix was another method that was used to detect the position of the fingers on the neck. Change in pressure exerted on the sensor would increase the pitch of the note. This behavior is implemented to replicate a small feature that musicians expect when playing a regular guitar with physical strings [3].

[DLM]

1.3.2 Article Search

Many features have been explored in guitar electronics. For example, when writing music for the guitar many people write either sheet music or tablature using computer software. This requires that the individual can read sheet music and is well versed in musical notation. To alleviate this requirement, Fiss and Kwasinski discuss a computer algorithm that converts guitar audio to guitar tablature or sheet music automatically. The article explains that this is hard to do using just guitar audio because there are up to five places on the guitar fretboard that produces the same musical note. The software would need to determine which one of the five finger positions are being played. Sheet music and tablature could be generated based on the data received from the instrument [4].

Another way finger position has been monitored on the fret board is by using infrared sensors to emulate strings. In [5], an infrared transmitter was placed on one side of the guitar neck with photodiodes on the body. A microcontroller would read the output of the photodiodes when a person held a chord on the neck. The microcontroller would send the finger position to a computer. From there, a computer application would produce the corresponding guitar tone.

In [6], a digital guitar was created to sound like an acoustic guitar. The actual pitch of the string was recreated as well as the tone of the pitch produced by the body of the guitar. In order to create a computational model, algorithms were developed to mimic the guitar body. The model incorporated the horizontal and vertical movement of the strings being plucked, which were dependent on the strength of each pluck. Through various filtering methods and the combination of the algorithms developed, the acoustic guitar sound synthesized was very similar the original guitar being studied [6].

In [7], a stringless guitar design was presented where the main problem analyzed was how to detect chords. This stringless guitar design consists of three parts. The first part included a “chord glove” composed of FSR and FLEX sensors, which would detect the guitar chords. The second part of the project involved the use of a processor that would recognize the guitar chord input from the “chord glove”. Finally, each chord was transmitted to sound processing software where the chord formation was displayed graphically. The guitar could detect strumming in up or down strokes [7].

[ARB, NAH, DLM]

1.4 Marketing Requirements

1. The guitar battery will be replaceable and rechargeable.
2. The guitar will last for at least four hours of constant usage.
3. The stringless guitar will have a standard guitar audio output jack.
4. The stringless guitar will be the same size as a normal guitar.
5. The stringless guitar will require no tuning.
6. The stringless guitar will respond and play in real-time.
7. The stringless guitar will operate over a typical range of temperatures.
8. The sensors of the stringless guitar will be durable.
9. The sensors of the stringless guitar will be replaceable.
10. The stringless guitar will weigh less than fourteen pounds.
11. The stringless guitar will have volume and tone controls.

[ARB, NAH, DLM, KZY]

1.5 Objective Tree

Figure 1 shows the objective tree for the Stringless Guitar. The objective tree lists the basic objectives met by the Stringless Guitar and are broken down into three different categories: Input and Output Data, Robust and Portable and User Friendly. The Input and Output Data category consists of design objectives that are related to the audio processing implementation of the project. The Robust and Portable category consists of design requirements related to the hardware components of the project. The User-Friendly category consists of design requirements that a user would expect from this project.

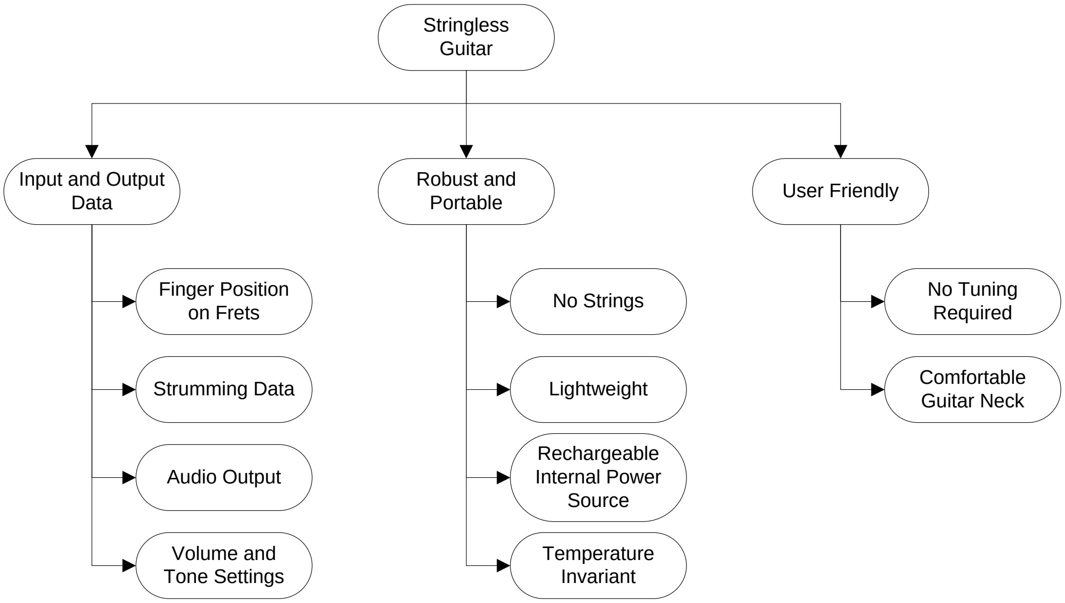


Figure 1: Objective Tree

[ARB, NAH, DLM, KZY]

2 Design Requirements Specifications

The design requirements for the Stringless Guitar project are listed in Table 1. The design requirements were derived from the marketing requirements listed above. Each design requirements corresponds to each of the objectives listed in the objective tree from Figure 1.

Table 1: Design Requirements

Market Requirements	Engineering Requirements	Justification
1	The guitar will have an internal power source that can be charged by an external power source.	To eliminate the need for the guitar to run off an external power source.
2	The guitar will last for at least 4 hours with constant usage.	To operate the components under worst-case conditions for the required amount of time.
3	The guitar will have a ¼” female audio output connector.	To connect to pre-existing guitar cables.
4	The guitar will have a scale length within 24 to 26 inches.	Therefore, the average user will be familiar with the dimensions.
5	The guitar will produce audio output ranging from 20 Hz to 20 kHz.	The user will not need to tune the guitar.
6	The guitar will output the corresponding audio file under a maximum allowable delay of 100 milliseconds.	The time the device takes to make calculations must be short enough so that the user does not notice a delay between their inputs and the audio output.
7	The guitar will operate within typical temperature ranges of outdoor concerts. Temperature ranges from 30^{O}F to 110^{O}F.	If the user is using the guitar in an outdoor venue, the electronics will perform as intended.
8	The guitar sensors will last at least one year with constant usage.	The guitar is designed such that the user will not need to change the sensors as often as one may need to change a typical set of guitar strings.
9	The guitar sensors will be replaceable.	If a sensor fails, the sensor can be replaced with a new sensor.
10	The guitar will weigh less than fourteen pounds.	The guitar will weigh similarly to other guitars.
11	The guitar will have volume and tone controls ranging from ±6 dB.	The user will be able to adjust the volume and tone of the guitar.

[ARB, NAH, DLM, KZY]

3 Accepted Technical Design

3.1 Hardware Design

3.1.1 Hardware Theory of Operation

The overall design of the stringless guitar is shown in Figure 2. The hardware implemented comprises four core modules as shown in Figure 3. The four modules are: Sensor Circuits, MCU and Memory Circuits, Amplifier Circuit, and Power Supply.

The Sensors Circuits module handles getting the player's finger and strumming inputs to the guitar. The Sensors Circuits module is shown in Figure 4. The hardware consists of finger and strumming sensors on the neck and body of the guitar, respectively. The finger position sensors senses the locations of the player's fingers on the neck of the guitar. The finger position sensors, when pressed, generates a digital high that is seen by a microcontroller. The finger position sensors are digital inputs to the microcontroller. There are a total of twenty finger position sensors per "string" simulating the twenty frets of a guitar. The strumming sensor will determine when a "string" is being strummed and how hard the player strummed. When a force is applied on the strum sensors, the sensors generates a voltage that is read by a microcontroller as an analog input. The strumming sensor does not require an outside voltage source.

The MCU and Memory Circuits modules handles reading and processing audio files from an external memory. The MCU and Memory Circuits modules is shown in Figure 5. The module consists of an array of six microcontrollers running in parallel. Each microcontroller will handle processing audio for one of the six guitar strings to improve the response of the guitar. Each microcontroller has its own external memory containing audio files corresponding to each note of the string. In other words, each microprocessor will handle up to 21 different tones. An audio file is read from external memory when the microcontroller receives inputs from both the strumming and finger sensors. The microcontroller will send a sixteen-bit digital data stream to the amplifier circuit as shown in Figure 6.

The Amplifier Circuit module is the module that handles how the analog audio outputs are generated and processed into an analog signal. This module is shown in Figure 6. The module contains six, sixteen-bit digital-to-analog converters, each of which will convert one sixteen-bit digital audio signal to an analog signal. The six analog signals are added together using a summing amplifier circuit to produce a single analog audio output. The volume and tone controls are incorporated into the summing amplifier circuit.

The power supply module is used to supply power to the guitar. The power supply module is shown in Figure 7. The module consists of a lithium ion battery and a linear voltage regulator. The +9 VDC source from the battery is used to power the summing amp and the strumming sensors. The +9 VDC source from the battery is also regulated down to +3.3 VDC to power the MCU, DAC and SD Card reader. An external supply is used to charge the battery when needed.

Table 2 through Table 14 describe the functionality of each of the modules and their components as discussed above.

[ARB, NAH, DLM, KZY]

3.1.2 Block Level Zero Hardware Diagram

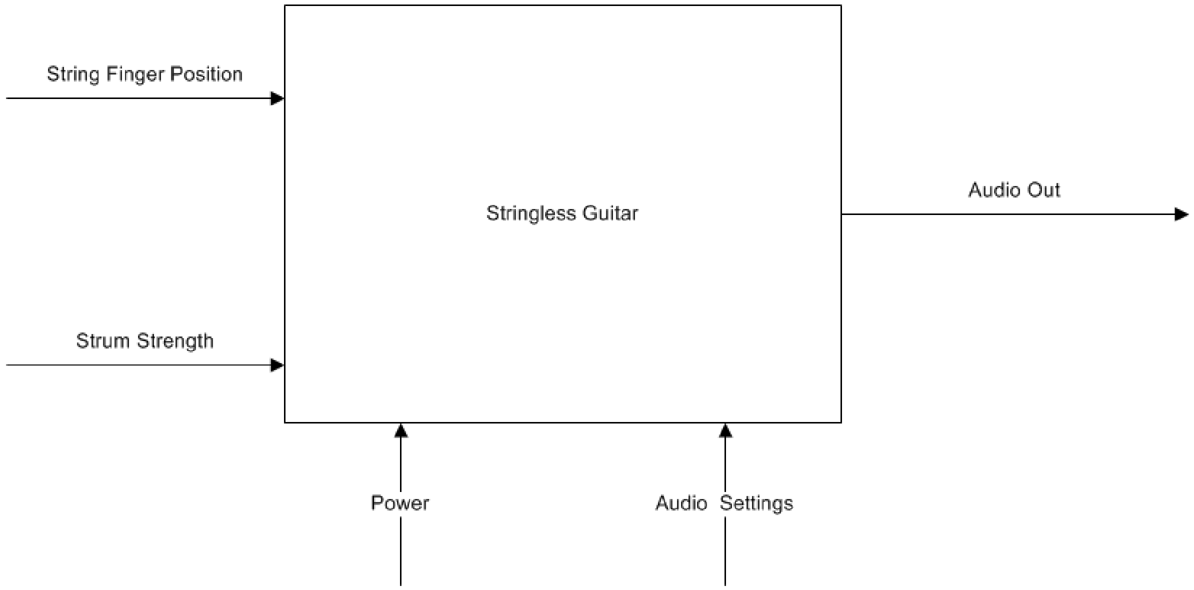


Figure 2: Block Level Zero Hardware Diagram

Table 2: Level Zero Hardware Functional Outputs

Module	Stringless Guitar
Designer(s)	ARB, NAH, DLM, KZY
Inputs	○ Power ○ Strum Strength ○ String Finger Position ○ Audio Settings
Outputs	○ Audio Out
Functionality	○ The stringless guitar will process the string finger positions and strumming strength into audio output. The audio output will be similar to the audio produced at the output of a standard guitar. The stringless guitar will also process audio settings (e.g. volume) and apply changes to the audio output.

3.1.3 Block Level One Hardware Diagram

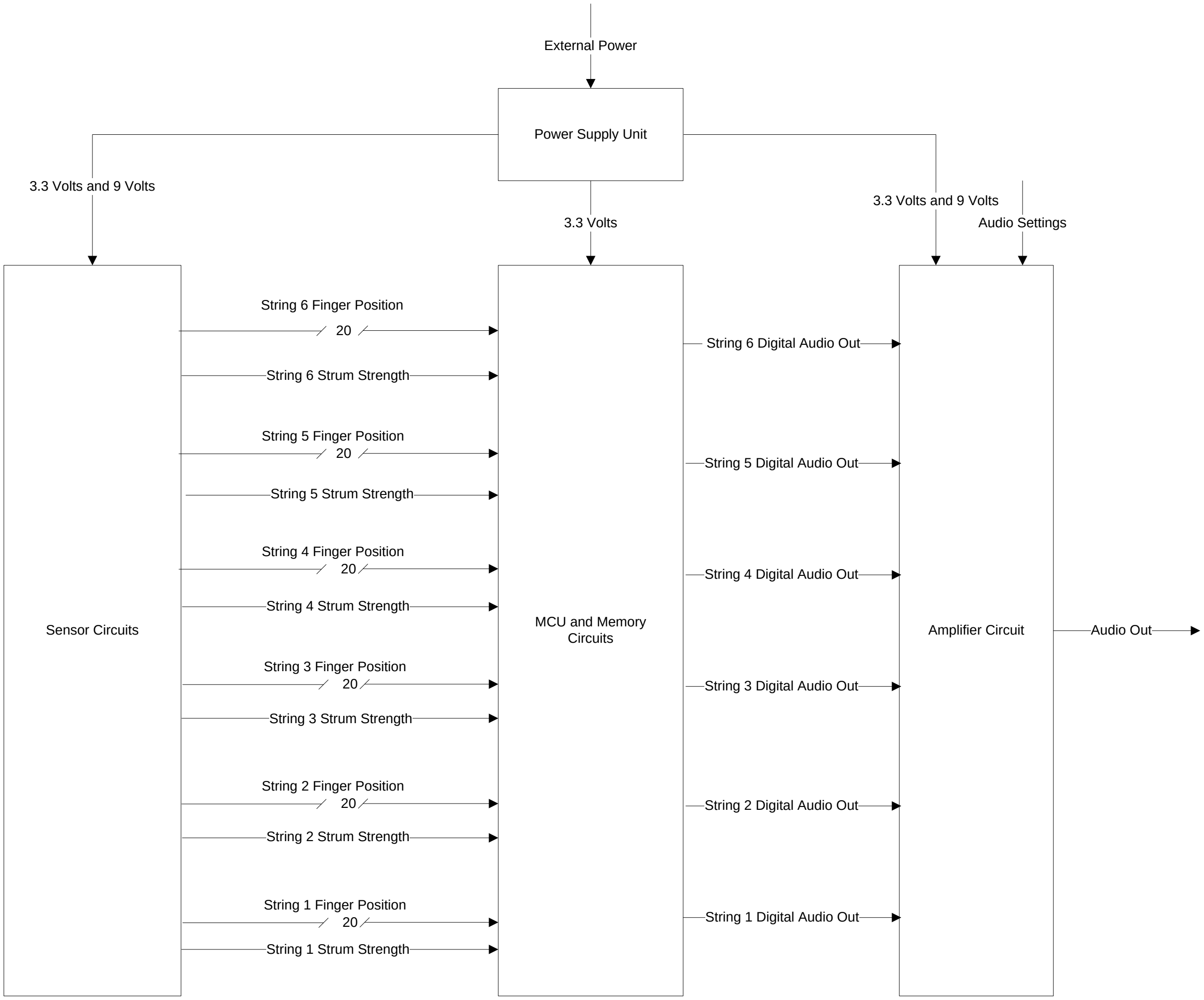


Figure 3: Block Level One Hardware Diagram

Table 3: Level One Hardware Functional Outputs – Sensor Circuits

Module	Sensor Circuits
Designer(s)	ARB, NAH, DLM
Inputs	o +3.3 VDC, 1 A o +9 VDC, 1 A
Outputs	o String Finger Positions (*1*-*6*) (Digital Voltages, 0 – 3.3 VDC) o String Strum Strengths (*1*-*6*) (Analog Voltages, 0 – 9.0 VDC)
Functionality	o The sensor circuits will produce output voltages corresponding to the finger positions on the neck of the guitar and how hard the guitar is strummed for each string.

Table 4: Level One Hardware Functional Outputs – MCU and Memory Circuits

Module	MCU and Memory Circuits
Designer(s)	ARB, NAH, DLM
Inputs	o String Finger Positions (*1*-*6*) (Digital Voltages, 0 – 3.3 VDC) o String Strum Strengths (*1*-*6*) (Analog Voltages, 0 – 9.0 VDC) o +3.3 VDC, 1A
Outputs	o String Digital Audio Out (*1*-*6*) (SPI communication protocol, 8MHz)
Functionality	o The microcontroller unit will process the finger position and strum strength sensor voltages of each string, and produce a digital audio output.

Table 5: Level One Hardware Functional Outputs – Amplifier Circuit

Module	Amplifier Circuit
Designer(s)	ARB, NAH, DLM
Inputs	o String Digital Audio Out (*1*-*6*) (SPI communication protocol, 8MHz) o Audio Settings o +9 VDC, 1 A o +3.3 VDC 1 A
Outputs	o Audio Out (Voltage Range of – 4.5 V to + 4.5 V)
Functionality	o The amplifier circuit will take multiple digital audio signals from the array of microcontrollers and produce a single continuous audio output. The audio settings will change the tone and volume of the audio output.

Table 6: Level One Hardware Functional Outputs – Power Supply

Module	Power Supply
Designer(s)	DLM
Inputs	○ +13.1 VDC, 1.5 A External Power Supply
Outputs	○ +3.3 VDC, 1 A ○ +9 VDC, 1 A
Functionality	○ The power supply will apply a constant supply voltage to each of the components so that the components will run accordingly.

3.1.4 Block Level Two Hardware Diagram

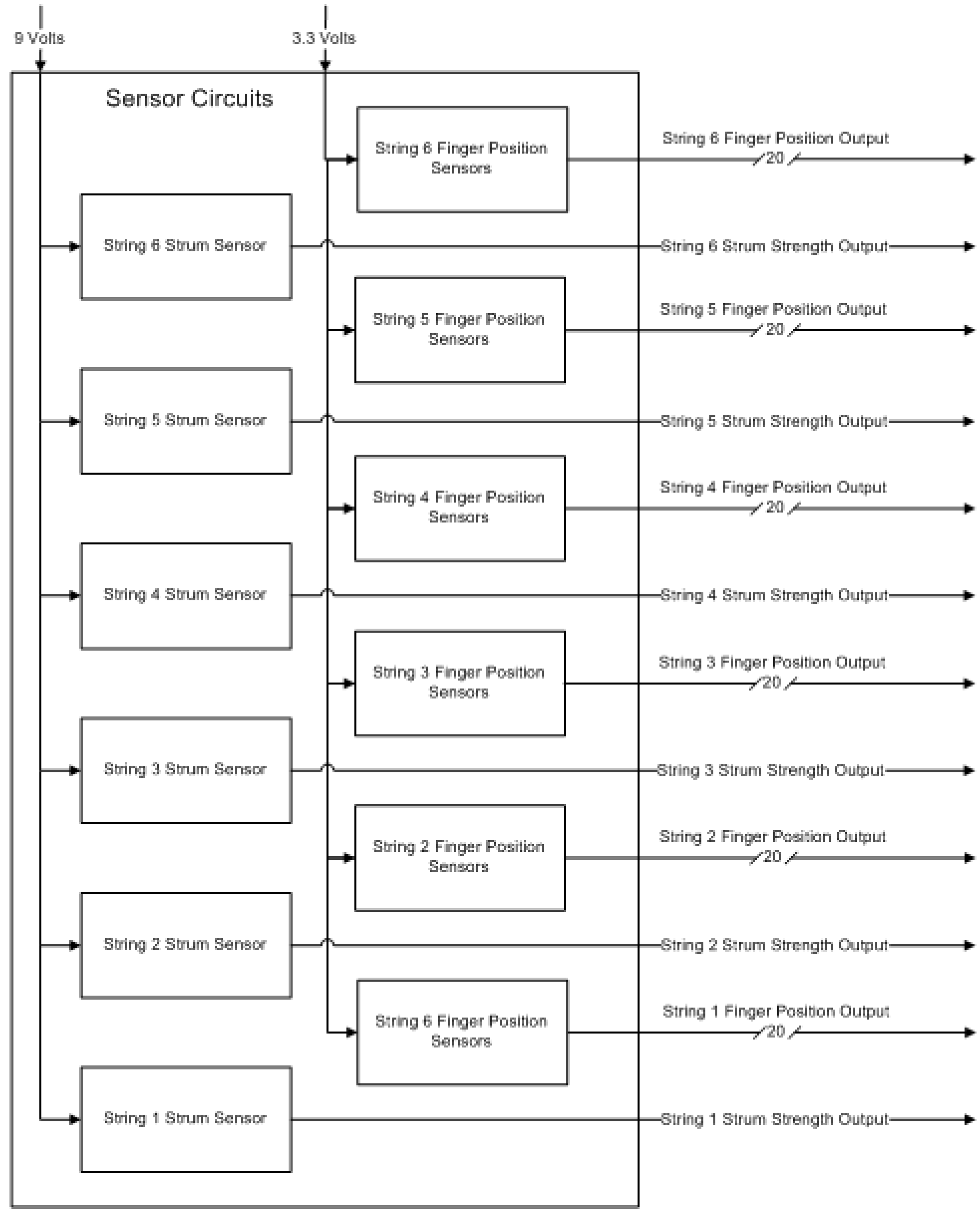

9 Volts
3.3 Volts
Sensor Circuits
String 6 Finger Position Sensors
String 6 Finger Position Output
20
String 6 Strum Sensor
String 6 Strum Strength Output
String 5 Finger Position Sensors
String 5 Finger Position Output
20
String 5 Strum Sensor
String 5 Strum Strength Output
String 4 Finger Position Sensors
String 4 Finger Position Output
20
String 4 Strum Sensor
String 4 Strum Strength Output
String 3 Finger Position Sensors
String 3 Finger Position Output
20
String 3 Strum Sensor
String 3 Strum Strength Output
String 2 Finger Position Sensors
String 2 Finger Position Output
20
String 2 Strum Sensor
String 2 Strum Strength Output
String 6 Finger Position Sensors
String 1 Finger Position Output
20
String 1 Strum Sensor
String 1 Strum Strength Output

Figure 4: Block Level Two Hardware Diagram – Sensor Circuits

Table 7: Level Two Hardware Functional Outputs – Sensor Circuits: String Finger Position Sensors

Module	String Finger Position Sensor (*1*-*6*)
Designer(s)	ARB, NAH, DLM
Inputs	○ +3.3 VDC, 1 A
Outputs	○ Finger Position Output (*1*-*6*) (Digital Voltages, 0 – 3.3 VDC)
Functionality	○ The finger position sensor will determine which fret is being pressed.

Table 8: Level Two Hardware Functional Outputs – Sensor Circuits: String Strum Strength Sensors

Module	String Strum Strength Sensor (*1*-*6*)
Designer(s)	ARB, NAH, DLM
Inputs	○ +9 VDC, 1 A
Outputs	○ Strum Strength Output (*1*-*6*) (Analog Voltages, 0 – 9 VDC)
Functionality	○ The strum strength will determine the amplitude and duration of the audio signal.

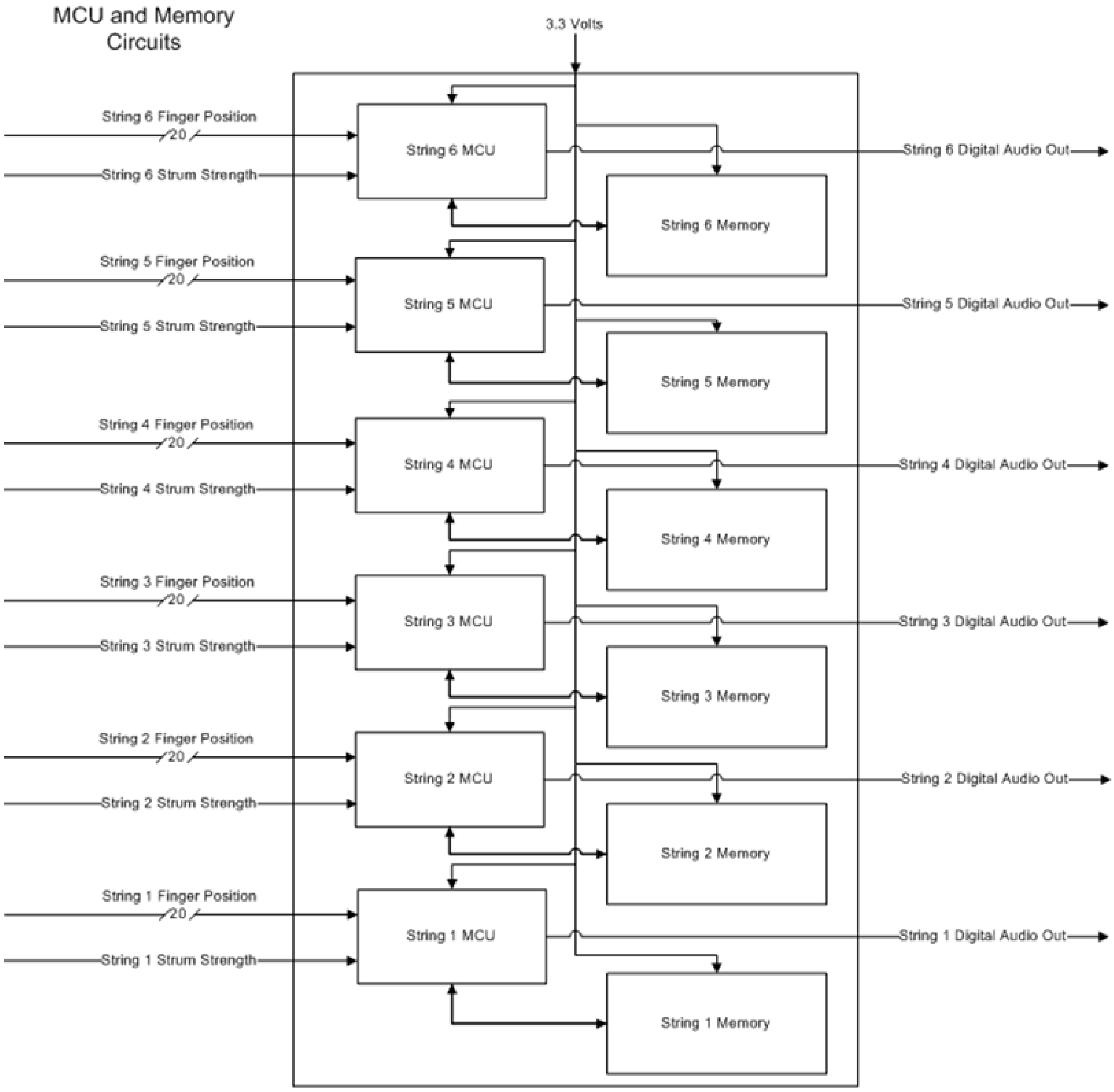


Figure 5: Block Level Two Hardware Diagram – MCU and Memory Circuits

Table 9: Level Two Hardware Functional Outputs – MCU and Memory Circuits: Microcontroller

Module	String Microcontroller (*1*-*6*)
Designer(s)	ARB, NAH, DLM
Inputs	○ + 3.3 VDC, 1 A ○ String Finger Position (*1*-*6*) (Digital Voltages, 0 – 3.3 VDC) ○ String Strum Strength (*1*-*6*) (Analog Voltages, 0 – 9 VDC) ○ String MCU Rx (*1*-*6*) (Digital Voltages, 0 – 3.3 VDC)
Outputs	○ Digital Audio Out (*1*-*6*) (Digital Voltages, 0 – 3.3 VDC) ○ String MCU Tx (*1*-*6*) (Digital Voltages 0 – 3.3 VDC)
Functionality	○ Each “string” has a microcontroller that reads a specific audio file corresponding to the finger positions and the strum strength sensor and create a digital audio output. Each of the microcontrollers will interface with an external memory, which will provide the audio files.

Table 10: Level Two Hardware Functional Outputs – MCU and Memory Circuits: Memory

Module	String Memory (*1*-*6*)
Designer(s)	ARB, NAH, DLM
Inputs	○ +3.3 VDC, 1 A ○ String MCU Tx (*1*-*6*) (Digital Voltages, 0 – 3.3 VDC)
Outputs	○ String MCU Rx (*1*-*6*) (Digital Voltages, 0 – 3.3 VDC)
Functionality	○ Each “string” has its own external memory that will store multiple audio files corresponding to the tones of that particular “string”. The microcontroller will interface with the external memory to read specific sound files as required.

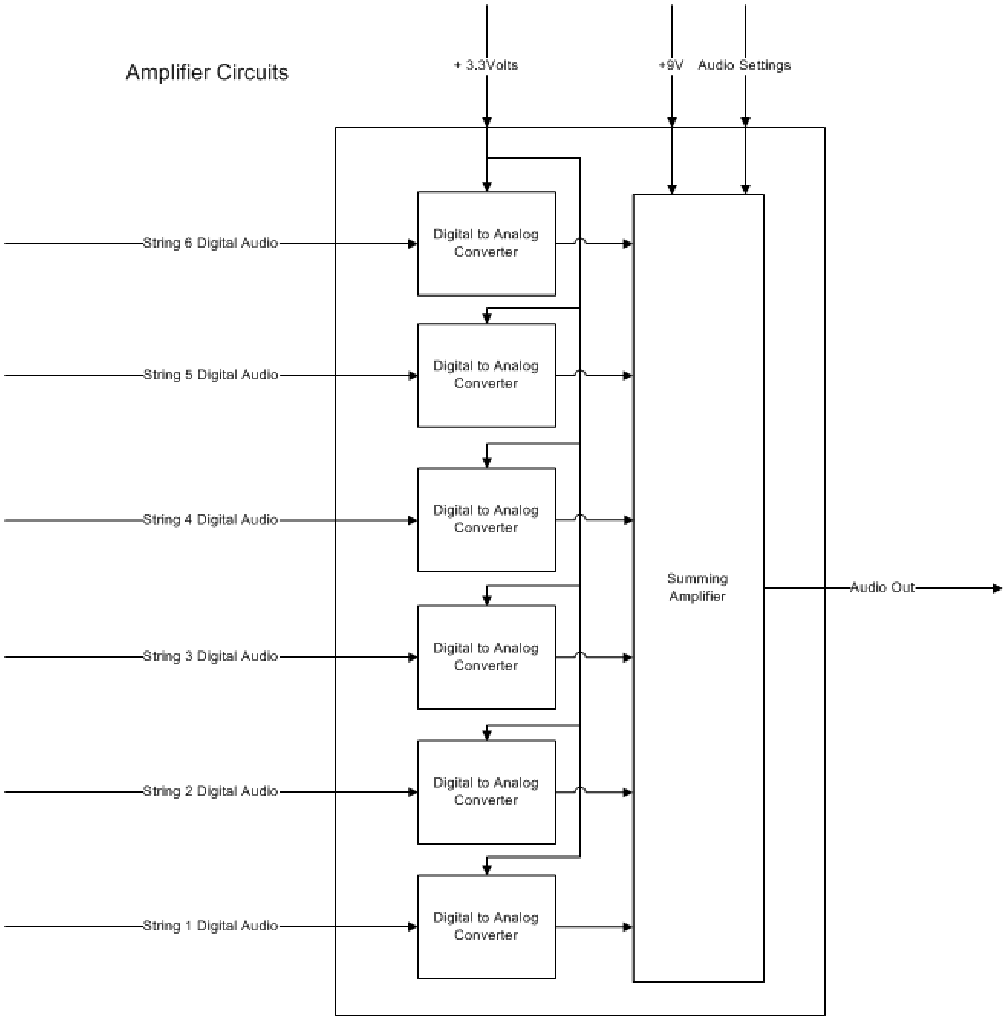


Figure 6: Block Level Two Hardware Functional Outputs – Amplifier Circuits

Table 11: Level Two Hardware Functional Outputs – Amplifier Circuits: DACs

Module	DAC (*1*-*6*)
Designer(s)	ARB, NAH, DLM
Inputs	○ 16-bit String Digital Audio (*1*-*6*) (SPI communication protocol, 8MHz) ○ +3.3 VDC, 1 A
Outputs	○ Audio Out Signal (*1*-*6*) (Analog Voltages, 0 – 3.3 VDC) ○
Functionality	○ Each "string" has a digital-to-analog converter that will convert the string's digital audio output into an analog audio out signal. The summing amplifier will receive the analog output signal from the DAC.

Table 12: Level Two Hardware Functional Outputs – Amplifier Circuits: Summing Amplifier

Module	Summing Amplifier
Designer(s)	ARB, NAH, DLM
Inputs	○ Audio Out Signals (*1*-*6*) (Analog Voltages, 0 – 3.3 VDC) ○ Audio Settings (Analog Volume and Tone Control) ○ +9 VDC, 1 A
Outputs	○ Audio Out (Analog Voltages, 0 – 9 VDC)
Functionality	○ The summing amplifier will take six analog audio out signals and convolute the signals into one audio out signal. The summing amplifier will also control the volume and tone of the audio out signal depending on the audio settings the user selects.

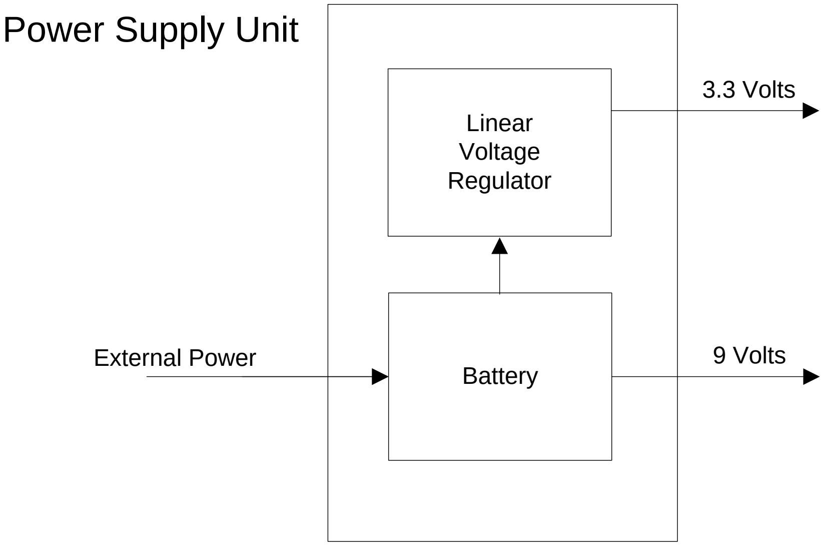


Figure 7: Block Level Two Hardware Diagram – Power Supply Unit

Table 13: Level Two Hardware Functional Outputs - Power Supply Unit: Battery

Module	Battery
Designer(s)	DLM
Inputs	○ +13.1VDC, 1.5 A External Supply
Outputs	○ +9VDC, 1A
Functionality	○ The battery is used to supply +9VDC to the input of the linear voltage regulator. The linear voltage regulator drops the +9V input to +3.3 V. The external supply voltage of +13.1VDC is used to charge the battery through a port on the circuit board. The +9VDC is coming from the port on the battery which is used to supply a constant +9VDC to each of component of the Stringless guitar.

Table 14: Level Two Hardware Functional Outputs - Power Supply Unit: Linear Voltage Regulator

Module	Linear Voltage Regulator
Designer(s)	DLM
Inputs	○ +9 VDC, 1A Battery Supply
Outputs	○ +3.3 VDC, 1A ○
Functionality	○ The linear regulator is supplied by a constant +9VDC from the battery, then the linear voltage regulator is used to supply a constant +3.3 VDC to each component of the Stringless guitar.

3.2 Software Design

3.2.1 Software Theory of Operation

The software design of the stringless guitar is shown in Figure 8. At startup, the device's peripherals are initialized and then the main application processes start. The main application processes comprise two main modules during runtime that run in sequence as shown in Figure 9. The first module, the Timeout module, is the module that handles resetting the microprocessor if the software stalls during runtime.

The second module, the Audio Process module, handles how the audio is processed when a single guitar string is played. The guitar uses a total of six processors in the final design. The microprocessor will receive input data with regards to where the user's fingers are on the neck of the guitar and whether the user strummed or picked the guitar string. If the user strums the guitar string, the microprocessor selects the appropriate guitar tone from a lookup table based on the finger position data. The lookup table has information regarding the file size, location and name that are mapped to each fret of the guitar. The finger position is used to read the corresponding audio file from the external memory. As the audio file is being read, the audio data is buffered into the microprocessor prior to being written to a digital to analog converter. A more detailed flow chart of how audio is processed in the Audio Process Module is shown in Figure 10.

For further implementation details, refer to the Software Components Section.

[KZY]

3.2.2 Flow Chart Level Zero Software Diagram

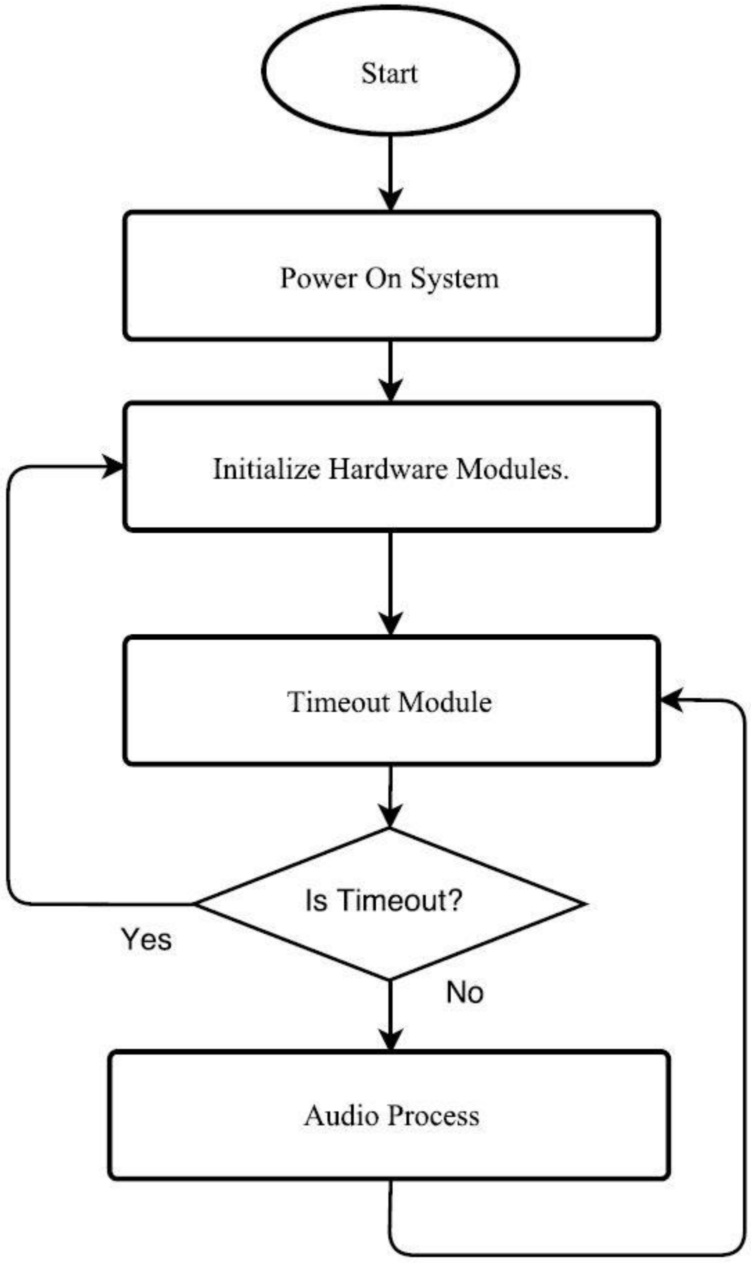


Figure 8: Flow Chart Level Zero Software Diagram

[KYZ]

3.2.3 Flow Chart Level One Software Diagram

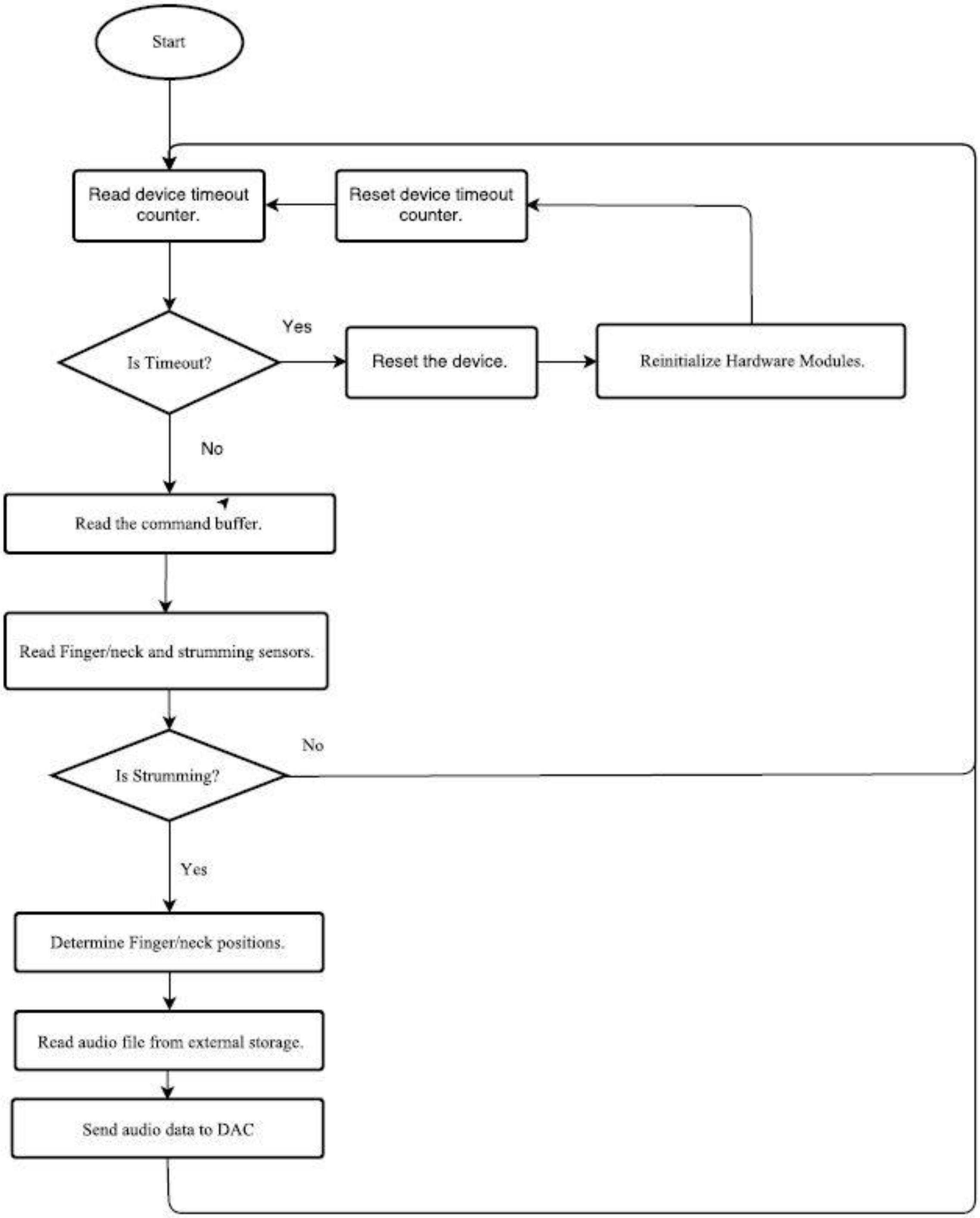

Start
Read device timeout counter.
Reset device timeout counter.
Yes
Is Timeout?
Reset the device.
Reinitialize Hardware Modules.
No
Read the command buffer.
Read Finger/neck and strumming sensors.
No
Is Strumming?
Yes
Determine Finger/neck positions.
Read audio file from external storage.
Send audio data to DAC

Figure 9: Flow Chart Level One Software Diagram – Run Main Processes

[KYZ]

3.2.4 Flow Chart Level Two Software Diagram

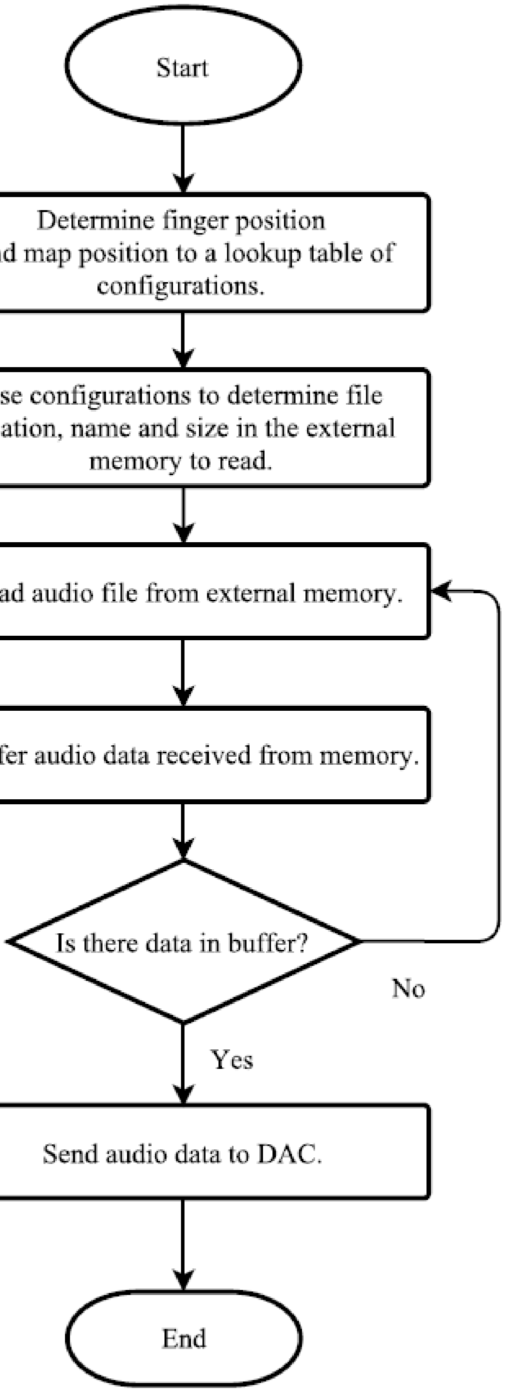

Start
Determine finger position and map position to a lookup table of configurations.
Use configurations to determine file location, name and size in the external memory to read.
Read audio file from external memory.
Buffer audio data received from memory.
Is there data in buffer?
No
Yes
Send audio data to DAC.
End

Figure 10: Flow Chart Level Two Software Diagram – Read Audio File from Storage

[KYZ]

3.3 Mechanical Design

3.3.1 Strum Sensor Mechanical Design

The stringless guitar has a custom mechanoelectrical strum sensing system shown in Figure 11. The physical design of the strum sensing system is as follows. First, thin general purpose steel plates are bolted together with thick aluminum plates and foam between them. The stiffness of the steel plate was determined such that when the user strums hard, the plate flexes 3mm. The aluminum plates are used to set the distance between the steel plates to approximately 10 mm (0.375 in.). The distance between the steel plates was decided to be 10 mm because the string spacing of a typical guitar was measured to be approximately 10 mm. "Loctite tak" was used between the steel plates and the aluminum spacers to create a good mechanical connection.

Foam is also used between the steel plates to provide mechanical isolation from plate to plate. Additionally, foam with thickness of one inch was used between the guitar body and the strum sensor assembly to prevent vibrations of individual steel plates from mechanically coupling through the guitar body. Foam was chosen to be used to dampen the steel plate oscillations because foam can be used as a vibration absorbing material. The foam used in the stringless guitar was reclaimed from discarded Tectronix packaging.

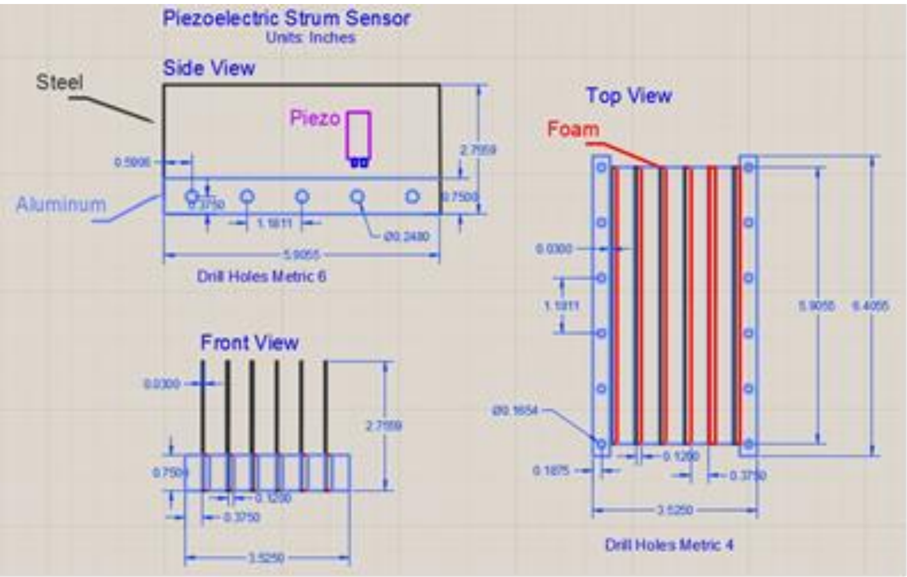


Figure 11: Mechanical Design of the Piezoelectric Strum Sensor

The piezoelectric flex sensors were mounted to the side of each steel plate as shown in Figure 11(a). The piezoelectric flex sensors were mounted using "Loctite tak".

The strum sensing system physically functions as follows. First, a steel plate is plucked or strummed. This force sets the plate into motion and it experiences oscillations damped by the foam. The top of the plate has a greater displacement then the bottom of the plate nearest the bolts because the bolts hold the plates relatively tightly together.

The components chosen for the strumming sensor were chosen for the following reasons. Steel was chosen because it is flexible and it is not malleable. When a force is applied to a steel plate, it is flexed out of shape. When the force is removed, the plate returns to its original form. This characteristic makes steel appealing to use in this strum sensing system. Aluminum was chosen to be used as spacers between the steel plates because aluminum relatively light. Foam is necessary to dampen the oscillations of the streel plates and to provide mechanical isolation from steel plate to steel plate.

[NAH]

3.4 Calculations

3.4.1 Battery Life Calculations

Calculations were done to determine the battery specs needed for the guitar. The calculations assume the worst-case scenario and the values can be found in [8]. The power needed for the stringless guitar can be calculated as

$$\boldsymbol{P = V \times I} \tag{1}$$

where $\boldsymbol{P}$ is the power, $\boldsymbol{V}$ is the voltage and $\boldsymbol{I}$ is the current. From [8], a single microprocessor requires a maximum current of 250 mA at 3.3 V. Using a total of six microprocessors, the maximum total current needed for the stringless guitar is 1.5 A at 3.3 V. Therefore, from Equation (1), the total power needed to operate the stringless guitar is 4.95 W.

The battery will be required to last for at least four hours. The energy consumed by the stringless guitar in this time can be calculated as

$$\boldsymbol{E = P \times T} \tag{2}$$

where $\boldsymbol{E}$ is energy consumed by the stringless guitar, $\boldsymbol{P}$ is the power consumed by the stringless guitar and $\boldsymbol{T}$ is the time that the stringless guitar must operate without a recharge. An approximation of 4.5 hours was used to make the calculation to ensure that there would be enough power in the battery to last for at least 4 hours. Therefore, from Equation (2), the energy that were calculated for the stringless guitar to last for 4.5 hours were 22.275 Wh.

The battery charge needed to last for the desired time can be calculated as

$$\boldsymbol{Q = \frac{E}{V}} \tag{3}$$

where $\boldsymbol{Q}$ is the battery charge, $\boldsymbol{E}$ is the energy consumed by the guitar and $\boldsymbol{V}$ is the voltage of the battery. The battery voltage that will be used in the design is 9 V. The energy was approximated at 22.275 Wh using Equation (2). Using the above values in Equation (3), the 9 V battery will be required approximately 2.475Ah to power the stringless guitar for at least four hours.

[NAH, DLM]

3.4.2 Storage Size, Bus Speed and Delay Calculations

The following calculations will be used to determine the number of files, the storage size as well as the bus speed needed for the guitar to function properly. The number of notes a guitar can play can be calculated as

$$\boldsymbol{N = S \times (F + 1)} \tag{4}$$

where ***N*** is the number of distinct notes, ***S*** is the number of strings and ***F*** is the number of frets per string. A typical guitar has six strings and 20 frets. Therefore, from Equation (4), the total number of notes a guitar can play is 126 distinct notes.

For the following calculations, the System Clock Frequency of the processor is assumed to be at 40 MHz. Assume that the file size of a four-second audio file is 1 MB. Using the calculations above, the total storage size needed is approximately 126 MB. By using six different processors to handle the six individual guitar strings, the total number of notes that each processor must handle will be 21 notes. The amount of external memory needed for each processor is approximately 21 MB.

Again, assume that the file size is 1 MB for four seconds of audio and the bus uses serial data to transfer data. The file size for one second of audio will be approximately 250 KB. Converting from bytes to bits, the file size of 250 KB will be 2 Mbits for one second of audio. The bus speed can be calculated as

$$\boldsymbol{B = \frac{F_s}{B_t}} \tag{5}$$

where $\boldsymbol{B}$ is the bus speed, $\boldsymbol{F_s}$ is the file size per second of audio in bits, and $\boldsymbol{B_t}$ is the number of bits transferred per clock cycle. From the above assumptions and calculations, $\boldsymbol{Fs}$ is 2 Mbits/s and $\boldsymbol{B_t}$ is 1 bit/cycle. Using these values in Equation (5), the bus speed was approximated at 2 MHz to transfer one second of audio per second.

The delay from when a guitar strum is detected to when the note is heard is difficult to determine due to the use of interrupts and priorities. Assuming worse case, the most delay will occur while reading data from external storage. Because reading data from external storage takes priority over all other functions, timing the read time was not possible. However, in the final design of the stringless guitar, there were no noticeable delay.

[KZY]

3.4.3 Strum Sensor Calculations

The piezoelectric strip produces a voltage in response to the user strumming the corresponding steel plate. The piezoelectric strip model LDT0-028K is shown in Figure 12.

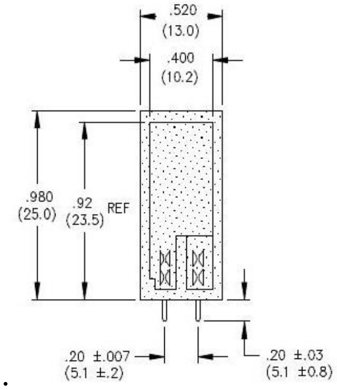


Figure 12: Piezoelectric Strip Model LDT0-028K

The relative maximum and relative minimum voltage produced by the LDT0-028K piezoelectric strip has been estimated using data provided in [9]. The voltage against piezoelectric strip tip deflection is shown in Figure 13. As explained in [9], to obtain this given measurement, the piezoelectric strip was held firmly by its contacts and a charge amplifier was used to measure the piezo strip's open circuit voltage as the tip was deflected.

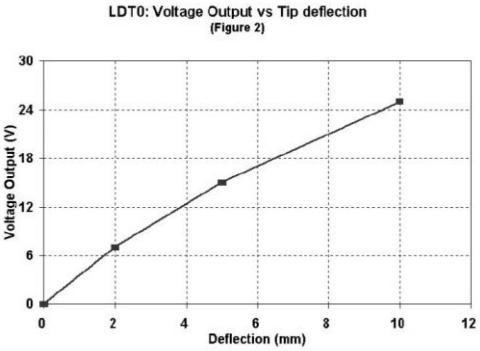


Figure 13: Piezoelectric voltage verses tip deflection

The relative maximum deflection that the tip of the steel plate experiences was measured to be 1 mm. This relative maximum deflection was measured with a "hard strum" from the user. Similarly, the relative minimum defection that the tip of the steel plate experiences was approximated as 0.1 mm. This relative minimum deflection corresponds with a "soft strum" from the user.

$$d_{r\_max} = 1 \ mm, \quad d_{r\_min} = 0.1 \ mm \tag{6}$$

Next, the relative minimum and relative maximum deflection angles were approximated using the above displacements approximations and a right triangle as shown in Figure 14.

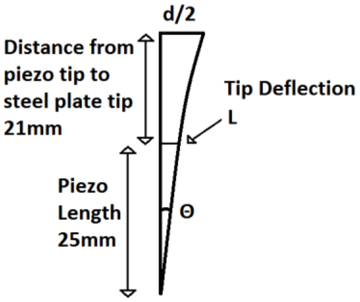


Figure 14: Piezo and Steel Plate Deflection

$$\Theta = \ tan^{-1}\left(\frac{\frac{d}{2}}{25+21}\right) \tag{7}$$

$$\Theta_{r\_max} = 0.622° \,, \quad \Theta_{r\_min} = 0.0622° \tag{8}$$

Using the relative maximum and relative minimum deflection angles above, the relative maximum and relative minimum deflection of the tip of the piezoelectric strip was calculated.

$$L = 25\tan(\Theta) \tag{9}$$

$$L_{r\_max} = 0.2717 \, mm \,, \quad L_{r\_min} = 0.02717 \, mm \tag{10}$$

Finally, the voltage that the piezoelectric strip will produce was approximated using Figure 13.

$$V_{piezo} \approx \frac{7}{2}L \tag{11}$$

$$V_{piezo\_max} \ \approx \frac{7}{2}L_{r\_max} = 0.95 \, V \tag{12}$$

$$V_{piezo\_min} \ \approx \frac{7}{2}L_{r\_min} = 0.095 \, V \,, \tag{13}$$

The relative maximum voltage produced by the piezoelectric strip is approximated at 0.95V and the relative minimum voltage produced by the piezoelectric strip is approximated at 0.095V.

[NAH]

3.5 Simulations

3.5.1 *Summing Amplifier Simulations*

Figure 15 shows the LTspice schematic used to simulate the summing amplifier circuit. The op amp is biased between the rails to improve amplification of the summed signals. The capacitor, C1, is used to remove the DC offset incurred from the op amp. A pull-down resistor, R10, is used to ensure that the output is not floating. Each branch is connected to different voltage sources to simulate different voltages that can be received from each DAC. Because all the audio signals are similar in magnitude, the same resistors are used for each branch.

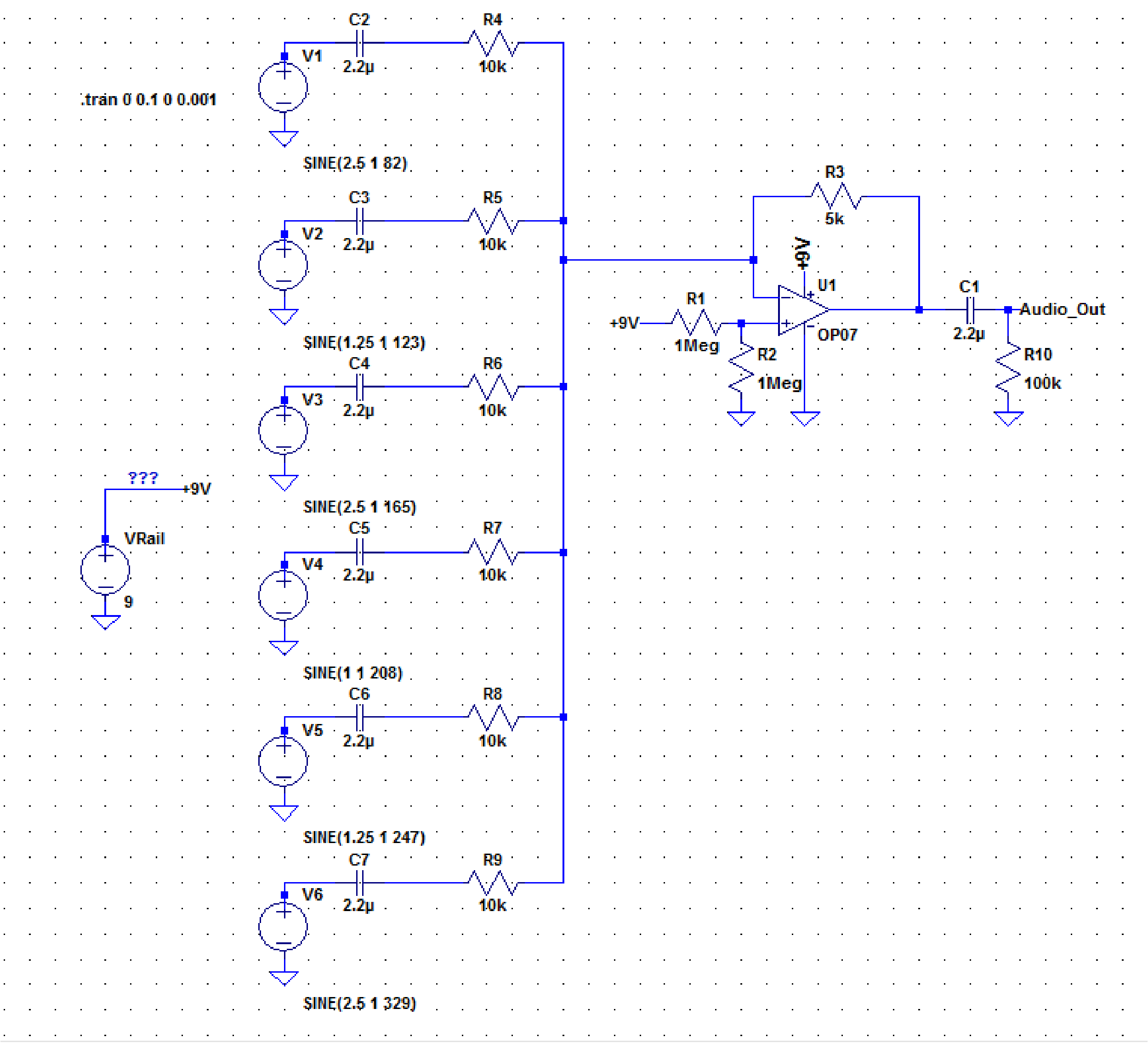


Figure 15: Summing Amplifier Simulation Schematic

The purpose of this simulation was to sum six sinusoidal input signals together. The six input signals represent the six audio output signals from the DAC. The input voltage **V1** corresponds to the first string of the guitar; while the input voltage **V6** corresponds to the sixth string of the guitar. The frequencies of these six inputs were set to represent the E major chord shown in Figure 16.

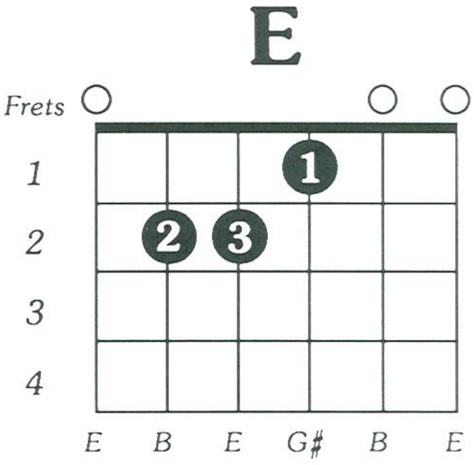


Figure 16: Finger Position of E Major Chord

The frequency of the first string was set to 82 Hz representing the lowest E note. The second string frequency was set to 123 Hz representing a B note. The third string frequency was set to 165 Hz representing a higher E note. The fourth string frequency was set to 208 Hz representing a G# note. The fifth string frequency was set to 247 Hz representing a higher B note. The sixth string frequency was set to 329 Hz representing a high E note. The output of the summing operational amplifier simulation is shown in Figure 17. The result of this simulation shows the superposition of these six sinusoidal input waveforms. The resulting audio output has the periodic form that was expected.

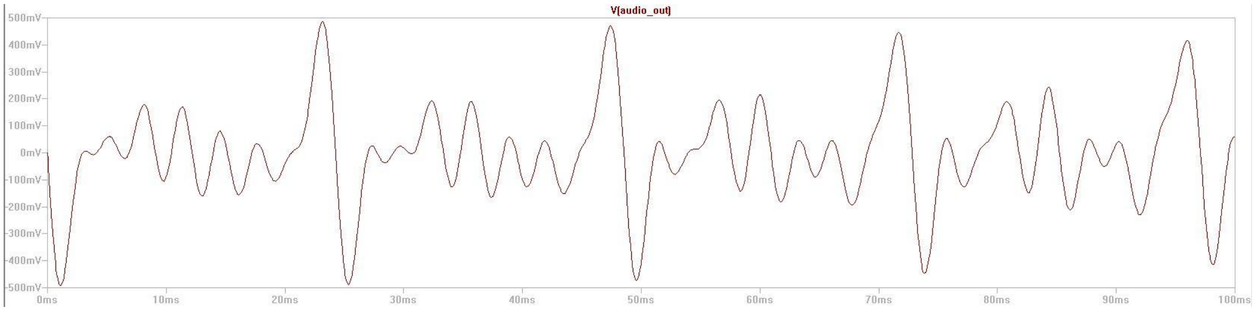


Figure 17: Summing Amplifier Audio Output Results

[ARB, NAH, DLM, KZY]

3.6 Schematics

3.6.1 Power Schematics

Figure 18 show the schematic for the power supply components of the stringless guitar. A 3.3V Texas Instruments Low-dropout linear regulator labeled **LR1** is used to regulate the 9V battery supply to 3.3V. The datasheet for the linear regulator can be found in [10]. A 10uF capacitor (**C1 and C2**) was placed on the 9V and 3.3V to filter out some of the noise in the system. A 1A fast blow fuse along with a SPST switch was placed in between the battery to protect the circuit from shorts and to turn on and off the guitar. The 9V coming from the battery along with the 3.3V were used to power up the different components in the stringless guitar. Furthermore, the battery could be charged by using the 12VDC port on the battery and the charging adapter provided by the manufacture.

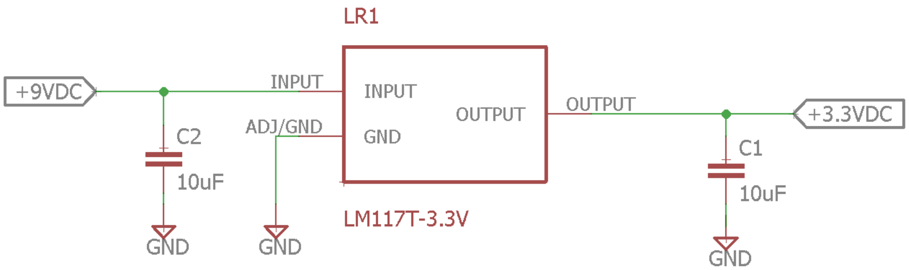


Figure 18: Power Schematic

[DLM]

3.6.2 Strum Sensor Schematic

To amplify the voltage produced by the piezoelectric strip, a UA741 operational amplifier is used in the circuit topology shown below in Figure 19. Resistors **R2**, **R3**, **R4** and **R5** set the DC bias of the Op-Amp to 4.5V. The resistance value **R1** and capacitance value **C1** were chosen experimentally to set the gain of the Op-Amp such that when the user strums hard, the output will produce around 3.3 Vp-p.

When the steel plate of the sensor is strummed and the piezo is flexed, a voltage is produced at the output of the Op-Amp that is proportional to the velocity of the piezo flexing. The output of the Op-Amp is AC coupled to the input of the ADC where the DC bias is set to 1.65V. There are one of these strum sensor OP-Amps for each of the six piezoelectric sensors.

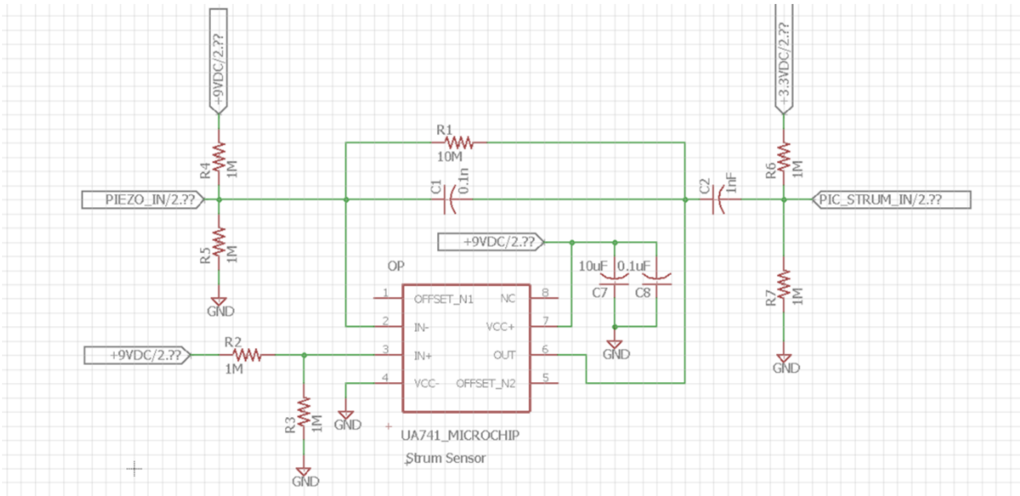


Figure 19: Piezoelectric Strip Operational Amplifier Circuit.

[NAH]

3.6.3 Fretboard Sensor Schematic and PCB Layout

Figure 20 shows the schematic of the Finger Position Sensor Board. The sensor board is connected to nine of the microprocessor's digital input/output pins. The input pins of the microprocessor will be tied to ground using a resistor of 12 MΩ, which is approximately three times the measured resistance of a human finger. The users' finger will act as a resistor and therefore complete the circuit creating a voltage that will be read as a digital high from the perspective of the microprocessor. To determine the fret being pressed, the microprocessor individually toggles pins 81 to 84 to a digital high of 3.3 VDC and proceeds to scan through input pins 76 to 80. When one of the frets is pressed, the corresponding pin is shown as a digital high, the microprocessor will exit the finger position sensor scan and continue to execute the rest of the program. If there are no frets detected, the microprocessor executes the finger position sensor scan and play the note of the open string. The only time the microprocessor will scan the sensors is after the triggering of the strum indicator. The Finger Position Sensors are connected to the main processor board through a connector of ten pins, shown as **Con3** in Figure 20.

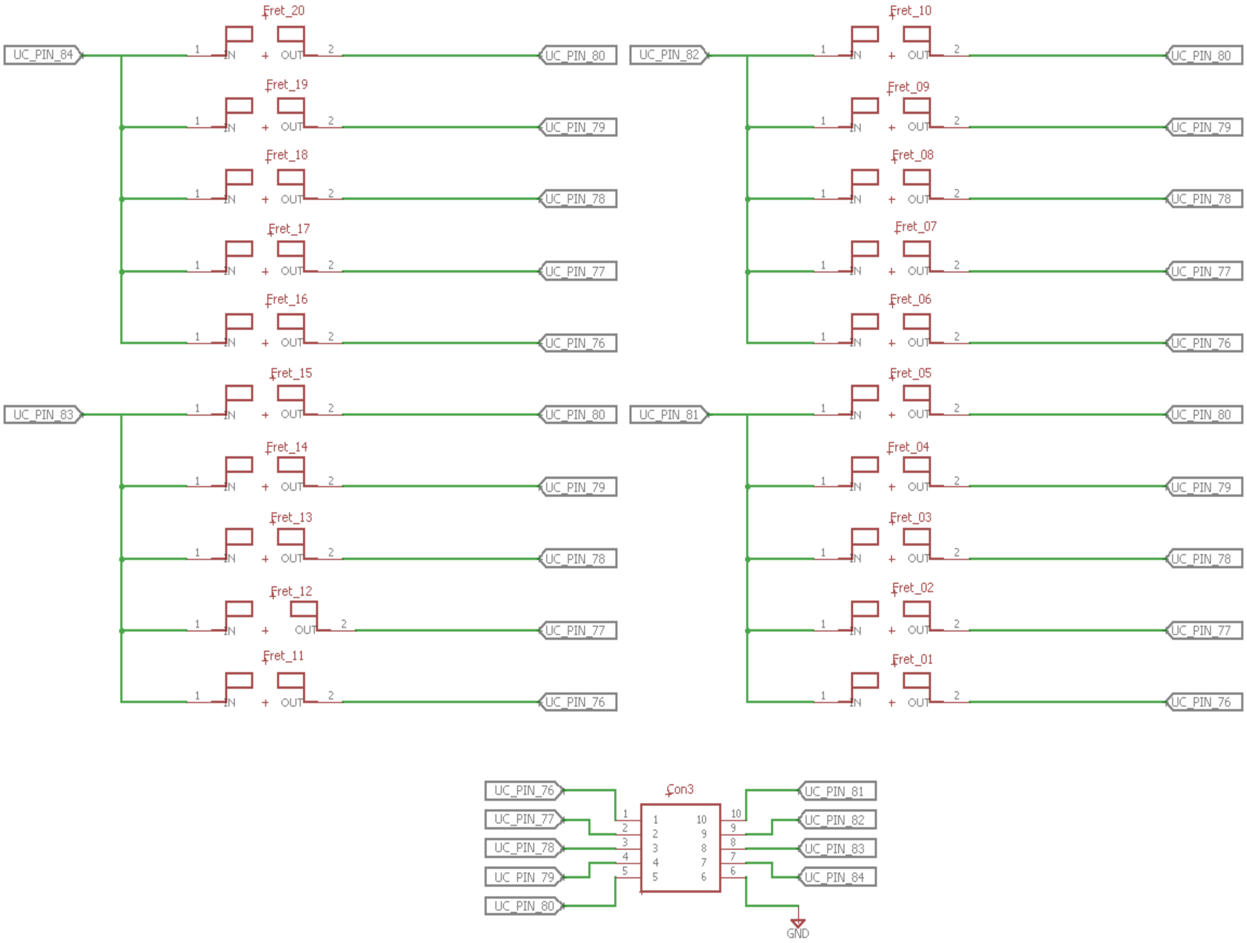


Figure 20: Finger Position Sensor Schematic

Figure 21 shows the overall PCB layout of the Finger Position Senor Board while Figure 22 and Figure 23 shows the finer details. Figure 22 shows the portion of the circuit board located at the head of the guitar. A hole with a radius of 1.25 mm will be drilled out to mount the board to the guitar head. The spacing between frets 1 and 2 is 2.38 mm. The spacing between frets are uniform and therefore not labeled for each individual instance. The lengths of the frets are shown clearly in Figure 21. The fret lengths were determined by measuring the actual length of each fret on a real guitar. Figure 23 shows the portion of the board that is located nearest the body of the guitar. Each mounting hole has a radius of 1.25 mm and a two by five grid of through hole pads with a radius of 0.5 mm each. The through hole pads will be used for the mounting of the two by five connector, **Con3**.

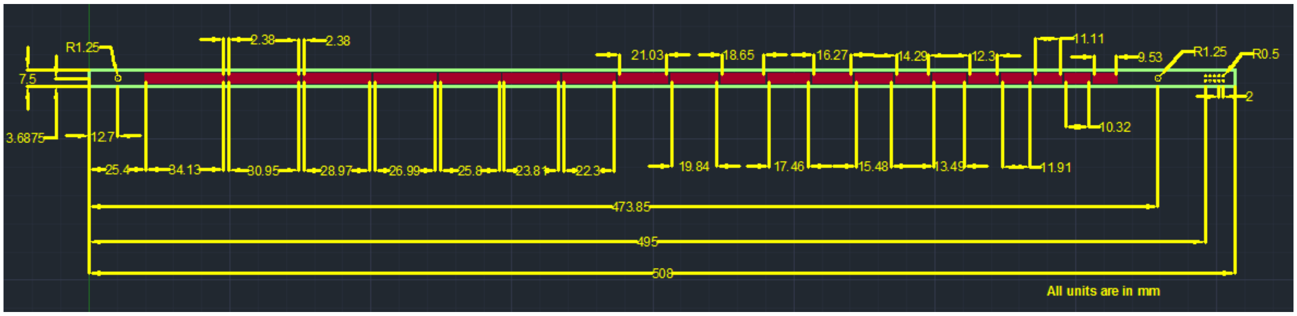


Figure 21: Drawing of the Finger Position Sensor PCB Layout

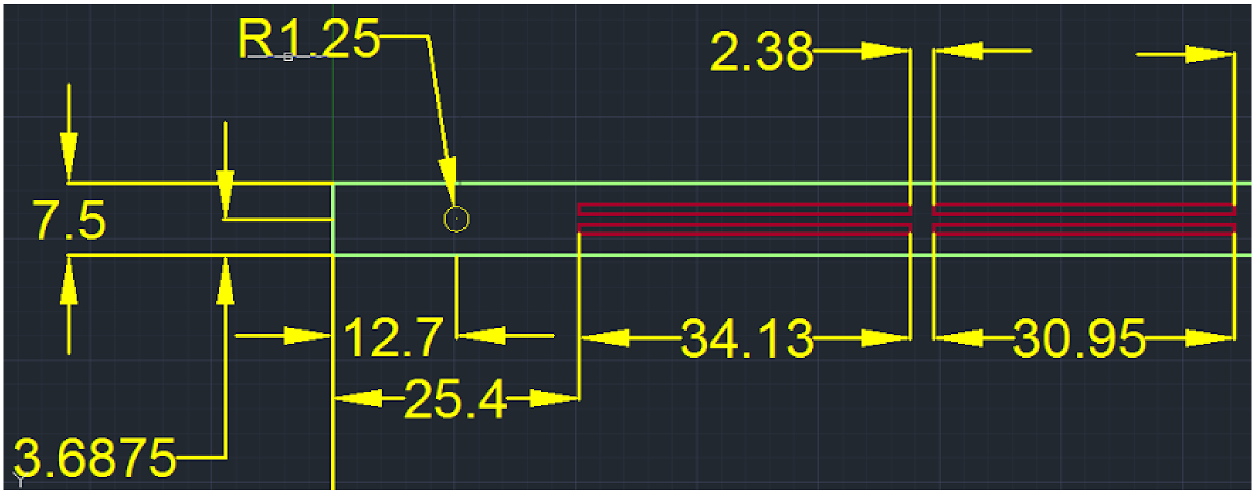


Figure 22: Drawing of the Left Half of the Finger Position Sensor PCB Layout

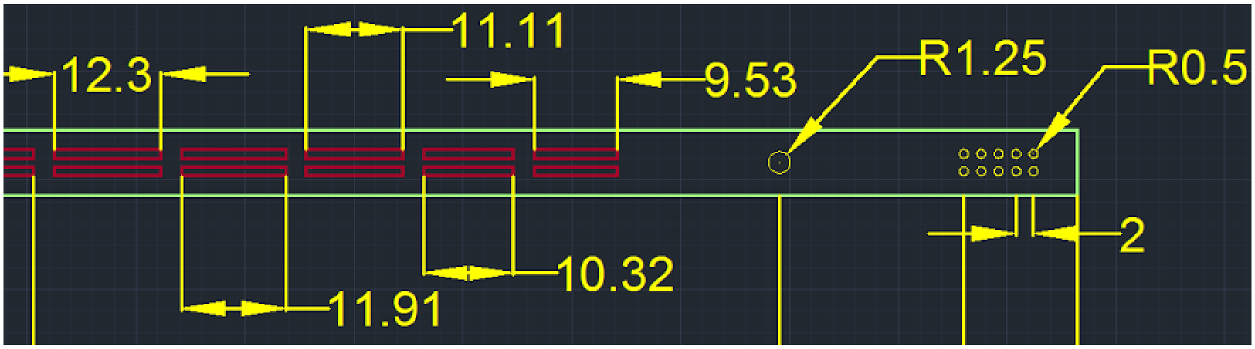


Figure 23: Drawing of the Right Half of the Finger Position Sensor PCB Layout

[ARB]

3.6.4 *Processing Board Schematic*

The schematic for the PIC32MX270F512L [11] is shown below in Figure 24. All the pins being used are labeled and can be seen more clearly in Table 15. All power pins are equipped with two decoupling capacitors, one value being 10 uF and the second being 0.1 uF. These capacitors were chosen to filter out any type of unwanted, high frequency spikes on the power rail. These capacitors help protect all the integrated circuits being used. The value of resistors R43 and R44 and capacitor C29 were chosen according to the manufacturers recommendations.

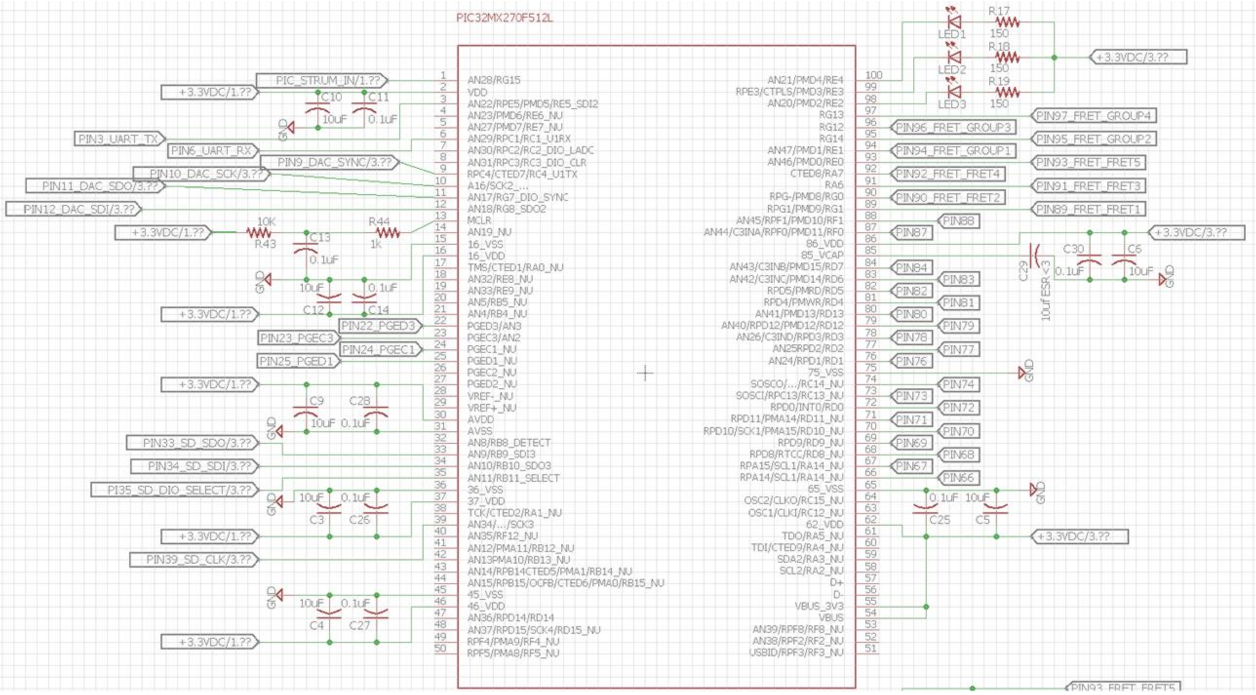


Figure 24: PIC32MX270F512L Schematic

[ARB]

3.6.5 *Schematic for External Connections*

The external connections can be seen in Figure 25 as J1, J2, J3, J4, and J5. The J1 connector handles the signals being sent to the Fret Sensor board. The resistors R8, R9, R10, R11, and R12 were chosen to be 20 MΩ after rounding up the minimum 12 MΩ. J2 and J3 were placed to make extra pins accessible if extra features need to be added. J4 is the connector that provides power to the integrated circuits, carries the DAC output to the summing amplifier shown in Figure 15, and connects the piezo sensor to the board. J5 allows for the programming pins to be accessible as well as the pins declared for UART communication. R13, R14, R15, and R16 are unpopulated allowing the user to choose the preferred pins. To use these pins, the pads will be shorted together.

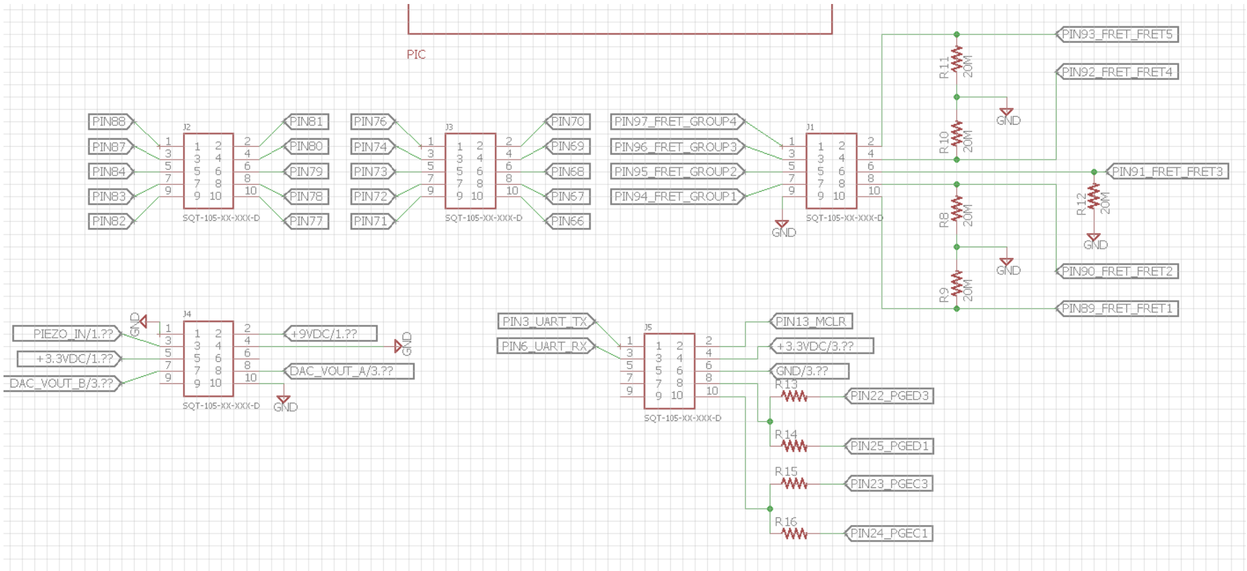


Figure 25: Header Connections

[ARB]

3.6.6 DAC and MicroSD Card Schematic

The MicroSD card and DAC schematics are shown in in Figure 26. R54, R55, R56, and R57 are used to keep the communication pins held high since the MicroSD card is an active low device. The DAC interfaces directly to the processor, requiring no additional resistors or capacitors.

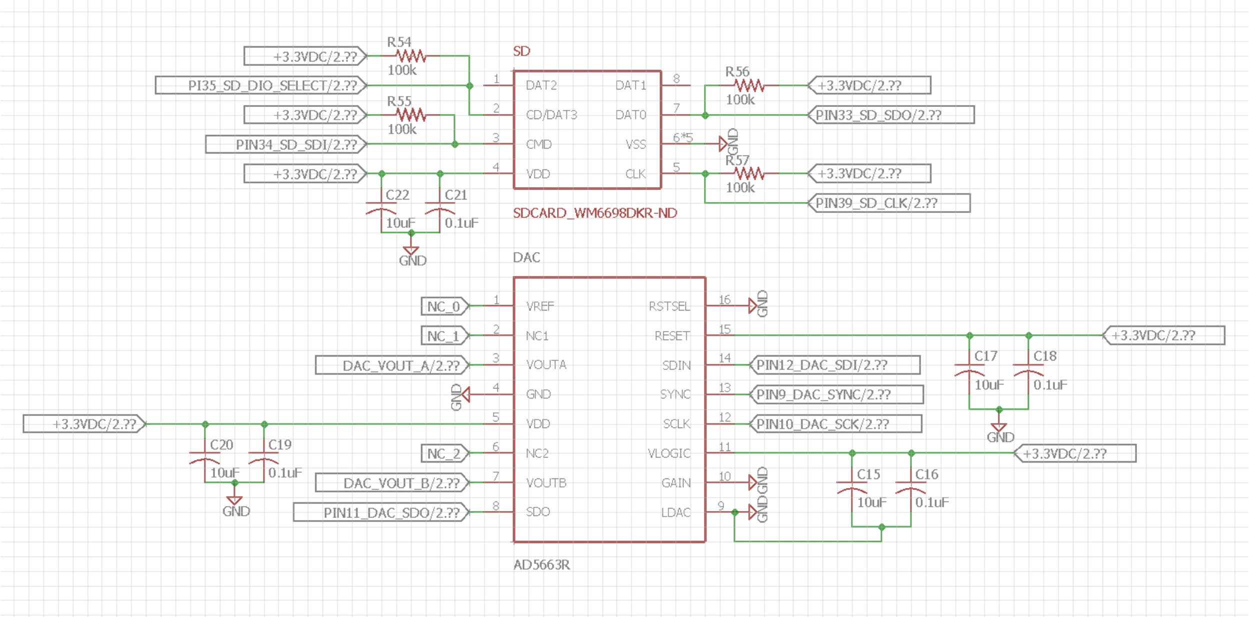


Figure 26: MicroSD Card and DAC Schematic

[ARB]

3.6.7 Processing Board PCB Layout

The main processing board is shown in Figure 27. The piezo electric operational amplifier circuit and the DAC are both placed relatively close to the connector J4. This allows the incoming and outgoing signal traces to be as short as possible. The MicroSD card is placed below the DAC and close to the edge of the board such that the hinge of the MicroSD card holder can fully open. The PIC was the placed as close as possible to the DAC and MicroSD card holder to allow for short traces but still allowing space for decoupling capacitors. The programming header J5 is placed close to the programming pins and the Fret Sensor header J1 was placed close to the respective pins. J2 and J3 were placed close to the edge of the board as to not be in the way of any other signals. The 3.3 VDC trace was intentionally made much larger than all other traces to allow for more current to be drawn if needed. All other traces were then routed in such a way that allowed every ground point on the board to have a direct path to the ground pin. The LEDs were placed in the top right corner of the board and four mounting holes were placed in all four corners of the board. Most vias that were placed ended up being covered up by mask, however, select vias were left uncovered acting as a test point.

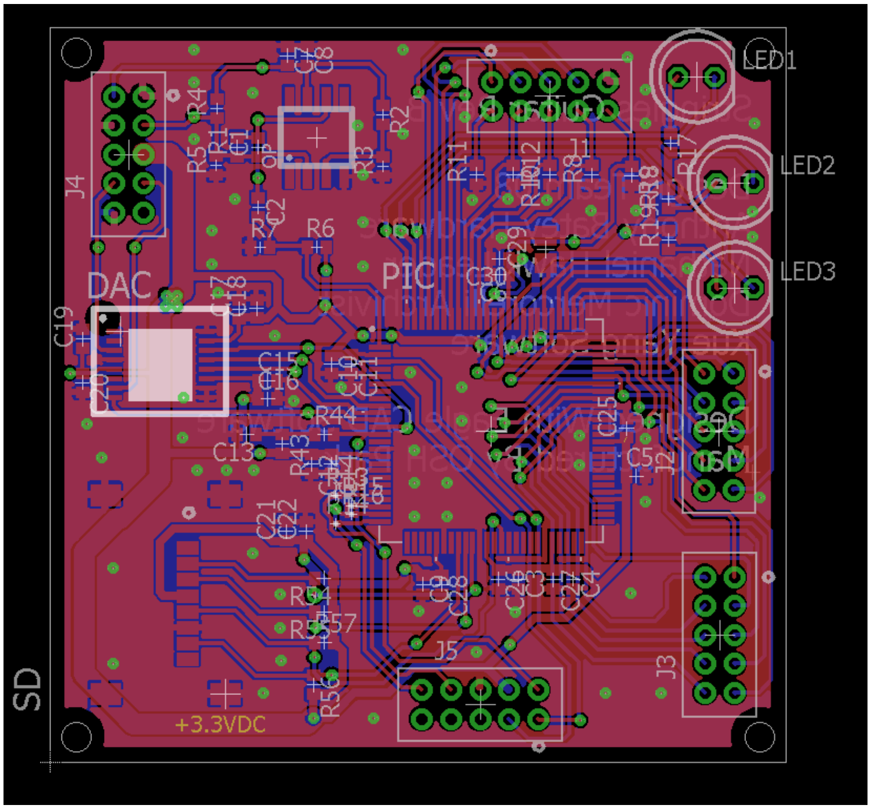


Figure 27: PCB Layout of Main Processing Board

[ARB]

3.6.8 Summing Amplifier Schematic

The summing amplifier schematic is shown in Figure 28. The summing amplifier is used to sum the six audio signals received from the six DACs. The rails of the amplifier will range from 0 V to +9 V and the amp is biased at 4.5 V. The resistor, labeled R1, could be changed to adjust the gain of the amplifier. The gain of the amplifier is given as

$$\boldsymbol{A_V = \frac{V_{OUT}}{V_{IN}} = \frac{R_f}{R_{IN}}} \tag{14}$$

where $\boldsymbol{R_f}$ is the feedback resistor, R1, and $\boldsymbol{R_{IN}}$ is the resistance of each branch. The resistance in the each of the six branches will be the same. For this application, the $\boldsymbol{R_{IN}}$ is 10 kΩ and $\boldsymbol{R_f}$ is 5 kΩ which means that the gain of the system is 0.5 [V/V]

A voltage divider is used to bias the op-amp between the rails to improve amplification of the input signal. A resistance of 1 MΩ was chosen for **R2** and **R3** to reduce the amount of current in the feedback loop and in the voltage divider. Capacitors (**C3, C7, C8, C9, C10, C11 and C12**) is used as a DC block to remove DC biasing at the output and inputs of the op amp. A simple RC network and voltage divider was placed after capacitor **C3** like seen in Figure 28 for tone control and volume control before the audio out was sent to the guitar amplifier through a ¼ female audio out connector.

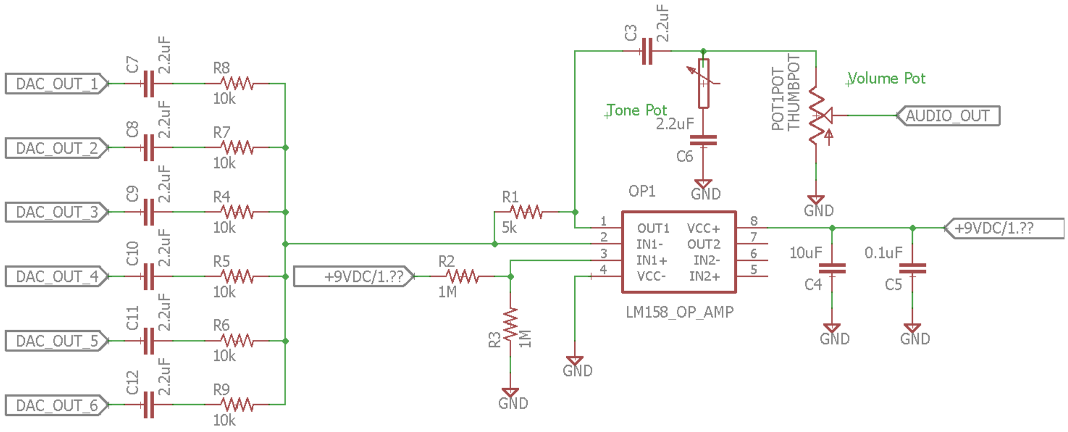


Figure 28: Summing Amplifier Schematic

[ARB, NAH, DLM, KZY]

3.7 Software Components

3.7.1 PIC Processor

The microprocessor used is Microchip's PIC32MX270F512L [11]. The PIC is a 100-pin microprocessor thatis configured for 40 MHz. The PIC has 48, 10-bit analog inputs, four UART modules, five Timer modules and four SPI modules. The application requires one UART module, two Timer modules and two SPI modules. Each module configurations (e.g. bus speed) are described in the pseudocode below. The pinout for the pic is listed in Table 15.

Table 15: PIC Pin Layout

PIN #	Function	Input/ Output	Connected Device	PIN #	Function	Input/ Output	Connected Device
1	AN28	OUT	Strum_Sensor	**51**	IO	IN/OUT	*Not Used*
2	VDD		+3.3V	**52**	IO	IN/OUT	*Not Used*
3	U1TX	OUT	UART_TX	**53**	IO	IN/OUT	*Not Used*
4	IO	IN/OUT	*Not Used*	**54**	VBUS		VDD
5	IO	IN/OUT	*Not Used*	**55**	VBUS_3V3		VDD
6	U1RX	IN	UART_RX	**56**	D-		*Not Used*
7	IO	IN/OUT	*Not Used*	**57**	D+		*Not Used*
8	IO	IN/OUT	*Not Used*	**58**	IO	IN/OUT	*Not Used*
9	DIO_SYNC	OUT	DAC_SYNC	**59**	IO	IN/OUT	*Not Used*
10	SCK2	OUT	DAC_SCK	**60**	IO	IN/OUT	*Not Used*
11	SDI2	OUT	DAC_SDO	**61**	IO	IN/OUT	*Not Used*
12	SDO2	IN	DAC_SDI	**62**	VDD		+3.3V
13	MCLR		Master Clear	**63**	IO	IN/OUT	*Not Used*
14	IO	IN/OUT	*Not Used*	**64**	IO	IN/OUT	*Not Used*
15	IO	IN/OUT	*Not Used*	**65**	VSS		GND
16	IO	IN/OUT	*Not Used*	**66**	IO	IN/OUT	*Not Used*
17	IO	IN/OUT	*Not Used*	**67**	IO	IN/OUT	*Not Used*
18	IO	IN/OUT	*Not Used*	**68**	IO	IN/OUT	*Not Used*
19	IO	IN/OUT	*Not Used*	**69**	IO	IN/OUT	*Not Used*
20	IO	IN/OUT	*Not Used*	**70**	IO	IN/OUT	*Not Used*
21	IO	IN/OUT	*Not Used*	**71**	IO	IN/OUT	*Not Used*
22	PGED3		Programmer_PGED	**72**	IO	IN/OUT	*Not Used*
23	PGEC3		Programmer_PGEC	**73**	IO	IN/OUT	*Not Used*
24	PGED1		Programmer_PGED	**74**	IO	IN/OUT	*Not Used*
25	PGEC1		Programmer_PGEC	**75**	VSS		GND
26	IO	IN/OUT	*Not Used*	**76**	IO	IN/OUT	*Not Used*
27	IO	IN/OUT	*Not Used*	**77**	IO	IN/OUT	*Not Used*
28	IO	IN/OUT	*Not Used*	**78**	IO	IN/OUT	*Not Used*
29	IO	IN/OUT	*Not Used*	**79**	IO	IN/OUT	*Not Used*
30	AVDD (2)		+3.3V	**80**	IO	IN/OUT	*Not Used*

31	AVSS (2)		GND	**81**	DIO8	OUT	Fret_15, Fret_14, Fret_13, Fet_12, Fret_11
32	DIO_DETECT	IN	SD_CARD_DETECT	**82**	IO	IN/OUT	*Not Used*
33	SDI3	OUT	SD_CARD_SDO	**83**	IO	IN/OUT	*Not Used*
34	SDO3	IN	SD_CARD_SDI	**84**	IO	IN/OUT	*Not Used*
35	DIO_SELECT	OUT	SD_CARD_SELECT	**85**	VCAP (3)		GND
36	VSS		GND	**86**	VDD		+3.3V
37	VDD		+3.3V	**87**	IO	IN/OUT	*Not Used*
38	IO	IN/OUT	*Not Used*	**88**	IO	IN/OUT	*Not Used*
39	SCK3	OUT	SD_CLK	**89**	DIO1	IN	Fret_01, Fret_06, Fret_11, Fret_16
40	IO	IN/OUT	*Not Used*	**90**	DIO2	IN	Fret_02, Fret_07, Fret_12, Fret_17
41	IO	IN/OUT	*Not Used*	**91**	DIO3	IN	Fret_03, Fret_08, Fret_13, Fret_18
42	IO	IN/OUT	*Not Used*	**92**	DIO4	IN	Fret_04, Fret_09, Fret_14, Fret_19
43	IO	IN/OUT	*Not Used*	**93**	DIO5	IN	Fret_05, Fret_10, Fret_15, Fret_20
44	IO	IN/OUT	*Not Used*	**94**	DIO6	OUT	Fret_05, Fret_04, Fret_03, Fret_02, Fret_01
45	VSS		GND	**95**	DIO7	OUT	Fret_10, Fret_09, Fret_08, Fret_07, Fret_06
46	VDD		+3.3V	**96**	IO	IN/OUT	*Not Used*
47	IO	IN/OUT	*Not Used*	**97**	DIO9	OUT	Fret_20, Fret_19, Fret_18, Fret_17, Fret_16
48	IO	IN/OUT	*Not Used*	**98**	DIO21	OUT	LED_ON
49	IO	IN/OUT	*Not Used*	**99**	DIO22	OUT	LED_ERROR
50	IO	IN/OUT	*Not Used*	**100**	DIO23	OUT	LED_No_SD

Notes:

1. Analog and Digital VDD and VSS pins are coupled with a 0.1uF capacitor.
2. Analog VDD and VSS are connected to Digital VDD and VSS pin.
3. A 10uF Tantalum or ceramic capacitor connected in series.
4. Pins 76-80 will have a pull-down resistor so that the pins would not be floating.

[ARB, DLM, KZY]

3.7.2 Digital to Analog Converter

The digital to analog converter used is Analog Devices’ AD5689R. The AD5689R has two 16-bit DACs channels that can be configured via SPI. The AD5689R can have a clock speed up to 50 MHz and runs on a power supply ranging from 2.7 V and 5.5 V. The input is 16-bits, mono audio data and will only require one channel for the output [12]. The pinout of AD5689R is shown in Figure 29.

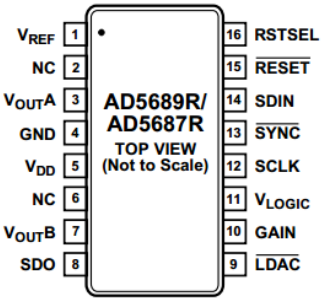
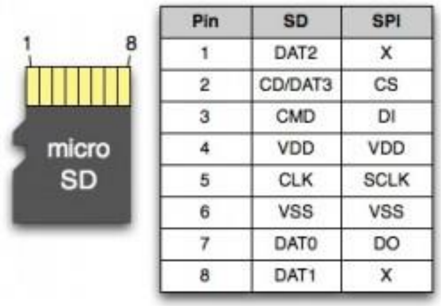


Figure 29: AD5689R DAC Package and Pin Layout

[KZY]

3.7.3 External Memory

The external memory used for the project is a microSD card because the main memory of the pic is small and the number of files that is required is large. The memory size required is approximated at 21 MB per pic as discussed in earlier sections. The application will interface with the microSD card via SPI and is configured using the Fatfs Library [13]. The pinout for microSD card using SPI is given in Figure 30.



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

Figure 30: MicroSD card pinout [14].

[KZY]

3.7.4 Sensor Algorithms

Figure 31 shows a timing diagram for the finger position sensor from the perspective of the microprocessor. The fret sensors are grouped into four groups of five frets. When scanning the frets, the pic will set each group high and then check if any of the frets within the group is also set high to determine which fret was pressed. The microprocessor will scan down each group until it finds a fret that was pressed. If no frets are pressed, the application will assume the user is playing an "open" string.

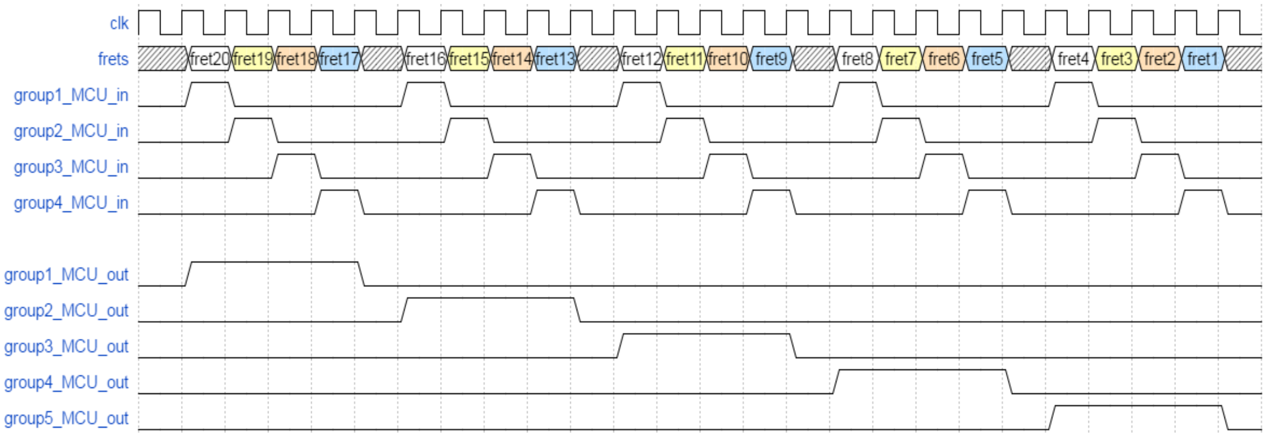


Figure 31: Finger Position Sensor Timing Diagram

The strum sensor uses an algorithm to determine how hard a user strums. The voltage at the input of the ADC after the user strums resembles a damped sinusoid. The damped sinusoid shown in Figure 32 shows a theoretical signal that would be expected if the user strums hard.

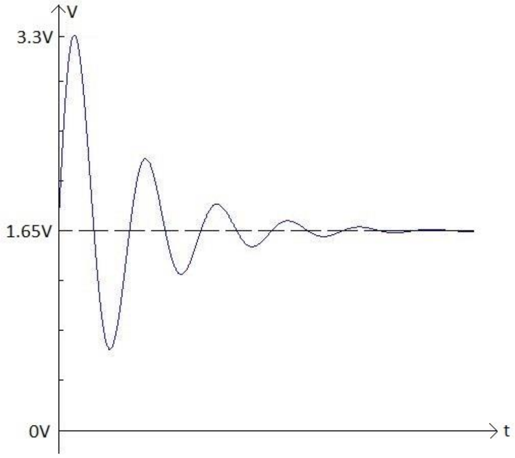


Figure 32: Theoretical Damped Sinusoid

The following algorithm detects local maximums relative to the 1.65V level.

The ADC continually samples (about one hundred times per 1.65V) crossing and it keeps a variable called tempmax. It continually updates the value of tempmax if the new value is greater than the current value. Once the new value is lower than the value in tempmax, the code reports that the local max average must be performed. Four of the samples just before this latest tempmax sample have been saved as well. The four previous samples and the current tempmax sample are doubled, added together and the result is divided by ten. The result of this calculation is stored in a local max array. This averaging is done because the real signal is not quite as clean as a theoretical damped sinusoid.

The algorithm detects 1.65V crossing and it uses this to clear the variable tempmax. Local maximums are repeatedly calculated like this.

The complement to this part of the algorithm is implemented to detect local minimums relative to the 1.65V level.

Each time a local maximum or local minimum is detected, the code preforms the new strum check. The magnitude of the new local maximum (or minimum) is calculated and compared with the average of the previous two local peak magnitudes. If the new peak magnitude is greater than the average of the previous two peak magnitudes plus a small threshold (0.1V), then a new strum is detected. When a new strum is detected, the sound file begins to get written to the DAC with a scaling factor determined from the most recent peak magnitude.

There are a few additional thresholds built in as well. Local peaks below 0.5V in magnitude are not considered. This is the noise threshold. The electrical noise is from the Op-Amp and Piezoelectric flex sensor circuitry.

Once a new strum is detected, a second strum can’t be detected for the time duration that it takes to detect at least two local peaks. The local peaks are still calculated though. This is because the local peak magnitude arrays are initialized to zero after each strum.

[ARB, NAH, KZY]

3.7.5 Sound Files

Sound files were sampled from a Fender Squire guitar. The sampling was done as follows. A Focusrite Scarlett Solo shown in Figure 33 was used to record the sound files. The three-state switch of the Fender Squire was set to the middle state, which enabled both of the single coil pickups. Volume and tone nobs of the guitar were turned all the way up. The guitar was strummed directly above the pickup closest to the bridge of the guitar with a Fender medium pick. The Fender Squire had a new set of D’Addario XL nickel wound super light gauge EXL120-3D strings and the guitar was set up and very well in tune.

Figure 33: Focusrite Scarlett Solo

The Scarlett Solo was able to sample the sound files at 196 kHz, 24 bit. One string was recorded at a time, but once recording a string every tone of that string was recorded in one session. Ableton Live sound engineering software was used to record and split up each total string sound file into small sound files containing each individual tone. A library of sound files was created for each string separately and each string library was loaded to a separate microSD card.

No effects were used to change the sound files from their raw form, but the amplitude of the files were normalized using an Ableton exporting feature so that each tone was the same volume. The sound files were exported as .WAV file format, 48kHz 16bit. This format is compatible with the PIC32 and the DAC that was selected for the stingless guitar. The 196 kHz, 24-bit audio files sound excellent relative to most any digital recording you can find today.

[NAH]

3.7.6 Pseudocode

The pseudocode for the main modules used are presented. The main modules are listed as follows: Main, IO, TIMER, ADC, SPI, UART and Audio. The tables corresponding to the modules listed above are Table 16, Table 17, Table 18, Table 19, Table 20, Table 21 and Table 22, respectively. Parts of the pseudocode algorithm for processing audio files from external memory were based on an example project from [15]. The application requires the Microchip’s Legacy Peripheral Libraries [16] and the Fatfs Library [13]. The Fatfs library is used to interface with the external memory while the peripheral library is used to configure some of the peripheral modules.

The Main module is shown in Table 16. The module is the main entry point for the entire application. The module is responsible for configuring the microprocessor (e.g. system clock), initializing peripherals (e.g. timers) and running the main processes of the application. The Timeout module discussed in earlier sections is the watchdog timer, which is a hardware timer built into the pic processor. When the watchdog timer is enabled, the timer will set a flag prior to running the application. In the main process loop, the flag is cleared in software, indicating that there were no stalls and then resets the flag. If the timer overflows and the flag has not been cleared within the configured period, the pic will assume a stall has occurred during runtime and resets the application from the very beginning.

Table 16: Main Module (Main Entry Point for the Application).

Main Module

```
INCLUDE Microchip_Peripheral_Library
INCLUDE IO_Module
INCLUDE TIMER_Module
INCLUDE ADC_Module
INCLUDE SPI_Module
INCLUDE Audio_Module

/* Clock Configurations: */
SYS_CLK = 40 MHz
PB_CLK = 40 MHz

/* Watch dog Configurations, assumes a 32 KHz clock speed. */
Watch_Dog_Timer = ENABLED
Watch_Dog_Timer_Prescalar = 1:2024  // Watch dog timer will time out after 2 seconds

Main:
    /* Initializes all peripherals. */
    Initialize_IO;        // Initializes all digital and analog IOs
    Initialize_TIMER;     // Initializes the TIMER module
    Initialize_ADC;       // Initializes the ADC module.
    Initialize_SPI;       // Initializes the SPI module.
    Initialize_UART;      // Initializes the UART module.
    Initialize_AUDIO;     // Initializes the SD_Card and DAC modules.

    /* Main Process */
    While(1):
        Clear_Watchdog_Timer();     // Clears the watch dog timer.
        AUDIO_Process();            // Processes audio data.
```

The IO module is shown in Table 17. The IO module is responsible for initialize all the pins that are used for each of the peripherals (e.g. ADC). The module will also handle scanning the fret sensors.

Table 17: IO Module.

IO_Module

```
int currentFret;          // Variable used to store the fret that is pressed.
boolean isProcess;        // Flag used to indicate that a fret press has been processed.

Initialize_IO():
        Analog_IO_Init;     // Initialize all Analog IO, used for ADC.
        Digital_IO_Init;    // Initialize all Digital IO, used for fret inputs.
        UART_IO_Init;       // Initialize IO for UART
        SPI_IO_Init;        // Initialize IO for SPI

IO_ReadSensors():

        currentFret = 0;    // Sets the current fret to 0, which means no frets has been pressed.

        /* Scans each group for a fret press and sets the current fret to the pressed fret. */
        currentFret = readGroup1Sensors();
        IF currentFret EQUAL 0 THEN
                currentFret = readGroup2Sensors();
        IF currentFret EQUAL 0 THEN
                currentFret = readGroup3Sensors();
        IF currentFret EQUAL 0 THEN
                currentFret = readGroup4Sensors();
```

The TIMER module is shown in Table 18. The module handles creating delays and writing audio data to the DAC. The module utilizes interrupts to process data periodically. The interrupts are enabled and disabled in the ADC and Audio modules.

Table 18: TIMER Module.

```
TIMER_Module
INCLUDE Audio_Module

Ms_Tick;         // Used to store the millisecond count.

Initialize_TIMER():
        Initialize_TIMER1();     // Initializes the TIMER1 module.
        Initialize_TIMER3();     // Initializes the TIMER3 module.

Initialize_TIMER1():
        TIMER1_ON_OFF = OFF;                // Turns off the TIMER1 module.
        TIMER1_PERIOD = 1_ms;               // Initialize the period for 1 ms.
        TIMER1_Interrupt = ENABLED;         // Enable the TIMER1 interrupt.
        TIMER1_ON_OFF = ON;                 // Turns on the TIMER1 module.

Initialize_TIMER3():
        TIMER3_PERIOD = 22_µs;              // Initialize the period for 22 µs.
        TIMER3_Interrupt = ENABLED;         // Enable the TIMER3 interrupt.
        TIMER3_ON_OFF = OFF;                // Turns off the TIMER3 module. Will be enabled in the ADC Module.

MSecond_Delay(delay) :
        msCount = Ms_Tick;                  // Stores the current millisecond count.
        while(Ms_Tick – msCount < delay);   // Wait until the delay is over.

Timer1_Interrupt():
        Ms_Tick = Ms_Tick + 1;              // Increments the millisecond count.
        Clear_Timer1_Interrupt_Flag();      // Clears the TIMER1 interrupt flag.

Timer3_Interrupt():
        Write_Data_To_DAC();                // Writes the audio to the DAC at the sampling rate of the original audio file.
        Clear_Timer3_Interrupt_Flag();      // Clears the TIMER3 interrupt flag.
```

The ADC module is shown in Table 19. The ADC module is responsible for reading the strumming sensor and starting the audio process as needed. The ADC interrupt checks for a voltage change read from the strumming sensor. If a change is detected, the application will assume that the user is strumming and start scanning the frets and the audio process.

Table 19: ADC Module.

ADC_Module

```
Initialize_ADC():
    ADC_ON_OFF = OFF;                                               // Turns off the ADC module.
    ADC_Channels = 1;                                               // Using only one ADC channel per processor
    ADC_Acquisition_Time = 3_µs                                     // Configuring sample acquisition time to 3 µs
    ADC_Interrupt_Config = Interrupt_When_Conversion_Done;          // ADC interrupts after conversion.
    ADC_Interrupt = ENABLED                                         // Enables ADC interrupts
    ADC_ON_OFF = ON;                                                // Turn on the ADC module.

ADC_Interrupt():
    data = read_ADC_Channel();              // Reads the data from the ADC channel.
    IF (data > 0) THEN                      // If the ADC data is not zero, strumming has occurred.
        currentFret = IO_ReadSensors();     // Scans the fret sensors, called from the IO module.
        setCurrentFile(currentFret);        // Sets the corresponding audio file
        Enable_Timer3();                    // Enables Timer3, which will start reading the audio file periodically.
```

The SPI module is shown in Table 20. Two SPI modules are used for the entire application. The first SPI module is used for interfacing with the external memory and is configured using the Fatfs library. The pseudocode shown below is used to interface with the DAC.

Table 20: SPI Module.

```
SPI_Module
Initialize_SPI():
        SPI2_ON_OFF = OFF;                         // Disables the SPI2 module.
        SPI2_Baud_Rate_Source = PBCLK;             // Sets the baud rate source to the peripheral clock
        SPI2_Baud_Rate = 8 MHz;                    // Sets the baud rate to 8 MHz
        SPI2_ON_OFF = ON;                          // Enable the SPI2 module.

SPI_ReadWrite(data) :
        dummy = SPI2_BUF;                          // Clears the SPI shift register.
        SPI2_BUF = data;                           // Write data to the SPI shift register.
        While(SPI2_Not_Done_Transmitting);         // Wait until transmission is done.
        dummy = SPI2_BUF                           // Read data from the shift register.
        return dummy;                              // Returns the read data
```

The UART module is given in Table 21. The module is responsible for processing commands received via serial communications. The module is used mainly for debugging and testing purposes only. The command module has the lowest priority out of all the modules discussed. The list of supported commands can be found in the Appendices under

UART ***Commands***.

Table 21: UART Module (Command Module).

UART_Module

```
INCLUDE IO_Module
INCLUDE TIMER_Module
INCLUDE ADC_Module
INCLUDE SPI_Module
INCLUDE Fatfs_Library

commandList =[
        {cmd1Name, cmd1Description, cmd1Handler},
        {cmd2Name, cmd2Description, cmd2Handler},
        etc…
]   // Stores the list of available commands that can be ran.
receiveBuffer;    // Buffer stores the commands received by the UART Receive Interrupt Handler.

Initialize_UART():
        UART1_ON_OFF = OFF;                // Turns off the UART module.
        UART1_Baud_Rate = 19200            // Configures UART’s baud rate to 19200
        UART1_TR_RX = ENABLED              // Enable only the TX and RX UART pins
        UART1_Interrupt_Rx = ENABLED       // Enables UART Receive interrupts
        UART1_ON_OFF = ON;                 // Turns on the UART module.

UART_Process():
        IF isThereACommand() THEN
                UART_Process_Command();  // Verifies a command was received and processes it.


UART_Process_Command():
        Cmd = parseCommand();                // Parses the command and command arguments received by the PIC.
        IF isValidCommand(cmd) THEN
                runCmdHandler();             // If the received command is valid, run the command handler.

UART_sendCharacter(char) :
        UART1_TX = char;                            // Writes data to the UART TX register.
        While(UART1_Not_Done_Transmitting);   // Wait until the transmission is completed.

UART_Interrupt():
        IF isReceiveData() THEN                    // Checks if there is any received data. If there is data, store the data.
                data = UART_RX_BUFFER;        // Reads data from the UART receive buffer.
                storeData(data);                   // Store the data into the receive buffer.
        Clear_UART_Flag();                         // Clears the UART interrupt flag.
```

The Audio module is shown in Table 22. The module is responsible for setting up handlers to read data from external memory and writing data to the DAC. The Fatfs library and the DAC are initialized in this module as well. The audio process checks if the entire file has been read and if all the data has been written to the DAC. The audio process ends when all data has been written out to the DAC.

Table 22: Audio Module.

Audio Module

```
INCLUDE Fatfs_Library
INCLUDE SPI_Module
INCLUDE ADC_Module
INCLUDE IO_Module

currentFilePtr          // Stores a pointer to the current file that is opened.
fileHeaderData          // Object that stores the header information of the current opened file (data size, sample rate, etc.).
audioDataBuffer         // Buffer stores the audio data read from the file.
currentFret             // Stores the current fret being processed.
readByteCount           // Stores the byte count read from the audio file.
writeByteCount          // Stores the byte count written to the DAC.

Initialize_Audio():
    Initialize_File_System_Library();    // Initializes Fatfs Library.
    currentFile = file1;                 // Defaults to the current file to an “open string” note.
    currentFret = 0;                     // Sets the fret to the current fret press
    readByteCount = 0;                   // Sets the byte count read from the file.
    writeByteCount = 0;                  // Sets the byte count for writing to the DAC.
    Initialize_DAC();                    // Turns on the DAC.

Audio_Process():
    IF (isTimerON() && !isDoneReadingData()) THEN
        Read_Audio_From_File():

isDoneReadingData():
    IF (readByteCount >= fileHeaderData.dataSize)  THEN
        return TRUE;
    return FALSE;

isDoneWritingData():
    IF (writeByteCount >= fileHeaderData.dataSize) THEN
        return TRUE;
    return FALSE;

setCurrentFile(fret):
    currentFile = openFile(fret);     // Sets the current file to the corresponding fret.

Read_Audio_From_File():
    audioData = Read_File(currentFile);          // Read the audio data from the audio file.
    readByteCount = readByteCount + 1;           // Increments the read byte count.
```

```
Write_Data_To_DAC():
        writeByteCount = writeByteCount + 1;        // Increments the byte write count.
        SPI_ReadWrite(DAC_Cmd_Addr);                // Writes the command and address bits to the dac.
        SPI_ReadWrite(audioDataBuffer);             // Writes the data from the audioData buffer to the dac.
```

[KZY]

3.8 Completed Prototype

Figure 34 shows the complete prototype of the stringless guitar. The individual components are shown in Figure 35 through Figure 37.

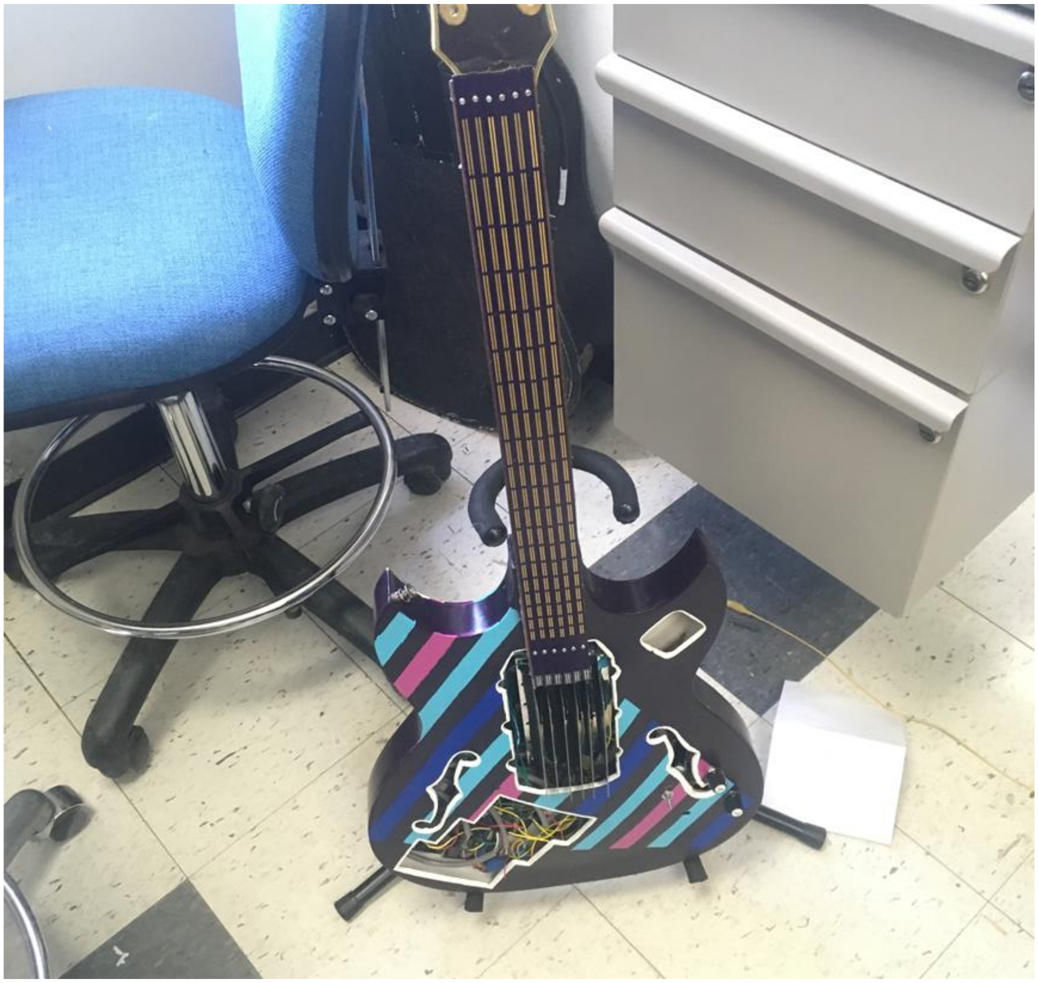


Figure 34: Completed Project Picture

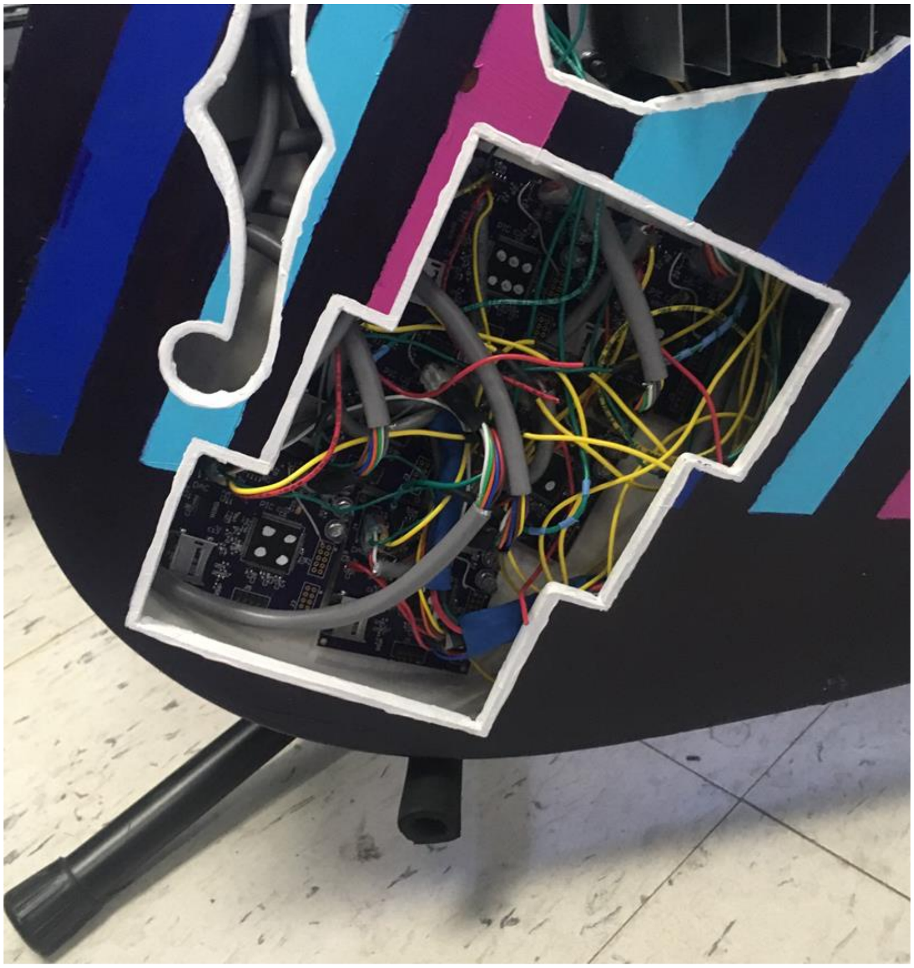


Figure 35: Picture of the PCBs in the Guitar

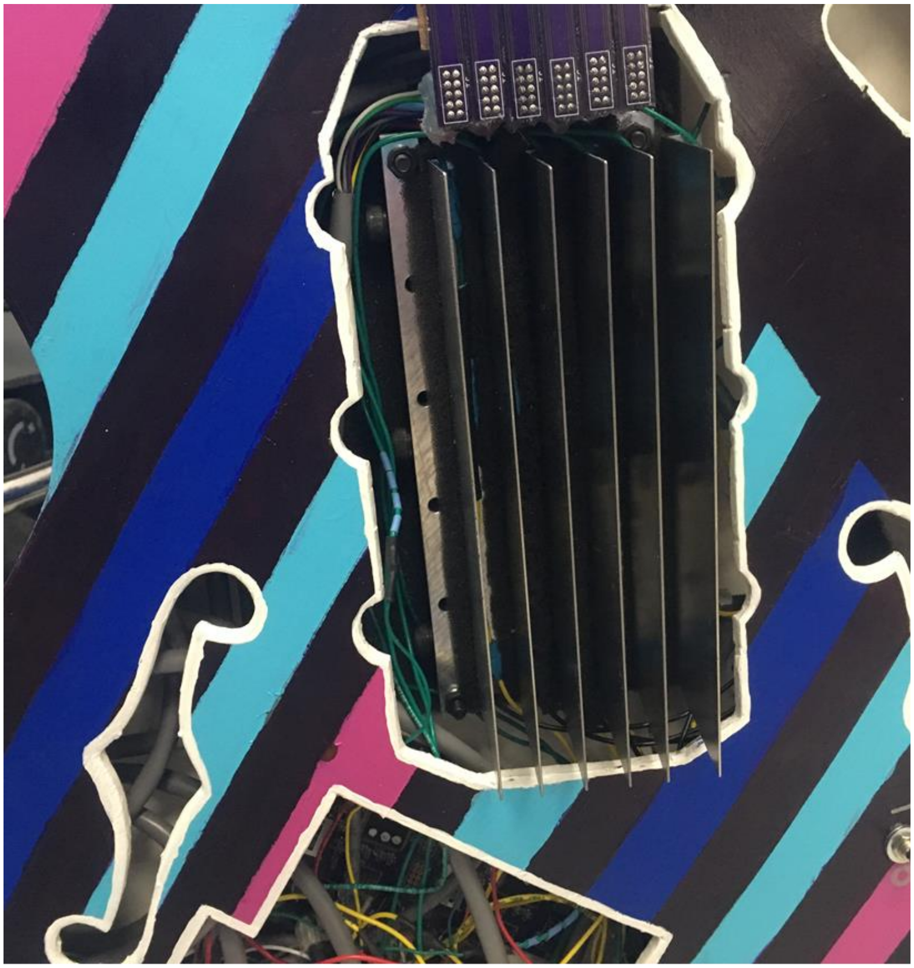


Figure 36: Picture of the Strum Sensors in Guitar

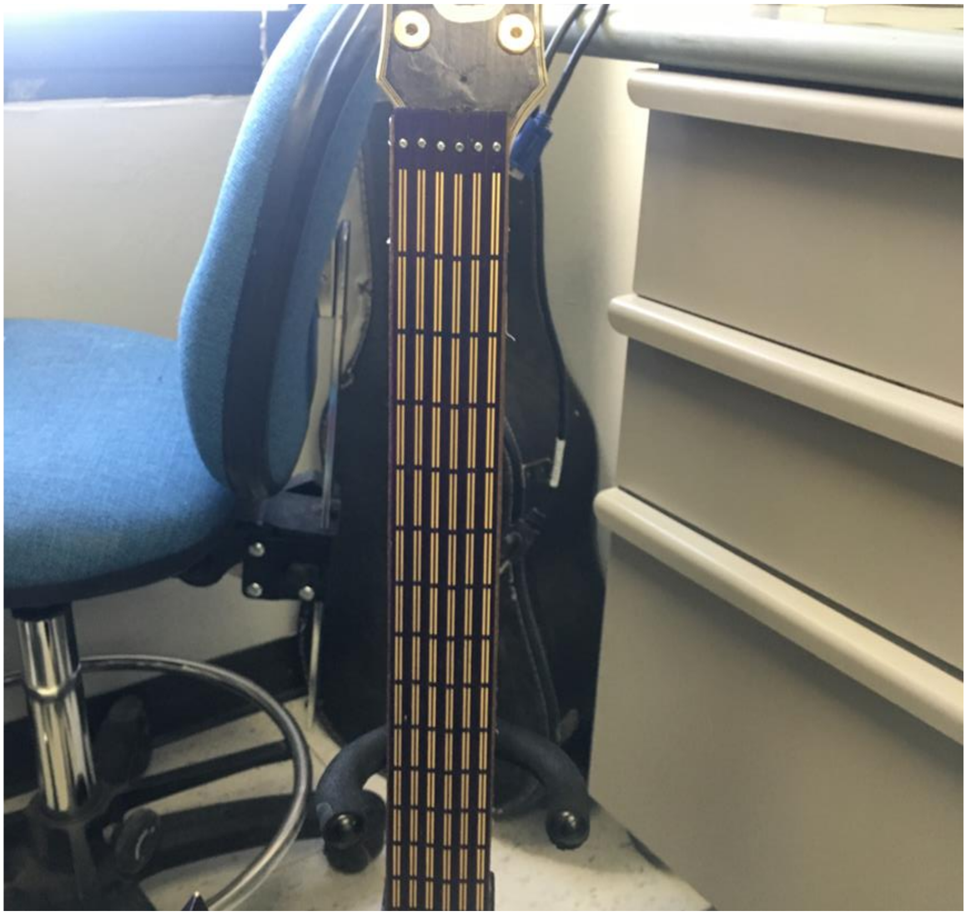


Figure 37: Picture of the Fret Sensors on the Guitar

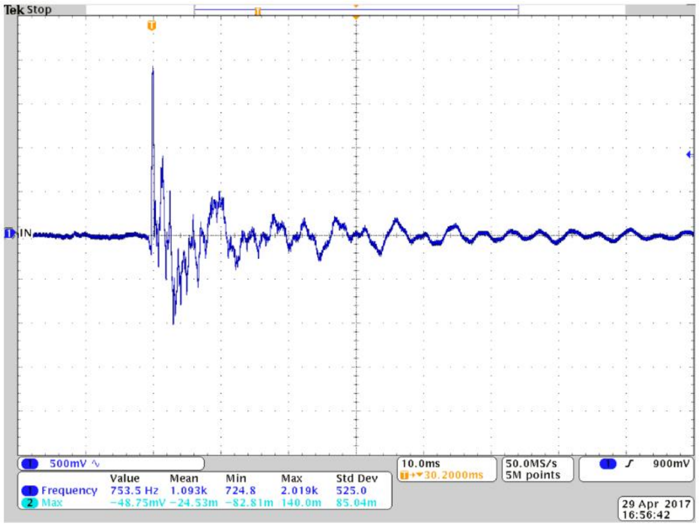
[NAH, DLM]

4 Testing Procedures

4.1 Strum Sensor Testing

To test the strum sensor, the stringless guitar is powered on. The voltage at the input of the ADC is measured at the PIC_STRUM_IN node shown in Figure 19.

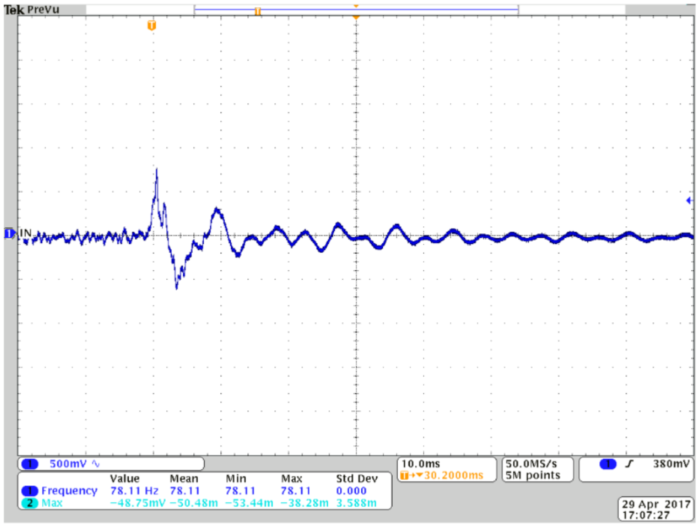
For a hard strum, the measured voltage should have a maximum peak to peak voltage near 3.3V. Figure 38 shows a typical voltage waveform that results from a hard strum. The maximum peak to peak voltage of this waveform is about 3V.



	Value	Mean	Min	Max	Std Dev
1 Frequency	753.5 Hz	1.093k	724.8	2.019k	525.0
2 Max	−48.75mV	−24.53m	−82.81m	140.0m	85.04m

Figure 38: String 1 Hard Strum

For a soft strum, the measured voltage should be greater than the minimum threshold within the PIC32 that corresponds to a soft strum. This minimum threshold for a soft strum currently corresponds to 1V peak to peak voltage. Figure 39 shows a typical voltage waveform that results from a soft strum. The maximum peak to peak voltage of this waveform is about 1.2V.



	Value	Mean	Min	Max	Std Dev
1 Frequency	78.11 Hz	78.11	78.11	78.11	0.000
2 Max	−48.75mV	−50.48m	−53.44m	−38.28m	3.588m

Figure 39: String 1 Soft Strum

[NAH]

4.2 Fret Board Testing

A continuity was made by placing one probe on a single header pin and the second on every exposed pad on the board. The test should be performed while checking the PBC layout. If any pads are connected that should not be, there is something wrong with the board or there is a short somewhere. Furthermore, testing was performed by hooking the fret sensor up to the PIC32 [8] and using the fret sensor algorithm which is explain Sensor Algorithms section of the report, each fret was pressed down and confirmed by displaying the fret number the PIC32 saw on the computer screen through UART.

[ARB]

4.3 Processing PCB Testing

Before placing any components on the PCB a continuity check was made to make sure all traces are properly routed. The 3.3 VDC trace should be check first to ensure power will be delivered to the proper places. The GND plane should be checked afterwards to make sure nothing is left floating. Finally check to make sure the signal traces are going to the proper places. The continuity check was performed while consulting the PCB layout. Once the board was assembled, the PIC was programmed and test vias were probed by using the oscilloscope’s probes to make sure that the right information was being sent and received. The test vias that were probed were the data lines from the PIC32 to the DAC, the data lines from the SD Card to the PIC, the Strum Sensor output to the Analog Pin on the PIC, and the analog audio out of the DAC.

[ARB]

4.4 Summing Amp Testing

To see if the summing amp would work correctly on the bench, a 293.665 Hz sine wave and 440 Hz sine wave were inputted into the summing amp schematic shown in Figure 15. Both sine waves had amplitudes of 0.5VDC peak to peak. A LT-Spice simulation was taken and was compare to the actual results taken from the bench circuit. Figure 40 show the results taken from LT-Spice and Figure 41 shows the results from the bench. Comparing the results, the summing amp worked on the bench correctly like expected from simulation. The bench results amplitude match the simulated results amplitude. Furthermore, the wave forms for both the simulated and bench results are identical, thus confirming that the summing amp would be able to work correctly with the six DAC audio signals coming in to produce one audio signal out.

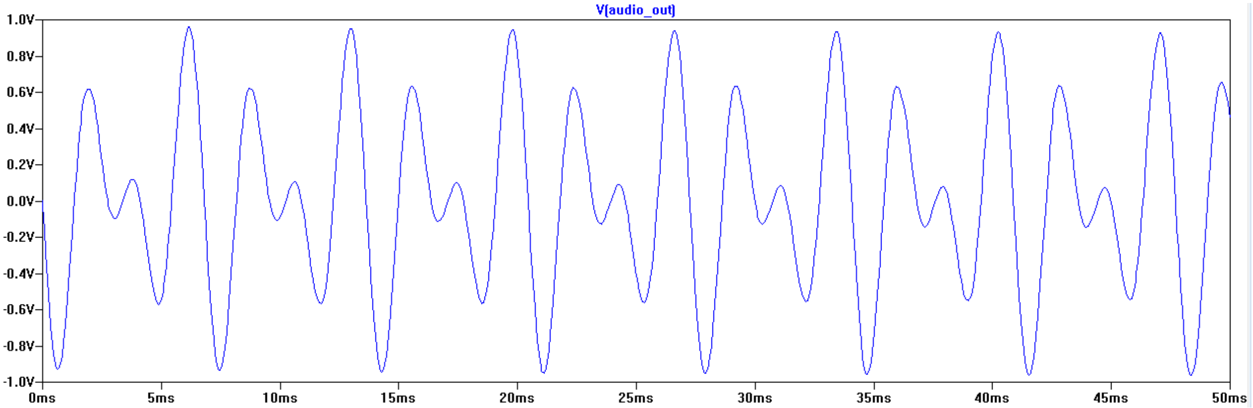
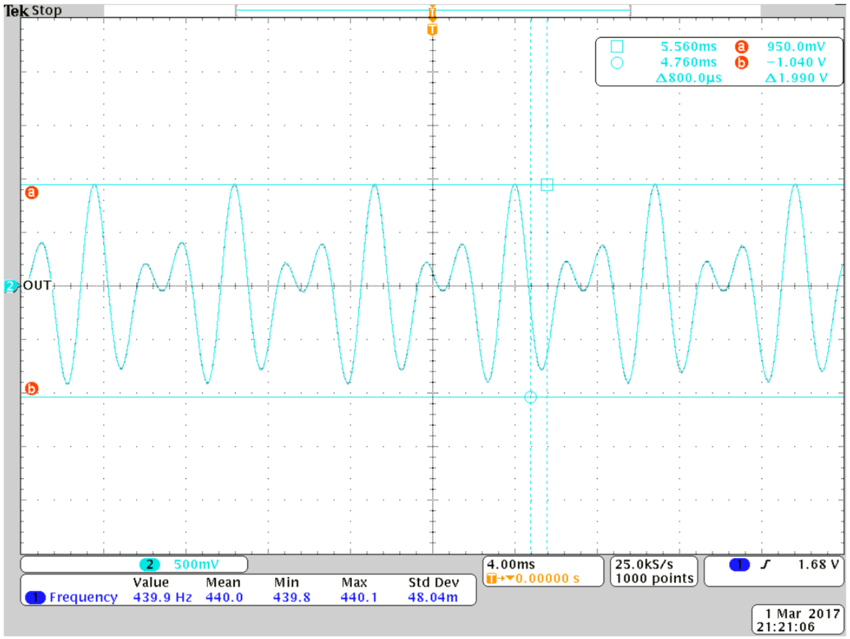


Figure 40: LT-Spice Simulation of Summing Amp



MSO4102B-L - 8:20:29 PM 3/1/2017

Figure 41: Bench Results of Summing Amp.

[DLM]

4.5 Power Testing

For testing the battery [17] , the [17] was turned on and using a voltage meter, a measurement of +12VDC, +9VDC and +5VDC were taken. Table 23 shows the actual and ideal results of the data collected for the different voltage supplies. The largest difference between the ideal and actual voltage reading was 0.33VDC for a percent error of 2.75%. For the +9VDC, which was the only power supply used off the battery for the design, there was a percent error of 1.22%. Which means that the actual voltage reading was close enough to ideal voltage reading to run the different components of the guitar.

Table 23: Ideal and Actual Voltages of Battery [17]

Ideal Voltage Readings	Actual Voltage Readings
+12VDC	+12.33VDC
+9VDC	+8.89VDC
+5VDC	+5.023VDC

Once the battery was tested, the +3.3VDC linear regulator [10] like seen in Figure 18 was set up on the bread board and tested. Once again, a voltage meter was used to measure the actual value of the linear regulator [10]. Table 24 contains the data collected from the voltage meter.

Table 24: Ideal and Actual Voltages of Linear Regulator [10]

Ideal Voltage Readings	Actual Voltage Readings
+3.3VDC	+3.28VDC

For the +3.3VDC, the percent error between the ideal and actual voltage reading is 0.6%. Which means that actual voltage of the +3.3VDC linear regulator, will be able to operate all the individual components of the stringless guitar. Furthermore, a measurement of the current was taken for one string of the guitar. When the PIC, strumming sensor, DAC, SD card reader, fret sensor is operating, the maximum currently that was measured was 50 mA. So, for all six strings a maximum current of 300 mA is needed to operate the guitar, which can be provide by the linear regulator [10] and the battery [17] for at least four hours. This was tested on senior design demo day, when the guitar was played constantly by users that came to the display table from 12:30pm until 4:30pm. Thus, showing that the guitar meets the design requirement of lasting for at least 4 hours with constant usage.

[DLM]

4.6 Overall Testing

This test requires that all the above tests have been successful. To begin, power on the guitar by switching on the battery switch near the guitar jack. The battery’s LEDs should light up if the battery is charged. Next, flip the switch on front of the guitar from off to on. After powering on, look inside the guitar case and ensure that six green LEDs are on. Then, use a guitar cable and hook up the guitar to an amp and/or speaker. Prior to plugging in the cable, turn the volume dial down (counter-clock wise). Finally, pick and/or strum the guitar and check if any sound can be heard from the amp and/or speakers. If no sound is heard, double check the connection between the guitar and the amp or speakers.

Furthermore, once all the components were placed inside the guitar body, the weight of the guitar body was taken by placing the guitar on the scaled. The weight of the guitar was 9 pounds which is well under the design requirement of 14 pounds.

[DLM, KZY]

5 Operation, Maintenance, and Repair Instructions

5.1 Operation Instructions

All tests discuss in the previous section should be done prior to using the Stringless Guitar. Assuming all tests have been successful, the first thing to do is that needs done is to power the guitar. First, turn on the battery by pressing the on switch on the battery located towards the guitar jack. Afterwards, press the power switch on the front panel of the guitar to turn on the rest of the guitar. Once both of those switches have been turned on, the green LEDs will light up on the boards to indicate that the boards are powered on and running correctly. If at any time a blue LED is on, the corresponding board has an initialization issue that needs to be fixed prior to continue use of the guitar. If at any time a red LED is on, the corresponding board has a system error and will either need replacing and/or reprogram prior to using the guitar. Playing the Stringless Guitar is similar to playing that of a regular guitar.

[DLM, KZY]

5.2 Maintenance Instructions

The Stringless Guitar does not require a lot of maintenance. The main components that need to be maintain is the guitar battery and the fret sensors. The battery can be recharged by connecting a charging connector to the 12 V port on the battery. Cleaning the frets will ensure proper conductivity of the copper traces when playing.

[DLM, KZY]

5.3 Repair Instructions

All components of the Stringless Guitar can be removed from the guitar and be replaced. For example, if one of the fret board sensors malfunctions, the screws holding the fret sensor in place can be unscrewed allowing the fret sensor to come off the guitar. Similarly, the piezoelectric strips are tacked on the strumming plates which are also screwed into the guitar. The PCBs are held in the guitar via Velcro strips attached on the back of each board and can be pulled off from the guitar.

If any blue LEDs comes on after powering up the guitar, the corresponding board has failed to initialize properly. First, check that the corresponding SD card has the correct files for the specific board. Each board should have a unique set of audio files associated with them. If the SD card is not the problem, the board will require reprogramming.

If any red LEDs comes on, the corresponding board has failed to run properly. First, reprogram the board and check again if the red LED turns on. If the red LED turns on again, replace the board with a new board.

[DLM, KZY]

6 Parts List

6.1 Part List

Table 25 contains the all the parts needed to complete the project coming out of fall design semester. However, most of these components ended up not being used because the components were part of the charging power circuit that was scrapped for a simpler circuit and battery that already had the recharging feature in it.

Table 25: Part List First Round

Qty.	Labels	Part Num.	Description
1	V_Bat	SLA-12V2-3	12 Volt 2.2 Ah Sealed Lead Acid Rechargeable Battery - F1 Terminal
1	V_supply	PP3-002B	Power Barrel Connector Plug 2.50mm ID(0.098"), 5.50mm OD (0.217") Free Hnaging (In-Line)
1	V_supply	PJ-202B	Power Barrel Connector Jack 2.50mm ID(0.098"), 5.50mm OD (0.217") Through Hole, Right Angle
1	V_supply	**N/A**	RGBZONE AC To DC 3A 12V 36W Output Switching Power Supply Converter Transformers With DC Jack for LED Light Strip CCTV
2	CON1, CON2	DT06-2S	TE Connectivity, Automotive Connectors DT Plug 2 Way
1	U1	NDC7002N	Fairchild Semiconductor, MOSFET 2N-CH 50V 0.51A SSOT6
3	U2, U3, U5	FDC5614P	Fairchild Semiconductor, MOSFET P-CH 60V 3A SSOT-6
1	D1	MBR140T3G	On Semiconductor, Diode Schottky 40V 1A Surface Mount Powermite
1	LED1	WP7113GD	Kingbright, Green 568nm LED Indication - Discrete 2.2V Radial
1	LED2	WP710A10LSRD	Kingbright, Red 640nm LED Indication - Discrete 1.65V Radial
1	C1	GRM155R71H103KA88D	Ceramic Capacitor, 10000pF ±10% 50V Ceramic Capacitor X7R 0402 (1005 Metric)
1	C2	GRM155R71C224KA12D	Ceramic Capacitor, 0.22µF ±10% 16V Ceramic Capacitor X7R 0402 (1005 Metric)
3	R3, R5, R7	CRCW060310K0FKEAHP	Vishay Dale, RES SMD 10K OHM 1% 1/4W 0603
2	R1, R4	ERJ-PA3F6200V	Panasonic, RES SMD 620 OHM 1% 1/4W 0603
4	R6, R31, R22, R23	ERJ-PA3F1003V	Panasonic, RES SMD 100K OHM 1% 1/4W 0603
1	R2	CRCW060327K0FKEAHP	Vishay Dale,, RES SMD 2.7K OHM 1% 1/4W 0603
1	R8	RCS0603100RFKEA	Vishay Dale, RES SMD 100 OHM 1% 1/4W 0603
1	F1	0438003.WR	Fuse, BRD MNT 3A 12VAC 32VDC 0603

1	R9,	AS520R0FLF	TT Electronics, Wirewound Resistors - Through Hole 20 OHM 1% 7W
1	U4	LT8616EFE#PBF	Linear Tech, Buck Switching Regulator IC Positive Adjustable 0.79V 2 Output 1.5A, 2.5A 28-TSSOP (0.173", 4.40mm Width) Exposed Pad
2	C3, C8	GRM1555C1H101JA01D	Murata, 100pF ±5% 50V Ceramic Capacitor C0G, NP0 0402 (1005 Metric)
2	C4, C6	GRM1555C1H4R7CA01D	Murata, 4.7pF ±0.25pF 50V Ceramic Capacitor C0G, NP0 0402 (1005 Metric)
1	C5	04026D226MAT2A	AVX, 22µF ±20% 6.3V Ceramic Capacitor X5R 0402 (1005 Metric)
1	C7	GRM21BR60G476ME15L	Murata, 47µF ±20% 4V Ceramic Capacitor X5R 0805 (2012 Metric)
2	C9, C10	GJM1555C1H1R0BB01D	Murata, 1pF ±0.1pF 50V Ceramic Capacitor C0G, NP0 0402 (1005 Metric)
1	C11	GRM155R60J105KE19D	Murata, 1µF ±10% 6.3V Ceramic Capacitor X5R 0402 (1005 Metric)
1	L1	SLF7032T-3R3M1R9-2PF	TDK, 3.3µH Shielded Wirewound Inductor 1.9A 27.6 mOhm Max Nonstandard
1	L2	NR6020T2R2N	Taiyo Yuden, 2.2µH Shielded Wirewound Inductor 2.7A 40.8 mOhm Max Nonstandard
6	R13, R19, R15, R16, R20, R21	HMC0603JT100M	Stackpole, RES SMD 100M OHM 5% 1/10W 0603
1	R14	RC0603FR-07187KL	Yageo, RES SMD 187K OHM 1% 1/16W 00603
1	R17	RC0603FR-07316KL	Yageo RES SMD 316K OHM 1% 1/10W 0603
1	R18	RC0603FR-0714K7L	Yageo, RES SMD 14.7K OHM 1% 1/10W 0603
1	D2	NSR20F20NXT56	Diode Schottky 20V 2A (DC) Surface Mount 2-DSN (1.6x.80)
1	SW1	M2011SS1W01/UC	NKK Switches, Toggle Switch SPST Panel Mount
6	-	PIC32MX270F512L	32-bit Microcontrollers - MCU 32bit MCU, 100TQFP 512KB Flash 64KB RAM
6	-	AD5689RARUZ	Digital to Analog Converters - DAC 16B 2-CH SPI IF w/ on-chip ref
6	-	**N/A**	mirco SD Card 19MB
6	-	472192001	8 Position Card Connector Secure Digital - microSD™ Surface Mount, Right Angle Gold
6	Con3	FTS-105-01-F-D	CONN HEADER 10POS DUAL .05" SMD
6	-	SFMC-105-01-LD	CONN Plug 10POS DUAL .05" SMD
30	-	RC0402JR-0720ML	RES SMD 20M OHM 1% 1/10W 0603
1	C1	CL05A106MP5NUNC	10µF ±20% 10V Ceramic Capacitor X5R 0402 (1005 Metric)
6	R25, R26, R27, R28, R29, R30, R31	ESR03EZPJ102	RES SMD 1K OHM 5% 1/4W 0603
7	-	LME49724	Audio Amplifier 1 Circuit Differential 8-SO
1	R24	P160KNPD-2QC25B2K	2k Ohm 1 Gang Linear Panel Mount Potentiometer None 1 Turn Conductive Plastic 0.2W, 1/5W PC Pins
1	-	Ecoflex0020Kit	Smooth-On ECOFLEX 00-20 SuperSoft Silicone

1	-	6544K13	General Purpose Low-Carbon Steel, Sheet, 0.030" Thick, 12" x 24"
6	-	1002794	SENSOR PIEZO FILM VIBRA TABS
1	-	8954K195	Ultra-Machinable 360 Brass, Rectangular Bar, 3/8" x 5/8" ,x 3 FEET LONG
2	-	CHS-06TB	Dip Switch SPST 6 Position Surface Mount Slide (Standard) Actuator 100mA 6VDC

Table 26 contains the all the parts needed to complete the final project which can be seen in the Completed Prototype section. Some the parts like connectors (Mfg P/N: FTS-105-01-F-D and SFMC-105-01-LD) , PICs, and DAC were sampled from the individual company. Other components needed like fuses and fuse holder were sampled from the ECE's stock room. So between sampling from the company, the ECE's stock room, the final part list and some of the components in Table 25, all components were gather or ordered to complete the project. The labels of each part in the table corresponds to their respective labels discussed in the Schematics section of the report.

Table 26: Final Part List

Qty.	Labels	Part Num.	Description
1			Battery: TalentCell Rechargeable 72W 100WH battery
10		LDT0-028K	**Strum-Sensor:** Peizo LDT with Crimps Vibration Sensor/Switch
7			**SD:** SD Memory Card
1		14072	**Power:** USB connector to leads
2		PP3-002B	**Power:** Power Barrel Connector Plug 2.50mm ID(0.098"), 5.50mm OD (0.217") Free Hnaging (In-Line)
2		LM1117T-3.3/NOPB	**Power:** Linear Voltage Regulator IC Positive Fixed Output 3.3V 800mA TO-220-3
3			**Summing Amp:** Bourns Guitar & Amp Potentiometer, 500K , Audio, Knurled Split Shaft
2			**Summing Amp:** Switchcraft L11 Mono Female 1/4-Inch Jack Long Shaft with Nut and Washer, Nickel Finish
3		LM158	**Summing Amp:** IC OPAMP GP 1.1MHZ 8SO
5		92095A491	bolt, stainless steel, metric 6, 1mm pitch, 80mm long
12		98676A200	nut, black oxide metric 4, 0.7mmpitch,
12		91239a158	bolt, black oxide metric 4, 0.7mmpitch,
6	PIC	PIC32MX270F512L	32-bit Microcontrollers - MCU 32bit MCU, 100TQFP 512KB

			Flash 64KB RA
6	DAC	AD5689RARUZ	Digital to Analog Converters - DAC 16B 2-CH SPI IF w/ on-chip re
30	J1, J4,J3,J5, J2	FTS-105-01-F-D	CONN HEADER 10POS DUAL .05" SM
30	J1, J4,J3,J5, J2	SFMC-105-01-LD	CONN Plug 10POS DUAL .05" SM
Main Boards (6 Boards)			
10	OP	UA741CDT	Board: Strum-Sensor: General Purpose Amplifier 1 Circuit 8-SOIC
10	R1	RCS040210M0JNED	Board: RES SMD 10M OHM 5% 1/5W 0402
50	R2,R3,R4,R5, R6,R7	CRCW04021M00JNEDHP	Board: RES SMD 1M OHM 5% 1/5W 0402
25	C1	04025C101KAT2A	Board: CAP CER 100PF 50V X7R 0402
15	C2	C0402C102J4RACTU	Board: CAP CER 1000PF 16V X7R 0402
100	C7,C22,C20, C15,C17, C10, C12, C9, C3, C4, C6, C5	CL05A106MP8NUB8	Board: CAP CER 10UF 10V X5R 0402
100	C8,C21,C19, C16, C18, C11,C13, C14, C28, C26, C27, C30, C25	GRM155R71C104JA88D	Board: CAP CER 0.1UF 16V X7R 0402
35	R54, R55,R56,R57	CRCW0402100KJNEDHP	Board: RES SMD 100K OHM 5% 1/5W 0402
15	R43	RCS040210K0JNED	Board: RES SMD 10K OHM 5% 1/5W 0402
15	R44	CRCW04021K00JNEDHP	Board: RES SMD 1K OHM 5% 1/5W 0402
10	C29	TPSR106K006R1000	Board: CAP TANT 10UF 6.3V 10% 0805 ESR
50	R10, R11, R9, R8, R12	RC0402JR-0720ML	Board: RES SMD 20M OHM 5% 1/16W 0402
25	R17,R18,R19	ERJ-PA2F1500X	Board: RES SMD 150 OHM 1% 1/5W 0402

[ARB, NAH, DLM, KZY]

6.2 Budget (Estimated)

Table 27 shows the estimated material cost for the parts needed to complete the project coming out of fall design semester. However, most of these components ended up not being used because the components were part of the charging power circuit that was scrapped for a simpler circuit and battery that already had the recharging feature in it. If a cost is not listed for a part, then the part has already been bought and/or sampled.

Table 27: Material Cost First Round

Qty.	Labels	Part Num.	Description	Cost	Total Cost
1	V_Bat	SLA-12V2-3	12 Volt 2.2 Ah Sealed Lead Acid Rechargeable Battery - F1 Terminal	$12.95	$12.95
1	V_supply	PP3-002B	Power Barrel Connector Plug 2.50mm ID(0.098"), 5.50mm OD (0.217") Free Hnaging (In-Line)	$1.36	$1.36
1	V_supply	PJ-202B	Power Barrel Connector Jack 2.50mm ID(0.098"), 5.50mm OD (0.217") Through Hole, Right Angle	$0.93	$0.93
1	V_supply	**N/A**	RGBZONE AC To DC 3A 12V 36W Output Switching Power Supply Converter Transformers With DC Jack for LED Light Strip CCTV	$9.99	$9.99
2	CON1, CON2	DT06-2S	TE Connectivity, Automotive Connectors DT Plug 2 Way	$1.42	$2.84
1	U1	NDC7002N	Fairchild Semiconductor, MOSFET 2N-CH 50V 0.51A SSOT6	$0.47	$0.47
3	U2, U3, U5	FDC5614P	Fairchild Semiconductor, MOSFET P-CH 60V 3A SSOT-6	$0.62	$1.86
1	D1	MBR140T3G	On Semiconductor, Diode Schottky 40V 1A Surface Mount Powermite	$0.39	$0.39
1	LED1	WP7113GD	Kingbright, Green 568nm LED Indication - Discrete 2.2V Radial	$0.38	$0.38
1	LED2	WP710A10LSRD	Kingbright, Red 640nm LED Indication - Discrete 1.65V Radial	$0.38	$0.38
1	C1	GRM155R71H103KA88D	Ceramic Capacitor, 10000pF ±10% 50V Ceramic Capacitor X7R 0402 (1005 Metric)	$0.10	$0.10
1	C2	GRM155R71C224KA12D	Ceramic Capacitor, 0.22µF ±10% 16V Ceramic Capacitor X7R 0402 (1005 Metric)	$0.10	$0.10
3	R3, R5, R7	CRCW060310K0FKEAHP	Vishay Dale, RES SMD 10K OHM 1% 1/4W 0603	$0.17	$0.51
2	R1, R4	ERJ-PA3F6200V	Panasonic, RES SMD 620 OHM 1% 1/4W 0603	$0.16	$0.32
4	R6, R31, R22, R23	ERJ-PA3F1003V	Panasonic, RES SMD 100K OHM 1% 1/4W 0603	$0.16	$0.64
1	R2	CRCW060327K0FKEAHP	Vishay Dale,, RES SMD 2.7K OHM 1% 1/4W 0603	$0.17	$0.17

1	R8	RCS0603100RFKEA	Vishay Dale, RES SMD 100 OHM 1% 1/4W 0603	$0.14	$0.14
1	F1	0438003.WR	Fuse, BRD MNT 3A 12VAC 32VDC 0603	$1.15	$1.15
1	R9,	AS520R0FLF	TT Electronics, Wirewound Resistors - Through Hole 20 OHM 1% 7W	$1.25	$1.25
1	U4	LT8616EFE#PBF	Linear Tech, Buck Switching Regulator IC Positive Adjustable 0.79V 2 Output 1.5A, 2.5A 28-TSSOP (0.173", 4.40mm Width) Exposed Pad	$9.40	$9.40
2	C3, C8	GRM1555C1H101JA01D	Murata, 100pF ±5% 50V Ceramic Capacitor C0G, NP0 0402 (1005 Metric)	$0.10	$0.20
2	C4, C6	GRM1555C1H4R7CA01D	Murata, 4.7pF ±0.25pF 50V Ceramic Capacitor C0G, NP0 0402 (1005 Metric)	$0.10	$0.20
1	C5	04026D226MAT2A	AVX, 22µF ±20% 6.3V Ceramic Capacitor X5R 0402 (1005 Metric)	$0.69	$0.69
1	C7	GRM21BR60G476ME15L	Murata, 47µF ±20% 4V Ceramic Capacitor X5R 0805 (2012 Metric)	$0.47	$0.47
2	C9, C10	GJM1555C1H1R0BB01D	Murata, 1pF ±0.1pF 50V Ceramic Capacitor C0G, NP0 0402 (1005 Metric)	$0.15	$0.30
1	C11	GRM155R60J105KE19D	Murata, 1µF ±10% 6.3V Ceramic Capacitor X5R 0402 (1005 Metric)	$0.10	$0.10
1	L1	SLF7032T-3R3M1R9-2PF	TDK, 3.3µH Shielded Wirewound Inductor 1.9A 27.6 mOhm Max Nonstandard	$1.08	$1.08
1	L2	NR6020T2R2N	Taiyo Yuden, 2.2µH Shielded Wirewound Inductor 2.7A 40.8 mOhm Max Nonstandard	$0.47	$0.47
6	R13, R19, R15, R16, R20, R21	HMC0603JT100M	Stackpole, RES SMD 100M OHM 5% 1/10W 0603	$0.40	$2.40
1	R14	RC0603FR-07187KL	Yageo, RES SMD 187K OHM 1% 1/16W 00603	$0.10	$0.10
1	R17	RC0603FR-07316KL	Yageo RES SMD 316K OHM 1% 1/10W 0603	$0.10	$0.10
1	R18	RC0603FR-0714K7L	Yageo, RES SMD 14.7K OHM 1% 1/10W 0603	$0.10	$0.10
1	D2	NSR20F20NXT56	Diode Schottky 20V 2A (DC) Surface Mount 2-DSN (1.6x.80)	$0.51	$0.51
1	SW1	M2011SS1W01/UC	NKK Switches, Toggle Switch SPST Panel Mount	$3.31	$3.31
6	-	PIC32MX270F512L	32-bit Microcontrollers - MCU 32bit MCU, 100TQFP 512KB Flash 64KB RAM	-	-
6	-	AD5689RARUZ	Digital to Analog Converters - DAC 16B 2-CH SPI IF w/ on-chip ref	-	-
6	-	**N/A**	mirco SD Card 19MB	-	-
6	-	472192001	8 Position Card Connector Secure Digital - microSD™ Surface Mount, Right Angle Gold	$1.11	$6.66
6	Con3	FTS-105-01-F-D	CONN HEADER 10POS DUAL .05" SMD	-	-
6	-	SFMC-105-01-LD	CONN Plug 10POS DUAL .05" SMD	-	-
30	-	RC0402JR-0720ML	RES SMD 20M OHM 1% 1/10W 0603	$0.01	$0.25
1	C1	CL05A106MP5NUNC	10µF ±20% 10V Ceramic Capacitor X5R 0402 (1005 Metric)	$0.36	$0.36

6	R25, R26, R27, R28, R29, R30, R31	ESR03EZPJ102	RES SMD 1K OHM 5% 1/4W 0603	$0.10	$0.60
7	-	LME49724	Audio Amplifier 1 Circuit Differential 8-SO	$3.58	$35.84
1	R24	P160KNPD-2QC25B2K	2k Ohm 1 Gang Linear Panel Mount Potentiometer None 1 Turn Conductive Plastic 0.2W, 1/5W PC Pins	$0.76	$0.76
1	-	Ecoflex0020Kit	Smooth-On ECOFLEX 00-20 SuperSoft Silicone	$42.95	$42.95
1	-	6544K13	General Purpose Low-Carbon Steel, Sheet, 0.030" Thick, 12" x 24"	$8.27	$8.27
6	-	1002794	SENSOR PIEZO FILM VIBRA TABS	-	-
1	-	8954K195	Ultra-Machinable 360 Brass, Rectangular Bar, 3/8" x 5/8" ,x 3 FEET LONG	$35.95	$35.95
2	-	CHS-06TB	Dip Switch SPST 6 Position Surface Mount Slide (Standard) Actuator 100mA 6VDC	$2.44	$4.88
				Total:	**$191.88**

Table 28 shows the material cost for the parts needed to complete the final project which can be seen in the Completed Prototype section. If a cost is not listed for a part, then the part has already been bought and/or sampled.

Table 28: Final Material Cost

Qty.	Labels	Part Num.	Description	Cost	Total Cost
1			Battery: TalentCell Rechargable 72W 100WH battery	-	-
10		LDT0-028K	Strum-Sensor: Peizo LDT with Crimps Vibration Sensor/Switch	$2.09	$20.90
7			SD: SD Memory Card	$4.98	$34.86
1		14072	Power: USB conector to leads	$4.95	$4.95
2		PP3-002B	Power: Power Barrel Connector Plug 2.50mm ID(0.098"), 5.50mm OD (0.217") Free Hnaging (In-Line)	$1.36	$2.72
2		LM1117T-3.3/NOPB	Power: Linear Voltage Regulator IC Positive Fixed Output 3.3V 800mA TO-220-3	$1.41	$2.82
3			Summing Amp: Bourns Guitar & Amp Potentiometer, 500K , Audio, Knurled Split Shaft	$5.65	$16.95
2			Summing Amp: Switchcraft L11 Mono Female 1/4-Inch Jack Long Shaft with Nut and Washer, Nickel Finish	$3.60	$7.20
5		92095A491	bolt, stainless steel, metric 6, 1mm pitch, 80mm long		$8.14
12		98676A200	nut, black oxide metric 4, 0.7mmpitch,		$8.08
12		91239a158	bolt, black oxide metric 4, 0.7mmpitch,		$12.30
6	-	PIC32MX270F512L	32-bit Microcontrollers - MCU 32bit MCU, 100TQFP 512KB Flash 64KB RAM	-	-

6	-	AD5689RARUZ	Digital to Analog Converters - DAC 16B 2-CH SPI IF w/ on-chip ref	-	-
30	J1, J2, J3,J4,J5	FTS-105-01-F-D	CONN HEADER 10POS DUAL .05" SMD	-	-
30	-	SFMC-105-01-LD	CONN Plug 10POS DUAL .05" SMD	-	-
Main Boards (6 Boards)					
50	R2,R3,R4,R5, R6,R7	CRCW04021M00JNEDHP	Board: RES SMD 1M OHM 5% 1/5W 0402	$0.06	$3.19
25	C1	04025C101KAT2A	Board: CAP CER 100PF 50V X7R 0402	$0.13	$3.32
15	C2	C0402C102J4RACTU	Board: CAP CER 1000PF 16V X7R 0402	$0.03	$0.42
100	C7,C22,C20, C15,C17, C10, C12, C9, C3, C4, C6, C5	CL05A106MP8NUB8	Board: CAP CER 10UF 10V X5R 0402	$0.10	$10.17
100	C8,C21,C19, C16, C18, C11,C13, C14, C28, C26, C27, C30, C25	GRM155R71C104JA88D	Board: CAP CER 0.1UF 16V X7R 0402	$0.01	$1.04
35	R54, R55,R56,R57	CRCW0402100KJNEDHP	Board: RES SMD 100K OHM 5% 1/5W 0402	$0.08	$2.87
15	R43	RCS040210K0JNED	Board: RES SMD 10K OHM 5% 1/5W 0402	$0.11	$1.58
15	R44	CRCW04021K00JNEDHP	Board: RES SMD 1K OHM 5% 1/5W 0402	$0.12	$1.79
10	C29	TPSR106K006R1000	Board: CAP TANT 10UF 6.3V 10% 0805 ESR	$0.59	$5.91
50	R10, R11, R9, R8, R12	RC0402JR-0720ML	Board: RES SMD 20M OHM 5% 1/16W 0402	$0.01	$0.29
25	R17,R18,R19	ERJ-PA2F1500X	Board: RES SMD 150 OHM 1% 1/5W 0402	$0.12	$3.06
				Total:	**$159.22**

Table 29 shows the budget log for the project. As seen from the log, different components were brought to either replace components that were broken or needed changed in order to get the project working. Other parts like a break-out board for the 100 pin PIC32 was brought in order to test the PIC32 on the bread board. Finally, PCB boards were order so that the components could be solder up nice and the final project would have a nice completed board to show off. Once everything was brought, there is a total of $7.76 left in the budget.

Table 29: Budget Log

Budget	*$500.00*		**Total Left:**	**$7.76**
Date	Amount	Description		
12/6/2016	$95.73	First Round of Ordering Parts (Table 27)		

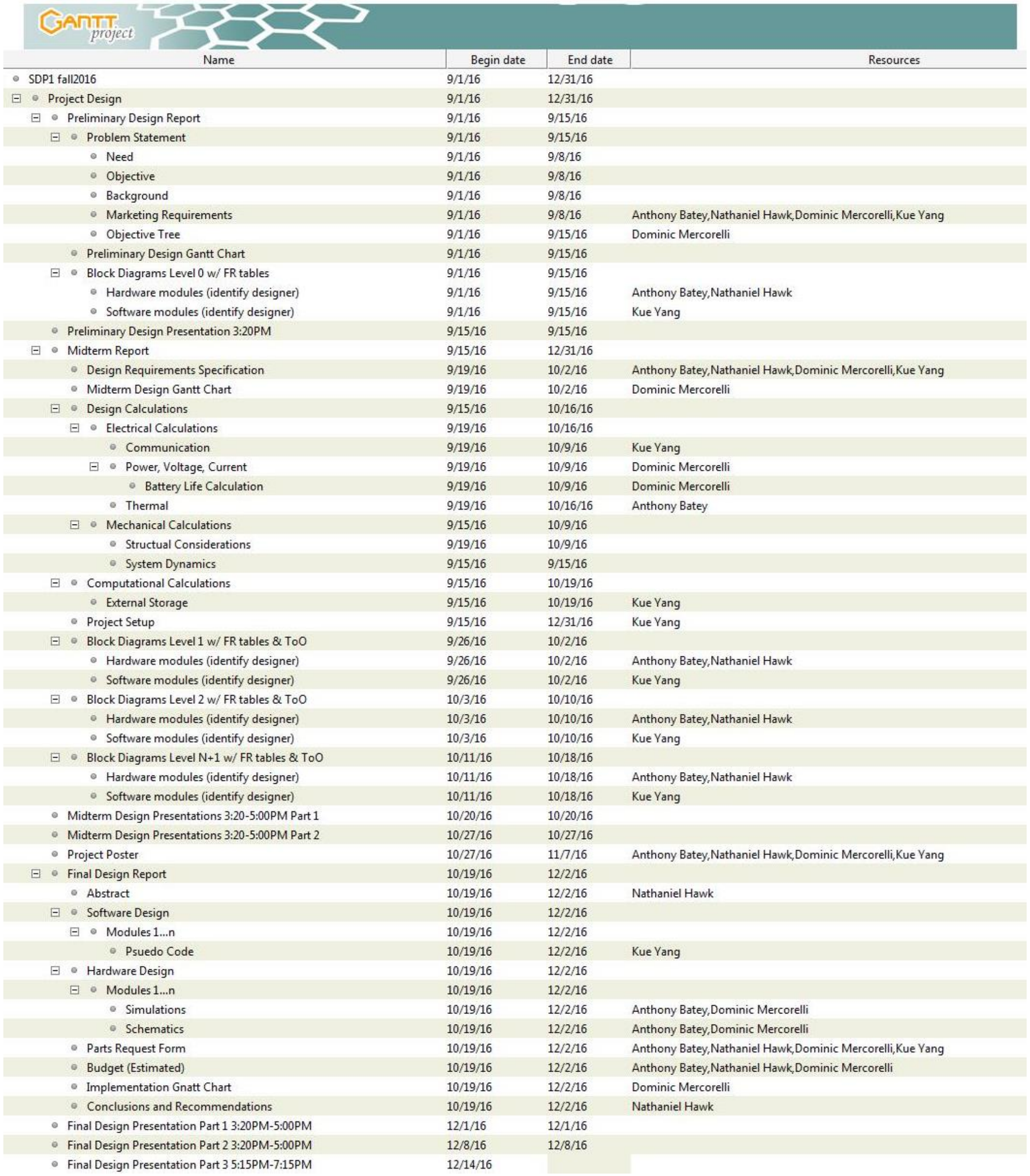
1/19/2017	$9.40	More Linear Voltage Regulator
1/24/2017	$19.49	Break-out board for the 100 pin PIC32
1/30/2017	$28.52	Screws and Bolts for Strum Sensor
2/3/2017	$29.60	Fret Board PCB Three Prototype
3/24/2017	$130.70	Final Ordering Parts (Table 28)
3/27/2017	$93.80	Final Fret Board Sensors
3/28/2017	$85.00	Final Process Boards

[ARB, NAH, DLM, KZY]

7 Project Schedules

7.1 Midterm Design Gantt

Figure 42 shows the midterm Gantt chart of the tasks that needs completed for the midterm design report. Each of the tasks has been assigned a due date and a team member(s) to complete the task.



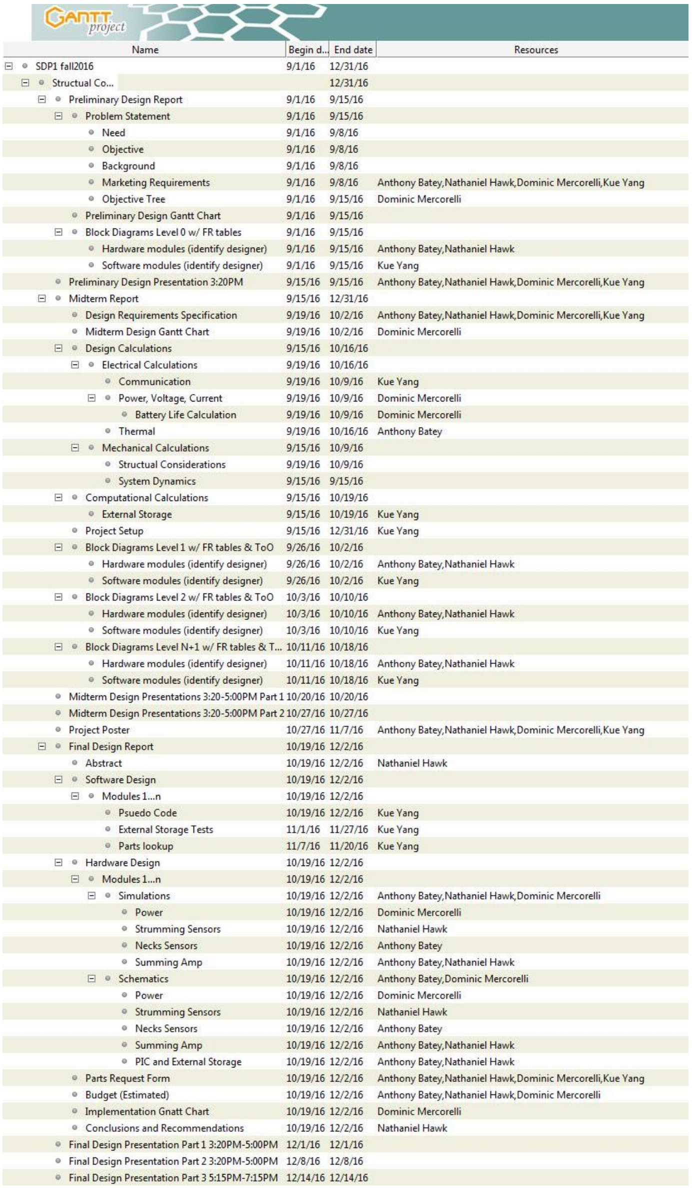
Name	Begin date	End date	Resources
SDP1 fall2016	9/1/16	12/31/16	
Project Design	9/1/16	12/31/16	
Preliminary Design Report	9/1/16	9/15/16	
Problem Statement	9/1/16	9/15/16	
Need	9/1/16	9/8/16	
Objective	9/1/16	9/8/16	
Background	9/1/16	9/8/16	
Marketing Requirements	9/1/16	9/8/16	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
Objective Tree	9/1/16	9/15/16	Dominic Mercorelli
Preliminary Design Gantt Chart	9/1/16	9/15/16	
Block Diagrams Level 0 w/ FR tables	9/1/16	9/15/16	
Hardware modules (identify designer)	9/1/16	9/15/16	Anthony Batey,Nathaniel Hawk
Software modules (identify designer)	9/1/16	9/15/16	Kue Yang
Preliminary Design Presentation 3:20PM	9/15/16	9/15/16	
Midterm Report	9/15/16	12/31/16	
Design Requirements Specification	9/19/16	10/2/16	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
Midterm Design Gantt Chart	9/19/16	10/2/16	Dominic Mercorelli
Design Calculations	9/15/16	10/16/16	
Electrical Calculations	9/19/16	10/16/16	
Communication	9/19/16	10/9/16	Kue Yang
Power, Voltage, Current	9/19/16	10/9/16	Dominic Mercorelli
Battery Life Calculation	9/19/16	10/9/16	Dominic Mercorelli
Thermal	9/19/16	10/16/16	Anthony Batey
Mechanical Calculations	9/15/16	10/9/16	
Structual Considerations	9/19/16	10/9/16	
System Dynamics	9/15/16	9/15/16	
Computational Calculations	9/15/16	10/19/16	
External Storage	9/15/16	10/19/16	Kue Yang
Project Setup	9/15/16	12/31/16	Kue Yang
Block Diagrams Level 1 w/ FR tables & ToO	9/26/16	10/2/16	
Hardware modules (identify designer)	9/26/16	10/2/16	Anthony Batey,Nathaniel Hawk
Software modules (identify designer)	9/26/16	10/2/16	Kue Yang
Block Diagrams Level 2 w/ FR tables & ToO	10/3/16	10/10/16	
Hardware modules (identify designer)	10/3/16	10/10/16	Anthony Batey,Nathaniel Hawk
Software modules (identify designer)	10/3/16	10/10/16	Kue Yang
Block Diagrams Level N+1 w/ FR tables & ToO	10/11/16	10/18/16	
Hardware modules (identify designer)	10/11/16	10/18/16	Anthony Batey,Nathaniel Hawk
Software modules (identify designer)	10/11/16	10/18/16	Kue Yang
Midterm Design Presentations 3:20-5:00PM Part 1	10/20/16	10/20/16	
Midterm Design Presentations 3:20-5:00PM Part 2	10/27/16	10/27/16	
Project Poster	10/27/16	11/7/16	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
Final Design Report	10/19/16	12/2/16	
Abstract	10/19/16	12/2/16	Nathaniel Hawk
Software Design	10/19/16	12/2/16	
Modules 1...n	10/19/16	12/2/16	
Psuedo Code	10/19/16	12/2/16	Kue Yang
Hardware Design	10/19/16	12/2/16	
Modules 1...n	10/19/16	12/2/16	
Simulations	10/19/16	12/2/16	Anthony Batey,Dominic Mercorelli
Schematics	10/19/16	12/2/16	Anthony Batey,Dominic Mercorelli
Parts Request Form	10/19/16	12/2/16	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
Budget (Estimated)	10/19/16	12/2/16	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli
Implementation Gnatt Chart	10/19/16	12/2/16	Dominic Mercorelli
Conclusions and Recommendations	10/19/16	12/2/16	Nathaniel Hawk
Final Design Presentation Part 1 3:20PM-5:00PM	12/1/16	12/1/16	
Final Design Presentation Part 2 3:20PM-5:00PM	12/8/16	12/8/16	
Final Design Presentation Part 3 5:15PM-7:15PM	12/14/16		

Figure 42: Midterm Design Gantt Chart

[DLM]

7.2 Final Design Gantt

Figure 43 shows the final Gantt chart of the tasks that needs completed for the final design report. Each of the tasks has been assigned a due date and a team member(s) to complete the task.



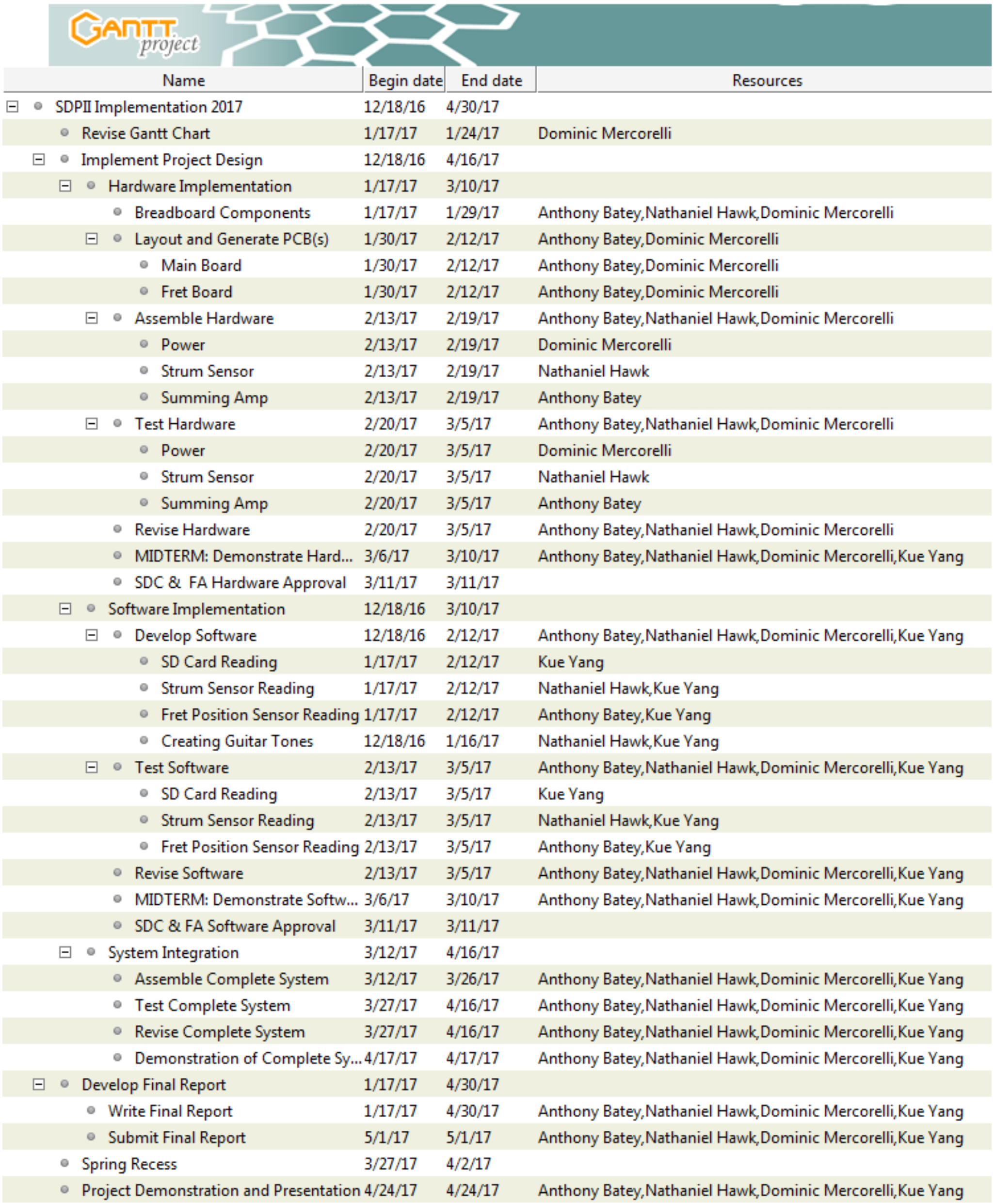
Name	Begin d...	End date	Resources
SDP1 fall2016	9/1/16	12/31/16	
Structual Co...		12/31/16	
Preliminary Design Report	9/1/16	9/15/16	
Problem Statement	9/1/16	9/15/16	
Need	9/1/16	9/8/16	
Objective	9/1/16	9/8/16	
Background	9/1/16	9/8/16	
Marketing Requirements	9/1/16	9/8/16	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
Objective Tree	9/1/16	9/15/16	Dominic Mercorelli
Preliminary Design Gantt Chart	9/1/16	9/15/16	
Block Diagrams Level 0 w/ FR tables	9/1/16	9/15/16	
Hardware modules (identify designer)	9/1/16	9/15/16	Anthony Batey,Nathaniel Hawk
Software modules (identify designer)	9/1/16	9/15/16	Kue Yang
Preliminary Design Presentation 3:20PM	9/15/16	9/15/16	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
Midterm Report	9/15/16	12/31/16	
Design Requirements Specification	9/19/16	10/2/16	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
Midterm Design Gantt Chart	9/19/16	10/2/16	Dominic Mercorelli
Design Calculations	9/15/16	10/16/16	
Electrical Calculations	9/19/16	10/16/16	
Communication	9/19/16	10/9/16	Kue Yang
Power, Voltage, Current	9/19/16	10/9/16	Dominic Mercorelli
Battery Life Calculation	9/19/16	10/9/16	Dominic Mercorelli
Thermal	9/19/16	10/16/16	Anthony Batey
Mechanical Calculations	9/15/16	10/9/16	
Structual Considerations	9/19/16	10/9/16	
System Dynamics	9/15/16	9/15/16	
Computational Calculations	9/15/16	10/19/16	
External Storage	9/15/16	10/19/16	Kue Yang
Project Setup	9/15/16	12/31/16	Kue Yang
Block Diagrams Level 1 w/ FR tables & ToO	9/26/16	10/2/16	
Hardware modules (identify designer)	9/26/16	10/2/16	Anthony Batey,Nathaniel Hawk
Software modules (identify designer)	9/26/16	10/2/16	Kue Yang
Block Diagrams Level 2 w/ FR tables & ToO	10/3/16	10/10/16	
Hardware modules (identify designer)	10/3/16	10/10/16	Anthony Batey,Nathaniel Hawk
Software modules (identify designer)	10/3/16	10/10/16	Kue Yang
Block Diagrams Level N+1 w/ FR tables & T...	10/11/16	10/18/16	
Hardware modules (identify designer)	10/11/16	10/18/16	Anthony Batey,Nathaniel Hawk
Software modules (identify designer)	10/11/16	10/18/16	Kue Yang
Midterm Design Presentations 3:20-5:00PM Part 1	10/20/16	10/20/16	
Midterm Design Presentations 3:20-5:00PM Part 2	10/27/16	10/27/16	
Project Poster	10/27/16	11/7/16	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
Final Design Report	10/19/16	12/2/16	
Abstract	10/19/16	12/2/16	Nathaniel Hawk
Software Design	10/19/16	12/2/16	
Modules 1...n	10/19/16	12/2/16	
Psuedo Code	10/19/16	12/2/16	Kue Yang
External Storage Tests	11/1/16	11/27/16	Kue Yang
Parts lookup	11/7/16	11/20/16	Kue Yang
Hardware Design	10/19/16	12/2/16	
Modules 1...n	10/19/16	12/2/16	
Simulations	10/19/16	12/2/16	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli
Power	10/19/16	12/2/16	Dominic Mercorelli
Strumming Sensors	10/19/16	12/2/16	Nathaniel Hawk
Necks Sensors	10/19/16	12/2/16	Anthony Batey
Summing Amp	10/19/16	12/2/16	Anthony Batey,Nathaniel Hawk
Schematics	10/19/16	12/2/16	Anthony Batey,Dominic Mercorelli
Power	10/19/16	12/2/16	Dominic Mercorelli
Strumming Sensors	10/19/16	12/2/16	Nathaniel Hawk
Necks Sensors	10/19/16	12/2/16	Anthony Batey
Summing Amp	10/19/16	12/2/16	Anthony Batey,Nathaniel Hawk
PIC and External Storage	10/19/16	12/2/16	Anthony Batey,Nathaniel Hawk
Parts Request Form	10/19/16	12/2/16	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
Budget (Estimated)	10/19/16	12/2/16	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli
Implementation Gnatt Chart	10/19/16	12/2/16	Dominic Mercorelli
Conclusions and Recommendations	10/19/16	12/2/16	Nathaniel Hawk
Final Design Presentation Part 1 3:20PM-5:00PM	12/1/16	12/1/16	
Final Design Presentation Part 2 3:20PM-5:00PM	12/8/16	12/8/16	
Final Design Presentation Part 3 5:15PM-7:15PM	12/14/16	12/14/16	

Figure 43: Final Design Gantt Chart

[DLM, KZY]

7.3 Proposed Implementation Gantt

Figure 44 shows the proposed implementation Gantt chart for next semester that will be completed prior to demo day. Each of the tasks has been assigned a start date, a due date and team member(s) to complete the task.



Name	Begin date	End date	Resources
SDPII Implementation 2017	12/18/16	4/30/17	
Revise Gantt Chart	1/17/17	1/24/17	Dominic Mercorelli
Implement Project Design	12/18/16	4/16/17	
Hardware Implementation	1/17/17	3/10/17	
Breadboard Components	1/17/17	1/29/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli
Layout and Generate PCB(s)	1/30/17	2/12/17	Anthony Batey,Dominic Mercorelli
Main Board	1/30/17	2/12/17	Anthony Batey,Dominic Mercorelli
Fret Board	1/30/17	2/12/17	Anthony Batey,Dominic Mercorelli
Assemble Hardware	2/13/17	2/19/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli
Power	2/13/17	2/19/17	Dominic Mercorelli
Strum Sensor	2/13/17	2/19/17	Nathaniel Hawk
Summing Amp	2/13/17	2/19/17	Anthony Batey
Test Hardware	2/20/17	3/5/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli
Power	2/20/17	3/5/17	Dominic Mercorelli
Strum Sensor	2/20/17	3/5/17	Nathaniel Hawk
Summing Amp	2/20/17	3/5/17	Anthony Batey
Revise Hardware	2/20/17	3/5/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli
MIDTERM: Demonstrate Hard...	3/6/17	3/10/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
SDC & FA Hardware Approval	3/11/17	3/11/17	
Software Implementation	12/18/16	3/10/17	
Develop Software	12/18/16	2/12/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
SD Card Reading	1/17/17	2/12/17	Kue Yang
Strum Sensor Reading	1/17/17	2/12/17	Nathaniel Hawk,Kue Yang
Fret Position Sensor Reading	1/17/17	2/12/17	Anthony Batey,Kue Yang
Creating Guitar Tones	12/18/16	1/16/17	Nathaniel Hawk,Kue Yang
Test Software	2/13/17	3/5/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
SD Card Reading	2/13/17	3/5/17	Kue Yang
Strum Sensor Reading	2/13/17	3/5/17	Nathaniel Hawk,Kue Yang
Fret Position Sensor Reading	2/13/17	3/5/17	Anthony Batey,Kue Yang
Revise Software	2/13/17	3/5/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
MIDTERM: Demonstrate Softw...	3/6/17	3/10/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
SDC & FA Software Approval	3/11/17	3/11/17	
System Integration	3/12/17	4/16/17	
Assemble Complete System	3/12/17	3/26/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
Test Complete System	3/27/17	4/16/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
Revise Complete System	3/27/17	4/16/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
Demonstration of Complete Sy...	4/17/17	4/17/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
Develop Final Report	1/17/17	4/30/17	
Write Final Report	1/17/17	4/30/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
Submit Final Report	5/1/17	5/1/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
Spring Recess	3/27/17	4/2/17	
Project Demonstration and Presentation	4/24/17	4/24/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang

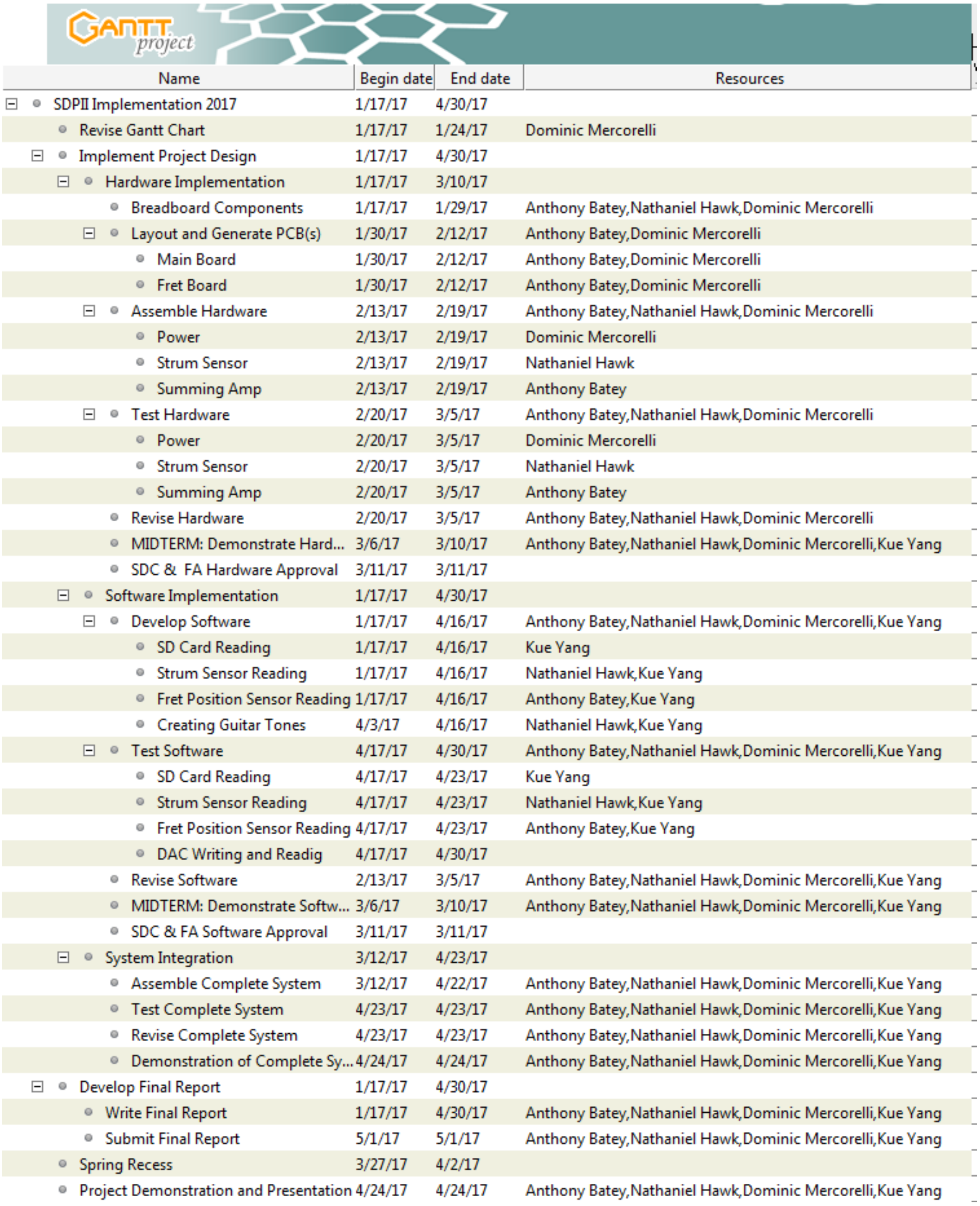
Figure 44: Proposed Implementation Gantt Chart

[DLM, KZY]

7.4 Actual Implementation Gantt

Figure 45 shows the Actual implementation Gantt chart for implementation semester. Each of the tasks has been was assigned to a team member and the start and end date corresponds to when the task actually got completed by the team member(s).

The tasks of the Actual Implementation Gantt Chart is the exactly the same as the tasks of the Proposed Implementation Gantt Chart. However, there was only one change to the dates that the task where completed. First off, everything was going as planned up until the midterm demonstration. Once the midterm demonstration was completed, the team switch to a different PIC32 in the family, for more pins. However, when this happened there was problem getting the new PIC32 to communicate with the DAC via SPI. So Kue, the software lead spent a couple of extra week trying to figure out why the DAC would not communicate with the PIC32. With this problem, everything after the midterm demonstration in the Gantt chart got pushed back a couple of weeks. However, that was the only big set back from the proposed implementation Gantt chart and the final implementation Gantt chart. All the tasks were still completed on time for Senior Design Demo Day.



Name	Begin date	End date	Resources
SDPII Implementation 2017	1/17/17	4/30/17	
Revise Gantt Chart	1/17/17	1/24/17	Dominic Mercorelli
Implement Project Design	1/17/17	4/30/17	
Hardware Implementation	1/17/17	3/10/17	
Breadboard Components	1/17/17	1/29/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli
Layout and Generate PCB(s)	1/30/17	2/12/17	Anthony Batey,Dominic Mercorelli
Main Board	1/30/17	2/12/17	Anthony Batey,Dominic Mercorelli
Fret Board	1/30/17	2/12/17	Anthony Batey,Dominic Mercorelli
Assemble Hardware	2/13/17	2/19/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli
Power	2/13/17	2/19/17	Dominic Mercorelli
Strum Sensor	2/13/17	2/19/17	Nathaniel Hawk
Summing Amp	2/13/17	2/19/17	Anthony Batey
Test Hardware	2/20/17	3/5/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli
Power	2/20/17	3/5/17	Dominic Mercorelli
Strum Sensor	2/20/17	3/5/17	Nathaniel Hawk
Summing Amp	2/20/17	3/5/17	Anthony Batey
Revise Hardware	2/20/17	3/5/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli
MIDTERM: Demonstrate Hard...	3/6/17	3/10/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
SDC & FA Hardware Approval	3/11/17	3/11/17	
Software Implementation	1/17/17	4/30/17	
Develop Software	1/17/17	4/16/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
SD Card Reading	1/17/17	4/16/17	Kue Yang
Strum Sensor Reading	1/17/17	4/16/17	Nathaniel Hawk,Kue Yang
Fret Position Sensor Reading	1/17/17	4/16/17	Anthony Batey,Kue Yang
Creating Guitar Tones	4/3/17	4/16/17	Nathaniel Hawk,Kue Yang
Test Software	4/17/17	4/30/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
SD Card Reading	4/17/17	4/23/17	Kue Yang
Strum Sensor Reading	4/17/17	4/23/17	Nathaniel Hawk,Kue Yang
Fret Position Sensor Reading	4/17/17	4/23/17	Anthony Batey,Kue Yang
DAC Writing and Readig	4/17/17	4/30/17	
Revise Software	2/13/17	3/5/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
MIDTERM: Demonstrate Softw...	3/6/17	3/10/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
SDC & FA Software Approval	3/11/17	3/11/17	
System Integration	3/12/17	4/23/17	
Assemble Complete System	3/12/17	4/22/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
Test Complete System	4/23/17	4/23/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
Revise Complete System	4/23/17	4/23/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
Demonstration of Complete Sy...	4/24/17	4/24/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
Develop Final Report	1/17/17	4/30/17	
Write Final Report	1/17/17	4/30/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
Submit Final Report	5/1/17	5/1/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang
Spring Recess	3/27/17	4/2/17	
Project Demonstration and Presentation	4/24/17	4/24/17	Anthony Batey,Nathaniel Hawk,Dominic Mercorelli,Kue Yang

Figure 45: Actual Implementation Gantt Chart

[DLM, KZY]

8 Design Team Information

Anthony R. Batey, EE, Sensor and Actuators, Embedded Systems Interfacing, Controls II, Embedded Sci. Computing, Analog IC Design, Electromagnetic Compatibility

Nathaniel A. Hawk, EE, Electrical Properties of Materials, Analog IC Design, Optical Electronic and Photonic Devices, Active Circuits, Embedded Systems Interfacing, Digital Communications

Dominic L. Mercorelli, EE, Sensor and Actuators, Embedded Systems Interfacing, Controls II, Active Circuits, Analog IC Design, Electromagnetic Compatibility

Kue Z. Yang, CpE, Data Structures, Analog IC Design, Embedded Sci. Computing, VLSI Circuits and Systems, System Simulation

9 Conclusions and Recommendations

In order to implement the stringless guitar many systems have been developed. A library of sound files was made by recording samples from a standard electric guitar. These sound files were loaded onto an SD card. Sensors were designed to detect the position of the users' fingers on the fretboard and to detect how hard the user strums the guitar. Software was developed in order to interface with these sensors, and algorithms were written in order to make logical decisions from the sensor data. Sound files were read from an SD card and were written out to a digital to analog converter as determined from the algorithms. A PCB layout of this subsystem was made. Six of these PCB boards were used to implement the six "strings" of our stringless guitar in parallel to improve response time. A summing amplifier, with subsequent volume and tone control stages, was used to sum the analog sound signals from these six boards together and the resulting signal was sent to an external guitar amplifier. A lithium ion battery pack was used with an additional linear voltage regulator to produce the necessary voltages. Finally, a semi-hollow guitar body was packed with all of these components.

Throughout this design, a constant effort was made to meet the design specifications and marketing requirements for the stringless guitar. A Gantt chart was used to keep track of the different tasks that were assigned to team members.

The overall outcome of the project was a success. The finger sensors were able to detect the position of a user's finger and the strum sensor could detect the strength of the users strum. The correct sound files were selected and played through the summing amp.

Future students should start their project early and should test their different subsystems intensely before assembling the final product. Always save revisions of code and design documentation so that when problems arise team members can revert back to a functional version.

[NAH, DLM, KZY]

10 References

[1] F. Evangelista, "Stringless Electronic Musical Instrument". U.S. Patent RE31,019, 31 August 1982.

[2] I. Mladek, "Stringless Twitch Fret Instrument". U.S. Patent 6,018,119, 25 January 2000.

[3] E. H. Chapman, "Stringless Fingerboard Synthesizer Controller". U.S. Patent 5,140,887, 25 August 1992.

[4] X. Fiss and A. Kwasinski, "Automatic Real-Time Electric Guitar Audio Transcription," *Acoustics Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on,* pp. 373-376, 2011.

[5] R. Setiyono, A. S. Prihatmanto and P. H. Rusmin, "Design and Implementation Infrared Guitar Based on Playing Chords," *System Engineering and Technology (ICSET), 2012 International Conference on,* pp. 1-5, 2012.

[6] K. Bradley, M.-H. Cheng and V. L. Stonick, ""Automated Analysis and Computationally Efficient Synthesis of Acoustic Guitar Strings and Body," *Applications of Signal Processing to Audio and Acoustics, 1995., IEEE ASSP Workshop on,* pp. 238-241, 1995.

[7] K. Myeongsu, S. Cho, K. Jongmyon and C. Uipil, ""Implementation of Non-Stringed Guiatr using ATMegal 128," *Strategic Technology, 2007. IFOST 2007. International Forum on,* pp. 585-588, 2007.

[8] Microchip Technology, *"32-bit Microcontrollers (up to 128 KB Flash and 32 KB SRAM) with Audio ad Graphics Interfaces, USB, and Advanced Analog",* DS61168E Datasheet, 2012.

[9] Measurement Specialties, *LDT with Crimps Vibration Sensor/Switch,* Hampton: LDT0-028K Datasheet, 2008.

[10] T. Instruments, *LM1117 800-mA Low-Dropout Linear Regulator,* LM1117 datasheet, 2016.

[11] Microchip, *"32-bit Microcontrollers (up to 512 KB Flash and 64 KB SRAM) with Audio/Graphics/Touch (HMI), CAN, USB, and Advanced Analog",* Datasheet., 2016.

[12] Analog Devices, *Dual 16-/12-Bit nanoDAC+ with 2 ppm/C On-Chip Reference,* Norwood: AD5689R Datasheet., 2014.

[13] "FatFs - Generic FAT File System Module," 2017. [Online]. Available: http://elm-

chan.org/fsw/ff/00index_e.html. [Accessed 8 Feburary 2017].

[14] "TechShop," 2013. [Online]. Available: https://www.techshopbd.com/product-categories/breakout-boards/1240/micro-sd-card-breakout-board-techshop-bangladesh. [Accessed 2 December 2016].

[15] S. T. Mahbub, "PIC32 Based Audio Player With MicroSD Storage," January 2015. [Online]. Available: http://tahmidmc.blogspot.com/2015/01/audio-player-using-pic32-microsd-card.html. [Accessed 2 November 2016].

[16] Microchip, "PIC32 Peripheral Library," Microchip, 2016. [Online]. Available: http://www.microchip.com/SWLibraryWeb/product.aspx?product=PIC32%20Peripheral%20Library. [Accessed 2 November 2016].

[17] Amazon, "TalentCell Rechargeable 72W 132WH 12V/11000mAh 9V/14500mAh 5V/26400mAh DC Output Lithium Ion Battery Pack for LED Strip and CCTV Camera, Portable Li-ion Power Bank, Black," TalentCell, 2017. [Online]. Available: https://www.amazon.com/TalentCell-Rechargeable-11000mAh-14500mAh-26400mAh/dp/B016BJCRUO.

[18] On Semiconductor, *"60V P-channel Logic Level PowerTrench MOSFET"*, FDC5614 datasheet, 2015.

[19] Linear Technology, *"Dual 42V Synchronous Monolithic Step-Down regulator with 6.5 uA Quiescent Current"*, LT8616 datasheet, 2015.

[20] On Semiconductor, *"Dual N-Channel Enchancement Mode Field Effect Transistor"*, NDC7002N datasheet, 1996.

[21] On Semiconductor, *"Small signal MOSFET, Single N-Channel with ESD Proctect"*, NTA4153N datasheet, 2014.

[22] Texas Instruments, *LinCMOS Precision Dual Operational Amplifiers,* Dallas: Datasheet., 2002.

[23] Littelfuse Inc., *438 Series-0603 Fast-Acting Fuse*, 0438003.WR Datasheet, 2015.

[DLM, KYZ]

11 Appendices

11.1 UART Commands

Table 30 lists commands that are supported by the firmware. Assuming a rs232 to USB connector is used, the microprocessor can be sent commands via a command terminal. The command terminal that was used for this project was Putty.

Table 30: UART Command List

Command	Function	Command Format
HELP	Display the list of commands available.	
LIST	Lists all WAV files from external memory.	
SET	Sets the file to read.	SET “filename”
RESET	Resets the file pointer to beginning of file.	
READ	Reads numOfBytes from current file. Read from beginning of the file if reset is set to 1.	READ “reset” “numOfBytes”
DAC	Sets an output value on the DAC.	DAC “value”
ZERO	Sets all DAC outputs to zero.	
SIN	Tests the DAC using a sin wave.	
TONE	Toggles on/off the Audio Timer (Timer 3).	
PDG	Get the current period set on Timer 3.	
PDS	Configures the Timer 3 period.	PDS “period”

11.2 Firmware Documentation

The firmware for this project was hosted on Github. The firmware can be found via the following link: https://github.com/KueYang/SeniorDesignProject.

11.3 Module Documentation

11.3.1 USB configurations

All USB related tasks are disabled and not used in the application.

11.3.2 System and Peripheral Clock Configurations

The configurations used to configure the system and peripheral clocks. The system clock is configured for 40 Mhz using the internal 8 MHz oscillator. The system clock configuration can be configured as follow: SYSCLK = 40 MHz (8 MHz Crystal / FPLLIDIV * FPLLMUL / FPLLODIV) where FPLL related configurations are used to adjust the clock. The peripheral clock is configured to the same clock speed as the system clock. The peripheral clock configuration can be configured as follow: PBCLK = 40 MHz (SYSCLK / FPBDIV) where the FPB configuration is used to adjust the clock.

11.3.3 Programming Configurations

The PIC is configured to be programmed on pins ICE3 and ICD3. JTAG and the internal switch over configurations are disabled. Code protection and program flash protection are disabled.

11.3.4 Watch Dog Timer Configurations

The watch dog is enabled in software for this application.

11.4 Data Structure Documentation

11.4.1 AUDIOINFO Struct Reference

AUDIOINFO data structure.

```
#include <FILES.h>
```

Data Fields

- UINT16 bitsPerSample
- UINT16 numOfChannels
- UINT16 sampleRate
- UINT16 blockAlign

- UINT32 dataSize
- char **fileName** [16]

Detailed Description

AUDIOINFO data structure.

The **AUDIOINFO** data structure is used to store the audio header data and file name.

Field Documentation

UINT16 bitsPerSample

Variable used to store the bits per sample.

UINT16 blockAlign

Variable used to store the block align.

UINT32 dataSize

Variable used to store the size of the file data.

char fileName[16]

Variable used to store the file name.

UINT16 numOfChannels

Variable used to store the number of channels.

UINT16 sampleRate

Variable used to store the sample rate.

The documentation for this struct was generated from the following file:

- FILES.h

11.4.2 COMMANDS Struct Reference

COMMANDS data structure.

```
#include <UART.h>
```

Data Fields

- const char * **name**
- const char * **description**
- void(* **handler**)(void)

Detailed Description

COMMANDS data structure.

The **COMMANDS** data structure is used to store a command with its corresponding command handler. The command handler is used in conjunction with the serial communication for processing a command that is received from the user.

Field Documentation

const char description*

Variable used to store a description of the command.

void(handler) (void)*

Variable used to point to the command handler.

const char name*

Variable used to store the command name

The documentation for this struct was generated from the following file:

- UART.h

11.4.3 COMMANDSTR Struct Reference

COMMANDSTR data structure.

```
#include <UART.h>
```

Data Fields

- char **name** [32]
- char **arg1** [32]
- char **arg2** [32]

Detailed Description

COMMANDSTR data structure.

The command string data structure is used to store a command with its corresponding arguments.

Field Documentation

char arg1[32]

Variable used to store the first command argument.

char arg2[32]

Variable used to store the second command argument.

char name[32]

Variable used to store the name of the command.

The documentation for this struct was generated from the following file:

- UART.h

11.4.4 FILES Struct Reference

FILES data structure.

```
#include <FILES.h>
```

Data Fields

- FIL **File**
- FSIZE_t **startPtr**
- DWORD cluster
- DWORD sector
- AUDIOINFO audioInfo

Detailed Description

FILES data structure.

The **FILES** data structure is used to store the file data used in conjunction with the Fatfs File System Library. The structure also stores the audio header data corresponding to the specified file.

The documentation for this struct was generated from the following file:

- FILES.h

11.4.5 MON_FIFO Struct Reference

MON_FIFO data structure.

```
#include <FIFO.h>
```

Data Fields

- char buffer [MON_BUFFERSIZE]
- UINT16 headPtr
- UINT16 tailPtr
- UINT16 bufferSize

Detailed Description

MON_FIFO data structure.

The FIFO data structure is used to create and store a FIFO queue.

Field Documentation

char buffer[MON_BUFFERSIZE]

Variable used to store the FIFO data.

UINT16 bufferSize

Variable used to stores the queue size.

UINT16 headPtr

Variable used to point to the front of the queue.

UINT16 tailPtr

Variable used to point to the back of the queue.

The documentation for this struct was generated from the following file:

- FIFO.h

11.5 File Documentation

11.5.1 ADC.c File Reference

```
#include <p32xxxx.h>
#include "plib/plib.h"
#include "HardwareProfile.h"
#include "STDDEF.h"
#include "IO.h"
#include "TIMER.h"
#include "ADC.h"
```

Macros

- #define NUM_OF_ADCCHANNELS 1
- #define ADC_ARRAY_SIZE 5
- #define **ADC_MIDRAIL** 480
- #define **ADC_NOISEMAG** 150
- #define **ADC_MINDELTA** 10
- #define **ADC_MINSAMPLE** 256

Functions

- void ADC_ZeroBuffer (void)
  Reinitializes ADC buffers to the midrail.
- void **ADC_Init** (void)
  Initializes the ADC module.
- void **__ISR** (_ADC_VECTOR, IPL2AUTO)
  ADC Interrupt Service Routine.

Variables

- BOOL isPositive
- UINT16 peakMax [ADC_ARRAY_SIZE]
- UINT16 localMax [ADC_ARRAY_SIZE]
- UINT32 sampleCount
- BOOL startStrumDetection

Detailed Description

Author:
Kue Yang
Date:
2/27/2017
The ADC module will handle reading strumming sensor data and kick start the audio playback process.

Macro Definition Documentation

#define ADC_ARRAY_SIZE 5

Defines the ADC local array size.

#define ADC_MIDRAIL 480

Defines the ADC mid-rail.

#define ADC_MINDELTA 10

Defines the minimum change in ADC sample for indicating strum.

#define ADC_MINSAMPLE 256

Defines the minimum sample count for strum detection.

#define ADC_NOISEMAG 150

Defines the minimum noise magnitude ADC thread hold.

#define NUM_OF_ADCCHANNELS 1

Defines the number of ADC channels used.

Function Documentation

void __ISR (_ADC_VECTOR , IPL2AUTO)

ADC Interrupt Service Routine.

The interrupt service routine is used read the strummer sensor.

Returns:

Void.

void ADC_Init (void)

Initializes the ADC module.

Returns:

Void

void ADC_ZeroBuffer (void)

Reinitializes ADC buffers to the midrail.

Returns:

Void

Variable Documentation

isPositive

Indicates if the sample is the positive or negative part of signal.

localMax

Stores max values for different peaks.

peakMax

Stores max values for a single peak.

sampleCount

Counts the number of adc samples since playing a tone.

startStrumDetection

Boolean used to enable the strum detection.

11.5.2 ADC.h File Reference

Functions

- void **ADC_Init** (void)
  Initializes the ADC module.

Detailed Description

Author:
Kue Yang
Date:
11/22/2016

Function Documentation

void ADC_Init (void)

Initializes the ADC module.

Returns:
Void

11.5.3 AUDIO.c File Reference

```
#include <p32xxxx.h>
#include <stdio.h>
#include <string.h>
#include "STDDEF.h"
#include "IO.h"
#include "DAC.h"
#include "FILES.h"
#include "FILEDEF.h"
#include "AUDIO.h"
```

Functions

- UINT8 AUDIO_GetHeader (int index)
  Reads the header of a WAV file.

- BOOL AUDIO_GetAudioData (FILES *file, UINT16 bytes)
  Reads a number of bytes from the audio file.
- void **AUDIO_Init** (void)
  Initializes the Audio module.
- void AUDIO_Process (void)
  Process audio data.
- void AUDIO_ListFiles (void)
  Displays the list of audio files.
- BOOL AUDIO_setNewFile (UINT16 selectedFile)
  Sets a new file to be read.
- void **AUDIO_setNewTone** (int fret, **UINT16** factor)
  Sets a new tone.
- void AUDIO_resetFilePtr (void)
  Resets the file pointer for selected file.
- UINT32 AUDIO_getBytesRead (void)
  Returns the number of bytes read.
- BOOL AUDIO_isDoneReading (void)
  Checks if reading the selected file is done.
- UINT32 AUDIO_getBytesWritten (void)
  Returns the number of bytes written.
- BOOL AUDIO_isDoneWriting (void)
  Checks if writing to the DAC is done.
- BOOL AUDIO_ReadFile (UINT16 bytesToRead)
  Reads a number of bytes from the audio file.
- BYTE * AUDIO_GetRecieveBuffer (void)
  Gets buffer pointer.
- void AUDIO_WriteDataToDAC (void)
  Writes audio data out to the DAC.

Variables

- FILES files [MAX_NUM_OF_FILES]
- BYTE receiveBuffer [REC_BUF_SIZE]
- UINT16 LAUDIOSTACK [AUDIO_BUF_SIZE]
- UINT16 RAUDIOSTACK [AUDIO_BUF_SIZE]
- UINT16 fileIndex
- UINT16 audioInPtr
- UINT16 audioOutPtr
- UINT32 bytesRead
- UINT32 bytesWritten
- BOOL hasReadFile
- UINT16 scaleFactor
- char **buf** [64]

Detailed Description

Author:

Kue Yang

Date:

11/22/2016

The Audio module will handle all audio processing related tasks. Tasks includes: initializing Fatfs File System library, reading data from external memory and writing audio data to the DACs.

Remarks:

The Audio module requires Fatfs File System library. The library uses the library from the pic24 example project.

Function Documentation

*BOOL AUDIO_GetAudioData (FILES * file, UINT16 bytes)*

Reads a number of bytes from the audio file.

- file The files to read from.
- bytes The number of bytes to read.
  - Returns:
    Returns a boolean indicating if the file was read successfully.
  - Return values:

TRUE	if the file was read successfully.
FALSE	if the file was read unsuccessfully.

UINT32 AUDIO_getBytesRead (void)

Returns the number of bytes read.

Returns:

Returns the number of bytes read

UINT32 AUDIO_getBytesWritten (void)

Returns the number of bytes written.

Returns:

Returns the number of bytes written.

UINT8 AUDIO_GetHeader (int index)

Reads the header of a WAV file.

- index The file that is being read.

- Returns:

  A code indicating if reading the file is successful or not.

- Return values:

1,Read	the header successfully
2,Header	size is invalid
3,Chunk	ID is invalid
4,Header	format is invalid
5,Chunk	ID 1 is invalid
6,Chunk	ID 1 Data is invalid
7,Chunk	ID 2 is invalid

BYTE AUDIO_GetRecieveBuffer (void)*

Gets buffer pointer.

Gets a pointer to the buffer that stores the bytes read from the SD Card.

Returns:

Returns a pointer to the read buffer.

void AUDIO_Init (void)

Initializes the Audio module.

Initializes the SD card and Microchip MDD File library. After initialization, all the audio files that are specific to the PIC are opened.

Returns:

Void

BOOL AUDIO_isDoneReading (void)

Checks if reading the selected file is done.

Returns:

Returns a boolean indicating if done reading the selected audio file.

Return values:

TRUE,if	done reading the file.
FALSE,if	not done reading the file.

BOOL AUDIO_isDoneWriting (void)

Checks if writing to the DAC is done.

Returns:

Returns a boolean indicating if writing to the DAC is done.

Return values:

TRUE,if	done writing to the DAC.
FALSE,if	not done writing to the DAC.

void AUDIO_ListFiles (void)

Displays the list of audio files.

Finds all WAV files in the root directory of the SD card and displays them via UART to serial.

Remarks:

Requires the UART module and SD card to be initialized.

Returns:

Void

void AUDIO_Process (void)

Process audio data.

The audio data will process audio data based on data received from the IO and ADC modules. The data received by the IO module will correspond to the fret location. The data received by the ADC module will correspond to the strumming data.

Remarks:

Requires the IO and ADC modules to be initialized.

Returns:

Void

BOOL AUDIO_ReadFile (UINT16 bytesToRead)

Reads a number of bytes from the audio file.

- bytesToRead The number of bytes to read.

- Returns:

Returns a boolean indicating if the file was read successfully.

- Return values:

TRUE	if the file was read successfully.
FALSE	if the file was read unsuccessfully or is already done reading.

void AUDIO_resetFilePtr (void)

Resets the file pointer for selected file.

Returns:

Void

BOOL AUDIO_setNewFile (UINT16 selectedFile)

Sets a new file to be read.

- selectedFile The selected file to be set.

- Returns:

  A boolean indicating if the file has been set successfully.

  - Return values:

TRUE,file	has been set successfully.
FALSE,file	has not been set successfully.

void AUDIO_setNewTone (int fret, UINT16 factor)

Sets a new tone.

Sets a new tone based on the fret that is passed into the function. Resets all related variables prior to reading the new tone.

- fret The fret that is being played.
- factor The strum strength scale factor.

- Returns:

  Void

void AUDIO_WriteDataToDAC (void)

Writes audio data out to the DAC.

Writes audio data out to the DAC. Handles stopping and resetting all the selected after writing all audio data out to the DAC.

Returns:

Void

Variable Documentation

audioInPtr

The index used to store audio data into to audio buffer.

audioOutPtr

The index used to write audio data out of the audio buffer.

buf

Buffer used to store a string.

bytesRead

Stores the number of bytes that have been read.

bytesWritten

Stores the number of bytes that have been written.

fileIndex

The index used to specify the audio file that is being read.

files

The list of audio files that are to be used.

hasReadFile

Stores boolean indicating the specified audio file has been read.

LAUDIOSTACK

A buffer used to store left channel audio data.

RAUDIOSTACK

A buffer used to store right channel audio data.

receiveBuffer

A buffer used to store data read from the audio file.

scaleFactor

Stores the scaling factor.

11.5.4 AUDIO.h File Reference

```
#include "WAVDEF.h"
#include "FILES.h"
```

Macros

- #define **REC_BUF_SIZE** 512
- #define AUDIO_BUF_SIZE REC_BUF_SIZE/4

Functions

- void **AUDIO_Init** (void)
  Initializes the Audio module.
- void AUDIO_Process (void)
  Process audio data.
- BYTE * AUDIO_GetRecieveBuffer (void)
  Gets buffer pointer.
- BOOL AUDIO_ReadFile (UINT16 bytesToRead)
  Reads a number of bytes from the audio file.
- void AUDIO_WriteDataToDAC (void)
  Writes audio data out to the DAC.
- void AUDIO_ListFiles (void)
  Displays the list of audio files.
- void **AUDIO_setNewTone** (int fret, **UINT16** factor)
  Sets a new tone.
- BOOL AUDIO_setNewFile (UINT16 selectedFile)
  Sets a new file to be read.
- void AUDIO_resetFilePtr (void)
  Resets the file pointer for selected file.
- UINT32 AUDIO_getBytesRead (void)
  Returns the number of bytes read.
- UINT32 AUDIO_getBytesWritten (void)
  Returns the number of bytes written.

Detailed Description

Author:
Kue Yang
Date:
11/22/2016

Macro Definition Documentation

#define AUDIO_BUF_SIZE REC_BUF_SIZE/4

Defines the audio buffer size.

#define REC_BUF_SIZE 512

Defines the receive buffer size.

Function Documentation

UINT32 AUDIO_getBytesRead (void)

Returns the number of bytes read.

Returns:

Returns the number of bytes read

UINT32 AUDIO_getBytesWritten (void)

Returns the number of bytes written.

Returns:

Returns the number of bytes written.

BYTE AUDIO_GetRecieveBuffer (void)*

Gets buffer pointer.

Gets a pointer to the buffer that stores the bytes read from the SD Card.

Returns:

Returns a pointer to the read buffer.

void AUDIO_Init (void)

Initializes the Audio module.

Initializes the SD card and Microchip MDD File library. After initialization, all the audio files that are specific to the PIC are opened.

Returns:

Void

void AUDIO_ListFiles (void)

Displays the list of audio files.

Finds all WAV files in the root directory of the SD card and displays them via UART to serial.

Remarks:

Requires the UART module and SD card to be initialized.

Returns:

Void

void AUDIO_Process (void)

Process audio data.

The audio data will process audio data based on data received from the IO and ADC modules. The data received by the IO module will correspond to the fret location. The data received by the ADC module will correspond to the strumming data.

Remarks:

Requires the IO and ADC modules to be initialized.

Returns:

Void

BOOL AUDIO_ReadFile (UINT16 bytesToRead)

Reads a number of bytes from the audio file.

- bytesToRead The number of bytes to read.

- Returns:

Returns a boolean indicating if the file was read successfully.

- Return values:

TRUE	if the file was read successfully.
FALSE	if the file was read unsuccessfully or is already done reading.

void AUDIO_resetFilePtr (void)

Resets the file pointer for selected file.

Returns:

Void

BOOL AUDIO_setNewFile (UINT16 selectedFile)

Sets a new file to be read.

- selectedFile The selected file to be set.

- Returns:

A boolean indicating if the file has been set successfully.

- Return values:

TRUE,file	has been set successfully.
FALSE,file	has not been set successfully.

void AUDIO_setNewTone (int fret, UINT16 factor)

Sets a new tone.

Sets a new tone based on the fret that is passed into the function. Resets all related variables prior to reading the new tone.

- fret The fret that is being played.
- factor The strum strength scale factor.

- Returns:

  Void

void AUDIO_WriteDataToDAC (void)

Writes audio data out to the DAC.

Writes audio data out to the DAC. Handles stopping and resetting all the selected after writing all audio data out to the DAC.

Returns:

Void

11.5.5 DAC.c File Reference

```
#include <p32xxxx.h>
#include "STDDEF.h"
#include "SPI.h"
#include "DAC.h"
```

Functions

- void **DAC_Init** (void)
  Initializes the DAC.
- DWORD DAC_WriteToDAC (BYTE cmd_addr, WORD data)
  Writes data to the DAC.
- void **DAC_Zero** (void)
  Sets the DAC output to mid-scale.
- void DAC_ZeroOutput (void)
  Sets the DAC output to zero.
- DWORD DAC_ReadBack (BOOL channelA)
  Reads the DAC registers.

Detailed Description

Author:

Kue Yang

Date:

11/22/2016

The DAC module handles all DAC related operations such as configuring and writing to the DAC. DAC configurations are defined in the header file.

Function Documentation

void DAC_Init (void)

Initializes the DAC.

Powers on the DAC with only one channel open.

Remarks:

Requires the SPI module to be initialized.

Returns:

Void.

DWORD DAC_ReadBack (BOOL channelA)

Reads the DAC registers.

- channelA The channel to read back data.

- Returns:

The DAC register value.

DWORD DAC_WriteToDAC (BYTE cmd_addr, WORD data)

Writes data to the DAC.

The current DAC can only handle 12-bit data. Therefore, the data that is transmitted are the first 12 MSB.

- cmd_addr The command and channel of the DAC.
- data The data that will be written to the DAC.

- Remarks:

Requires SPI and the DAC to be initialized.

- Returns:

Returns a boolean indicating if writing to the DAC is successful.

- Return values:

TRUE	If the file was read successfully
FALSE	If the file was read unsuccessfully

void DAC_Zero (void)

Sets the DAC output to mid-scale.

Remarks:

Requires SPI and the DAC to be initialized.

Returns:

Void

void DAC_ZeroOutput (void)

Sets the DAC output to zero.

Remarks:

Requires SPI and the DAC to be initialized.

Returns:

Void

11.5.6 *DAC.h File Reference*

Macros

- #define **AC_ZERO** 0x7777
- #define **SYNC** PORTCbits.RC4
- #define **DAC_A** 0x1
- #define **DAC_B** 0x8
- #define DAC_B_A DAC_B | DAC_A
- #define POWER_ON_DAC_B_A 0x3C
- #define CMD_WRITE_TO_DAC 0x1
- #define CMD_UPDATE_DAC 0x2
- #define CMD_WRITE_UPDATE_DAC 0x3
- #define CMD_POWER_ON_OFF 0x4
- #define CMD_LDAC_MASK_REG 0x5
- #define CMD_SOFT_RESET 0x6
- #define CMD_INTERN_REF 0x7
- #define CMD_SET_DCEN_REG 0x8
- #define CMD_READ_CHN_REG 0x9
- #define WRITE_INPUT_CHN_A (CMD_WRITE_TO_DAC << 4) | DAC_A
- #define UPDATE_CHN_A (CMD_UPDATE_DAC << 4) | DAC_A
- #define WRITE_UPDATE_CHN_A (CMD_WRITE_UPDATE_DAC << 4) | DAC_A
- #define READ_CHN_A (CMD_READ_CHN_REG << 4) | DAC_A
- #define WRITE_INPUT_CHN_B (CMD_WRITE_TO_DAC << 4) | DAC_B
- #define UPDATE_CHN_B (CMD_UPDATE_DAC << 4) | DAC_B
- #define WRITE_UPDATE_CHN_B (CMD_WRITE_UPDATE_DAC << 4) | DAC_B
- #define READ_CHN_B (CMD_READ_CHN_REG << 4) | DAC_B
- #define POWER_ON_OFF_CHN_A_B (CMD_POWER_ON_OFF << 4) | DAC_B_A
- #define WRITE_INPUT_CHN_A_B (CMD_WRITE_TO_DAC << 4) | DAC_B_A
- #define UPDATE_CHN_A_B (CMD_UPDATE_DAC << 4) | DAC_B_A
- #define WRITE_UPDATE_CHN_A_B (CMD_WRITE_UPDATE_DAC << 4) | DAC_B_A

Functions

- void **DAC_Init** (void)
  Initializes the DAC.
- void **DAC_Zero** (void)
  Sets the DAC output to mid-scale.
- void DAC_ZeroOutput (void)

Sets the DAC output to zero.

- DWORD DAC_WriteToDAC (BYTE cmd_addr, WORD data)
  Writes data to the DAC.

Detailed Description

Author:
Kue Yang
Date:
11/22/2016

Macro Definition Documentation

#define AC_ZERO 0x7777

Defines the mid-scale value for the 16-bit DAC.

#define CMD_INTERN_REF 0x7

Defines the configuration bit to enable internal references.

#define CMD_LDAC_MASK_REG 0x5

Defines the configuration bit to mask the LDAC pin.

#define CMD_POWER_ON_OFF 0x4

Defines the configuration bit to power on the DAC.

#define CMD_READ_CHN_REG 0x9

Defines the configuration bit to read the DAC SDO pin.

#define CMD_SET_DCEN_REG 0x8

Defines the configuration bit to enable daisy chain.

#define CMD_SOFT_RESET 0x6

Defines the configuration bit to soft reset the DAC.

#define CMD_UPDATE_DAC 0x2

Defines the configuration bit to update the DAC, single channel.

#define CMD_WRITE_TO_DAC 0x1

Defines the configuration bit to write to DAC, single channel.

#define CMD_WRITE_UPDATE_DAC 0x3

Defines the configuration bit to write and update the DAC, both channels.

#define DAC_A 0x1

Defines the selection bit for channel A on the DAC.

#define DAC_B 0x8

Defines the selection bit for channel B on the DAC.

#define DAC_B_A DAC_B | DAC_A

Defines the selection bit for both channel A and channel B.

#define POWER_ON_DAC_B_A 0x3C

Defines the power on selection for channel A and channel B.

#define POWER_ON_OFF_CHN_A_B (CMD_POWER_ON_OFF << 4) | DAC_B_A

Defines the command to turn on DAC and both channel A and channel B.

#define READ_CHN_A (CMD_READ_CHN_REG << 4) | DAC_A

Defines the command to read channel A.

#define READ_CHN_B (CMD_READ_CHN_REG << 4) | DAC_B

Defines the command to read channel B.

#define SYNC PORTCbits.RC4

Defines the Enable pin used to write data to DAC.

#define UPDATE_CHN_A (CMD_UPDATE_DAC << 4) | DAC_A

Defines the command to update the registers for channel A.

#define UPDATE_CHN_A_B (CMD_UPDATE_DAC << 4) | DAC_B_A

Defines the command to update the registers for both channel A and channel B.

#define UPDATE_CHN_B (CMD_UPDATE_DAC << 4) | DAC_B

Defines the command to update the registers for channel B.

#define WRITE_INPUT_CHN_A (CMD_WRITE_TO_DAC << 4) | DAC_A

Defines the command to write to channel A.

#define WRITE_INPUT_CHN_A_B (CMD_WRITE_TO_DAC << 4) | DAC_B_A

Defines the command to write to both channel A and channel B.

#define WRITE_INPUT_CHN_B (CMD_WRITE_TO_DAC << 4) | DAC_B

Defines the command to write to channel B.

#define WRITE_UPDATE_CHN_A (CMD_WRITE_UPDATE_DAC << 4) | DAC_A

Defines the command to write and update channel A.

#define WRITE_UPDATE_CHN_A_B (CMD_WRITE_UPDATE_DAC << 4) | DAC_B_A

Defines the command to write and update both channel A and channel B.

#define WRITE_UPDATE_CHN_B (CMD_WRITE_UPDATE_DAC << 4) | DAC_B

Defines the command to write and update channel B.

Function Documentation

void DAC_Init (void)

Initializes the DAC.

Powers on the DAC with only one channel open.

Remarks:

Requires the SPI module to be initialized.

Returns:

Void.

DWORD DAC_WriteToDAC (BYTE cmd_addr, WORD data)

Writes data to the DAC.

The current DAC can only handle 12-bit data. Therefore, the data that is transmitted are the first 12 MSB.

- cmd_addr The command and channel of the DAC.
- data The data that will be written to the DAC.

- Remarks:
  Requires SPI and the DAC to be initialized.
- Returns:
  Returns a boolean indicating if writing to the DAC is successful.
- Return values:

TRUE	If the file was read successfully
FALSE	If the file was read unsuccessfully

void DAC_Zero (void)

Sets the DAC output to mid-scale.

Remarks:
Requires SPI and the DAC to be initialized.
Returns:
Void

void DAC_ZeroOutput (void)

Sets the DAC output to zero.

Remarks:
Requires SPI and the DAC to be initialized.
Returns:
Void

11.5.7 FIFO.c File Reference

```
#include "STDDEF.h"
#include "FIFO.h"
```

Functions

- BOOL FIFO_MonPush (MON_FIFO *fifo, char ch)
  Pushes data into the FIFO queue.
- char FIFO_MonPop (MON_FIFO *fifo)
  Pops data from the FIFO queue.

Detailed Description

Author:
Kue Yang

Date:
11/22/2016

The FIFO module handles all FIFO related tasks. The FIFO module handles pushing and popping data into a given FIFO queue for processing.

Function Documentation

*char FIFO_MonPop (MON_FIFO * fifo)*

Pops data from the FIFO queue.

  - fifo The FIFO buffer that will be reading data from.

- Returns:

  Returns the next character in the FIFO buffer.

*BOOL FIFO_MonPush (MON_FIFO * fifo, char ch)*

Pushes data into the FIFO queue.

  - fifo The FIFO buffer that will be receiving data.
  - ch The data that will be inserted into the FIFO.

- Returns:

  Returns a boolean to indicate whether operation is successful or not.

- Return values:

TRUE	If pushing data to queue is successful.
FALSE	If pushing data to queue is unsuccessful.

11.5.8 FIFO.h File Reference

Data Structures

- struct MON_FIFO

  ***MON_FIFO** data structure.*

Macros

- #define **MON_BUFFERSIZE** 1024

Typedefs

- typedef struct **MON_FIFO MON_FIFO**

MON_FIFO *data structure.*

Functions

- char FIFO_MonPop (MON_FIFO *fifo)
  Pops data from the FIFO queue.
- BOOL FIFO_MonPush (MON_FIFO *fifo, char ch)
  Pushes data into the FIFO queue.

Detailed Description

Author:
Kue Yang
Date:
11/22/2016

Macro Definition Documentation

#define MON_BUFFERSIZE 1024

Defines the buffer size used for the FIFO queue.

Typedef Documentation

typedef struct MON_FIFO MON_FIFO

MON_FIFO data structure.
The FIFO data structure is used to create and store a FIFO queue.

Function Documentation

*char FIFO_MonPop (MON_FIFO * fifo)*

Pops data from the FIFO queue.

  - fifo The FIFO buffer that will be reading data from.
- Returns:
  Returns the next character in the FIFO buffer.

*BOOL FIFO_MonPush (MON_FIFO * fifo, char ch)*

Pushes data into the FIFO queue.

  - fifo The FIFO buffer that will be receiving data.
  - ch The data that will be inserted into the FIFO.

- Returns:

  Returns a boolean to indicate whether operation is successful or not.

- Return values:

TRUE	If pushing data to queue is successful.
FALSE	If pushing data to queue is unsuccessful.

11.5.9 FILEDEF.h File Reference

Macros

- #define **PIC1**
- #define MAX_NUM_OF_FILES 21
- #define **FILE_0** 0
- #define **FILE_1** 1
- #define **FILE_2** 2
- #define **FILE_3** 3
- #define **FILE_4** 4
- #define **FILE_5** 5
- #define **FILE_6** 6
- #define **FILE_7** 7
- #define **FILE_8** 8
- #define **FILE_9** 9
- #define **FILE_10** 10
- #define **FILE_11** 11
- #define **FILE_12** 12
- #define **FILE_13** 13
- #define **FILE_14** 14
- #define **FILE_15** 15
- #define **FILE_16** 16
- #define **FILE_17** 17
- #define **FILE_18** 18
- #define **FILE_19** 19
- #define **FILE_20** 20

Variables

- const char * fileNames [MAX_NUM_OF_FILES]

Detailed Description

Author:
Kue Yang
Date:
11/22/2016

Macro Definition Documentation

#define FILE_0 0

Defines file index 0.

#define FILE_1 1

Defines file index 1.

#define FILE_10 10

Defines file index 10.

#define FILE_11 11

Defines file index 11.

#define FILE_12 12

Defines file index 12.

#define FILE_13 13

Defines file index 13.

#define FILE_14 14

Defines file index 14.

#define FILE_15 15

Defines file index 15.

#define FILE_16 16

Defines file index 16.

#define FILE_17 17

Defines file index 17.

#define FILE_18 18

Defines file index 18.

#define FILE_19 19

Defines file index 19.

#define FILE_2 2

Defines file index 2.

#define FILE_20 20

Defines file index 20.

#define FILE_3 3

Defines file index 3.

#define FILE_4 4

Defines file index 4.

#define FILE_5 5

Defines file index 5.

#define FILE_6 6

Defines file index 6.

#define FILE_7 7

Defines file index 7.

#define FILE_8 8

Defines file index 8.

#define FILE_9 9

Defines file index 9.

#define MAX_NUM_OF_FILES 21

Defines the max number of audio files to be open.

PIC1

Defines the selected PIC.

Variable Documentation

fileNames

Stores the list of audio file names.

11.5.10 FILES.c File Reference

```
#include <p32xxxx.h>
#include <stdio.h>
#include "STDDEF.h"
#include "./fatfs/diskio.h"
#include "./fatfs/ffconf.h"
#include "./fatfs/ff.h"
#include "FILES.h"
```

Detailed Description

Author:

Kue Yang

Date:

11/22/2016

The **FILES** module will handle all file related tasks. Tasks includes: opening and closing files, searching for files and reading files. This module requires the Fatfs File System Library.

11.5.11 FILES.h File Reference

```
#include "./fatfs/ff.h"
```

Data Structures

- struct AUDIOINFO
- **AUDIOINFO** data structure. struct **FILES**
- **FILES** data structure.

Typedefs

- typedef struct AUDIOINFO AUDIOINFO
  ***AUDIOINFO** data structure.*
- typedef struct **FILES FILES**
  ***FILES** data structure.*

Functions

- FRESULT **FILES_ReadFile** (FIL *file, **BYTE** *buffer, **UINT16** bytes, **UINT16** *ptr)
  Reads a file.
- FRESULT **FILES_FindFile** (DIR *dir, FILINFO *fileInfo, const char *fileName)
  Finds a file.
- **BOOL FILES_ListFiles** (const char *selectedName)
  Displays a list of files.
- FRESULT **FILES_CloseFile** (FIL *file)
  Closes a file.
- FRESULT **FILES_OpenFile** (FIL *file, const char *fileName, int mode)
  Opens a file.

Detailed Description

Author:
Kue Yang
Date:
11/22/2016

Typedef Documentation

typedef struct AUDIOINFO AUDIOINFO

AUDIOINFO data structure.

The **AUDIOINFO** data structure is used to store the audio header data and file name.

typedef struct FILES FILES

FILES data structure.

The **FILES** data structure is used to store the file data used in conjunction with the Fatfs File System Library. The structure also stores the audio header data corresponding to the specified file.

Function Documentation

*FRESULT FILES_CloseFile (FIL * file)*

Closes a file.

Closes the specified file

- file The file data structure

- Returns:

  Returns a code indicating if a file successfully closes or not.

*FRESULT FILES_FindFile (DIR * dir, FILINFO * fileInfo, const char * fileName)*

Finds a file.

Finds a specific file

- dir The directory data structure
- fileInfo The file data structure
- fileName The file name to find

- Returns:

  Returns a code indicating if a file successfully found or not.

*BOOL FILES_ListFiles (const char * selectedName)*

Displays a list of files.

Displays a list of files and indicates which file is currently selected.

- selectedName The file name that is selected.

- Returns:

  Returns a boolean indicating if the list of files is found.

*FRESULT FILES_OpenFile (FIL * file, const char * fileName, int mode)*

Opens a file.

Opens the specified file with the specified access attributes.

- file The file data structure
- fileName The name of the file to be read
- mode The file access mode

- Returns:

  Returns a code indicating if a file successfully opens or not.

*FRESULT FILES_ReadFile (FIL * file, BYTE * buffer, UINT16 bytes, UINT16 * ptr)*

Reads a file.

Reads a specific file

    - file The file data structure
    - buffer The buffer to store the bytes read.
    - bytes The number of bytes to read
    - ptr A pointer to the number of bytes read

- Returns:

  Returns a code indicating if a file successfully read or not.

11.5.12 HardwareProfile.h File Reference

Defines System Configurations.

Macros

- #define **SYS_FREQ** (40000000L)
- #define **GetPeripheralClock**() (**SYS_FREQ**/(1 << OSCCONbits.PBDIV))
- #define GetInstructionClock() (SYS_FREQ)
- #define **CLEAR_WATCHDOG_TIMER** WDTCONbits.WDTCLR = 0x01;

Detailed Description

Defines System Configurations.

Author:
Kue Yang
Date:
11/22/2016

Macro Definition Documentation

#define CLEAR_WATCHDOG_TIMER WDTCONbits.WDTCLR = 0x01;

Clears the watchdog timer.

#define GetInstructionClock() (SYS_FREQ)

Calculates and returns the Instruction Clock Speed.

#define GetPeripheralClock() (SYS_FREQ/(1 << OSCCONbits.PBDIV))

Calculates and returns the Peripheral Clock Speed.

#define SYS_FREQ (40000000L)

Defines the System Clock Speed.

11.5.13 IO.c File Reference

```
#include <p32xxxx.h>
#include <stdio.h>
#include "HardwareProfile.h"
#include "STDDEF.h"
#include "IO.h"
```

Macros

- #define FRET_GROUP_COUNT 4
- #define FRETS_PER_GROUP 5

Functions

- void **IO_setGroupOutput** (int group)
  Sets the fret group to be scan.
- void **IO_Init** (void)
  Initializes the IO module.
- int IO_scanFrets (void)
  Scans a selection of frets.

Detailed Description

Author:

Kue Yang

Date:

11/22/2016

The IO module will handle all IO related tasks. The module will be initialize both the analog and digital IOs for all other modules (e.g. UART). The process of checking for finger placement for each frets is handled in this module.

Macro Definition Documentation

#define FRET_GROUP_COUNT 4

Defines the number of fret groups.

#define FRETS_PER_GROUP 5

Defines the number of frets per group.

Function Documentation

void IO_Init (void)

Initializes the IO module.

Initializes all pins used for all hardware modules in the application.

Returns:

Void

int IO_scanFrets (void)

Scans a selection of frets.

Scans all frets groups to determine which fret was pressed. Sets the currently selected fret to the fret that is pressed. Defaults to an open fret.

Returns:

Void

void IO_setGroupOutput (int group)

Sets the fret group to be scan.

- group The fret group to be scan.

- Returns:

Void

11.5.14 IO.h File Reference

Macros

- #define **GROUP4_OUT** PORTEbits.RE1
- #define **GROUP3_OUT** PORTGbits.RG14
- #define **GROUP2_OUT** PORTDbits.RD4
- #define **GROUP1_OUT** PORTGbits.RG13
- #define **FRET5** PORTGbits.RG1
- #define **FRET4** PORTGbits.RG0
- #define **FRET3** PORTAbits.RA6
- #define **FRET2** PORTAbits.RA7
- #define **FRET1** PORTEbits.RE0
- #define **ON_LED** PORTEbits.RE2
- #define **ERROR_LED** PORTEbits.RE3
- #define **INITIALIZE_LED** PORTEbits.RE4

Functions

- void **IO_Init** (void)
  Initializes the IO module.
- int IO_scanFrets (void)

Scans a selection of frets.

Detailed Description

Author:
Kue Yang
Date:
11/22/2016

Macro Definition Documentation

#define ERROR_LED PORTEbits.RE3

Defines the LED for ERROR.

#define FRET1 PORTEbits.RE0

Defines the input PORT for Fret 1

#define FRET2 PORTAbits.RA7

Defines the input PORT for Fret 2

#define FRET3 PORTAbits.RA6

Defines the input PORT for Fret 3

#define FRET4 PORTGbits.RG0

Defines the input PORT for Fret 4

#define FRET5 PORTGbits.RG1

Defines the input PORT for Fret 5

#define GROUP1_OUT PORTGbits.RG13

Defines the output PORT for Group 1

#define GROUP2_OUT PORTDbits.RD4

Defines the output PORT for Group 2

#define GROUP3_OUT PORTGbits.RG14

Defines the output PORT for Group 3

#define GROUP4_OUT PORTEbits.RE1

Defines the output PORT for Group 4

#define INITIALIZE_LED PORTEbits.RE4

Defines the LED for finishing INITIALIZATION.

#define ON_LED PORTEbits.RE2

Defines the LED for ON.

Function Documentation

void IO_Init (void)

Initializes the IO module.

Initializes all pins used for all hardware modules in the application.

Returns:

Void

int IO_scanFrets (void)

Scans a selection of frets.

Scans all frets groups to determine which fret was pressed. Sets the currently selected fret to the fret that is pressed. Defaults to an open fret.

Returns:

Void

11.5.15 main.c File Reference

The main entry point of the application.

```
#include <p32xxxx.h>
#include "plib/plib.h"
#include "HardwareProfile.h"
#include "STDDEF.h"
#include "IO.h"
#include "TIMER.h"
#include "ADC.h"
#include "SPI.h"
#include "UART.h"
#include "DAC.h"
#include "AUDIO.h"
```

Functions

- int **main** (void)

The main entry point of the application.

Detailed Description

The main entry point of the application.

Author:

Kue Yang

Date:

11/22/2016

Function Documentation

int main (void)

The main entry point of the application.

Returns:

An integer 0 upon exit success.

11.5.16 SPI.c File Reference

```
#include <p32xxxx.h>
#include "plib/plib.h"
#include "HardwareProfile.h"
#include "STDDEF.h"
#include "SPI.h"
```

Detailed Description

Author:

Kue Yang

Date:

11/22/2016

The SPI module will handle all SPI related tasks. The SPI module is used by the DAC module to write to the DAC.

11.5.17 SPI.h File Reference

Functions

- void **SPI_Init** (void)
  Initializes the SPI modules.
- BYTE SPI1_ReadWrite (BYTE)
  Reads and write data to SPI buffer.
- BYTE SPI2_ReadWrite (BYTE)
  Reads and write data to SPI buffer.
- void **SPI3_Init** (int clk)
  Initializes the SPI3 module.
- BYTE SPI3_ReadWrite (BYTE ch)
  Reads and write data to SPI buffer.

Detailed Description

Author:
Kue Yang
Date:
11/22/2016

Function Documentation

BYTE SPI1_ReadWrite (BYTE ch)

Reads and write data to SPI buffer.

- ch The data to be written to SPI buffer.

- Returns:

  Returns the data read from the SPI buffer.

BYTE SPI2_ReadWrite (BYTE ch)

Reads and write data to SPI buffer.

- ch The data to be written to SPI buffer.

- Returns:

  Returns the data read from the SPI buffer.

void SPI3_Init (int clk)

Initializes the SPI3 module.

- clk The clock rate to set the SPI module to.

- Returns:

  Void.

BYTE SPI3_ReadWrite (BYTE ch)

Reads and write data to SPI buffer.

- ch The data to be written to SPI buffer.

- Returns:

  Returns the data read from the SPI buffer.

void SPI_Init (void)

Initializes the SPI modules.

Returns:

Void.

11.5.18 STDDEF.h File Reference

Macros

- #define **TRUE** 1
- #define **FALSE** 0
- #define **BOOL** int
- #define **INT32_MAX_NUM** 1<<31
- #define **INT16_MAX_NUM** 1<<15

Typedefs

- typedef unsigned char **UINT8**

Typedef definition for UINT8.

- typedef unsigned short **UINT16**

Typedef definition for UINT16.

- typedef unsigned long **UINT32**

Typedef definition for UINT32.

- typedef unsigned char **BYTE**

Typedef definition for BYTE datatype.

- typedef unsigned short **WORD**

Typedef definition for WORD datatype.

- typedef unsigned long **DWORD**

Typedef definition for DWORD datatype.

Detailed Description

Author:

Kue Yang

Date:

11/22/2016

Macro Definition Documentation

#define BOOL int

Boolean datatype definition.

#define FALSE 0

Boolean datatype.

#define INT16_MAX_NUM 1<<15

Defines the max value for a 16-bit variable.

#define INT32_MAX_NUM 1<<31

Defines the max value for a 32-bit variable.

#define TRUE 1

Boolean datatype.

11.5.19 TIMER.c File Reference

```
#include <p32xxxx.h>
#include "plib/plib.h"
#include "HardwareProfile.h"
#include "STDDEF.h"
#include "./fatfs/diskio.h"
#include "AUDIO.h"
#include "TIMER.h"
```

- #define **ONE_MS_PERIOD** 40000

Detailed Description

Author:

Kue Yang

Date:

11/22/2016

The TIMER module will handle timers and delays used in the application.

Macro Definition Documentation

#define ONE_MS_PERIOD 40000

Timer 1 Period for one ms.

11.5.20 TIMER.h File Reference

```
#include "STDDEF.h"
```

Functions

- void **TIMER_Init** (void)

Initializes all timer modules.

- void TIMER_Process (void)

Runs timer related operations.

- UINT32 TIMER_GetMSecond (void)

Returns the millisecond count.

- void TIMER_MSecondDelay (int)

Delays the application for a given set time.

- BOOL TIMER1_IsON (void)

Checks if Timer 1 is on/off.

- void TIMER1_ON (BOOL ON)

Toggles on/off Timer 1.

- BOOL TIMER3_IsON (void)

Checks if Timer 3 is on/off.

- void TIMER3_ON (BOOL ON)

Toggles on/off Timer 3.

- Void TIMER3_SetSampleRate (UINT16 sampleRate)

Sets the Timer 3 period.

Detailed Description

Author:
Kue Yang
Date:
11/22/2016

Function Documentation

BOOL TIMER1_IsON (void)

Checks if Timer 1 is on/off.

Returns:

Returns a boolean indicating if Timer 1 is on/off.

Return values:

TRUE	Timer 1 is on.
FALSE	Timer 1 is off.

void TIMER1_ON (BOOL_ON)

Toggles on/off Timer 1.

- ON Toggles the timer on/off (TRUE/FALSE).

- Returns:

Void

BOOL TIMER3_IsON (void)

Checks if Timer 3 is on/off.

Returns:

Returns a boolean indicating if Timer 3 is on/off.

Return values:

TRUE	Timer 3 is on.
FALSE	Timer 3 is off.

void TIMER3_ON (BOOL ON)

Toggles on/off Timer 3.

- ON Toggles the timer on/off (TRUE/FALSE).
- Returns:

Void

void TIMER3_SetSampleRate (UINT16 sampleRate)

Sets the Timer 3 period.

- sampleRate The sample rate to set Timer 3 at.
- Returns:

Void

UINT32 TIMER_GetMSecond (void)

Returns the millisecond count.

Returns:

Returns the millisecond count of the application since startup.

void TIMER_Init (void)

Initializes all timer modules.

Returns:

Void

void TIMER_MSecondDelay (int timeDelay)

Delays the application for a given set time.

- timeDelay The delay in milliseconds.
- Returns:

Void

void TIMER_Process (void)

Runs timer related operations.

Returns:

Void

11.5.21 UART.c File Reference

```
#include <p32xxxx.h>
#include "plib/plib.h"
#include <stdio.h>
#include <string.h>
#include "HardwareProfile.h"
#include "STDDEF.h"
#include "Timer.h"
#include "FIFO.h"
#include "DAC.h"
#include "AUDIO.h"
#include "UART.h"
```

Macros

- #define **DESIRED_BAUDRATE** (19200)
- #define WRITE_BUFFER_SIZE 128
- #define **CMD_SIZE** 16
- #define DESCRIPTION_SIZE (WRITE_BUFFER_SIZE-CMD_SIZE)

Functions

- int **UART_GetBaudRate** (int desireBaud)

Calculates the baud rate configuration.

- BOOL UART_isBufferEmpty (MON_FIFO *buffer)

Checks if the specified buffer is empty.

- char UART_getNextChar (MON_FIFO *buffer)

Pops a character from the buffer.

- void **UART_putNextChar** (**MON_FIFO** *buffer, char ch)

Pushes the specified character into the buffer.

- void UART_processCommand (void)

Processes commands received by the UART module.

- int MON_parseCommand (COMMANDSTR *cmd, MON_FIFO *buffer)

Parses the received commands.

- **COMMANDS MON_getCommand** (const char *cmdName)

Gets the handler for the specified command.

- void **MON_GetHelp** (void)

Displays the list of available commands.

- void MON_GetFileList (void)

Command used to display a list of files.

- void **MON_Set_File** (void)

Command used to select a specific file.

- void MON_Reset_File (void)

Command used to reset the selected file pointer.

- void MON_Read_File (void)

Command used to read the selected file.

- void **MON_TestDAC** (void)

Command used to write data to the DAC.

- void **MON_ZeroDAC** (void)

Command used to zero the DAC's output.

- void **MON_SinDAC** (void)

Command used to write a sin wav to the DAC.

- void MON_Timer_ON_OFF (void)

Command used to Toggle on/off the Timer 3 module.

- void MON_Timer_Get_PS (void)

Command used to display a Timer 3 period.

- void MON_Timer_Set_PS (void)

Command used to set Timer 3 period.

- void **UART_Init** (void)

Initialize the UART module.

- void **UART_Process** (void)

Processes all UART related tasks.

- void **MON_SendChar** (const char *character)

Transmit a character.

- void **MON_SendString** (const char *str)

Transmit a string.

- void **MON_SendStringNR** (const char *str)

Transmit a string.

- char **MON_lowerToUpper** (const char *ch)

Changes character to undercase.

- **BOOL MON_stringsMatch** (const char *str1, const char *str2)

Checks if two strings matches.

- void **MON_removeWhiteSpace** (const char *string)

Removes white spaces from a string.

- UINT16 MON_getStringLength (const char *string)

Gets the length of a string.

- void **__ISR** (_UART1_VECTOR, IPL2AUTO)

The UART1 Interrupt Service Routine.

Variables

- COMMANDSTR cmdStr
- MON_FIFO rxBuffer
- MON_FIFO txBuffer
- BOOL cmdReady
- UINT16 actualBaudRate

- UINT16 numOfCmds
- COMMANDS MON_COMMANDS []

Detailed Description

Author:

Kue Yang

Date:

11/22/2016

The UART module will handle serial communication. The module will be used to receive commands via serial communications and can be use as a debugging tool. The serial communication uses the rs232 serial interface.

Macro Definition Documentation

#define CMD_SIZE 16

The size of the command name string.

#define DESCRIPTION_SIZE (WRITE_BUFFER_SIZE-CMD_SIZE)

The size of command description string.

#define DESIRED_BAUDRATE (19200)

The desired UART baud rate.

#define WRITE_BUFFER_SIZE 128

The buffer size for writing.

Function Documentation

void __ISR (_UART1_VECTOR , IPL2AUTO)

The UART1 Interrupt Service Routine.

The UART1 interrupt service routine will handle receiving data. If data is received, the data is pushed into a FIFO queue for later processing. If data is received is a return key, the received data has ended and the command is ready flag is set.

Returns:

Void.

*COMMANDS MON_getCommand (const char * cmdName)*

Gets the handler for the specified command.

  - cmdName The name of the command.
- Returns:
  Returns the command handler.

void MON_GetFileList (void)

Command used to display a list of files.

Returns:

Void.

void MON_GetHelp (void)

Displays the list of available commands.

Commands Handlers.

Returns:

Void.

*UINT16 MON_getStringLength (const char * _string)*

Gets the length of a string.

- string The string that is being modified.

- Returns:

Returns the string length.

*char MON_lowerToUpper (const char * _ch)*

Changes character to undercase.

Changes character to undercase.

- ch The character to undercase.

- Returns :

The undercase character

*int MON_parseCommand (COMMANDSTR * _cmd, MON_FIFO * _buffer)*

Parses the received commands.

- cmd The command data structure used to store the command.
- buffer The receive buffer.

- Returns:

Returns the number of command arguments.

void MON_Read_File (void)

Command used to read the selected file.

Returns:

Void.

*void MON_removeWhiteSpace (const char * _string)*

Removes white spaces from a string.

- string The string that is being modified.

- Returns:

Void.

void MON_Reset_File (void)

Command used to reset the selected file pointer.

Returns:

Void.

*void MON_SendChar (const char * character)*

Transmit a character.

Add character to transmit fifo buffer.

- character The character to transmit.

- Returns:

Void.

*void MON_SendString (const char * str)*

Transmit a string.

Adds string to transmit fifo buffer.

- str The string to transmit.
- Returns:

Void.

*void MON_SendStringNR (const char * str)*

Transmit a string.

Transmit a string without a newline or return character.

- str The string to transmit.

- Returns:

Void.

void MON_Set_File (void)

Command used to select a specific file.

Returns:

Void.

void MON_SinDAC (void)

Command used to write a sin wav to the DAC.

Returns:

Void.

*BOOL MON_stringsMatch (const char * str1, const char * str2)*

Checks if two strings matches.

- str1 The first string to compare.
- str2 The second string to compare.

- Returns:

Returns a boolean indicating if the strings matches.

- Return values:

TRUE	if both strings matches.
FALSE	if the strings does not match.

void MON_TestDAC (void)

Command used to write data to the DAC.

Returns:

Void.

void MON_Timer_Get_PS (void)

Command used to display a Timer 3 period.

Returns:

Void.

void MON_Timer_ON_OFF (void)

Command used to Toggle on/off the Timer 3 module.

Returns:

Void.

void MON_Timer_Set_PS (void)

Command used to set Timer 3 period.

Returns:

Void.

void MON_ZeroDAC (void)

Command used to zero the DAC's output.

Returns:

Void.

int UART_GetBaudRate (int desireBaud)

Calculates the baud rate configuration.

UART Helper Functions.

- desireBaud The desired baud rate.

- Returns:

Returns the baud rate for the given peripheral clock.

*char UART_getNextChar (MON_FIFO * buffer)*

Pops a character from the buffer.

- buffer The buffer to pop the character from.
  - Returns:

    Returns the character that is popped off from the buffer.

void UART_Init (void)

Initialize the UART module.

Returns:

Void

*BOOL UART_isBufferEmpty (MON_FIFO * buffer)*

Checks if the specified buffer is empty.

FIFO helper functions.

- buffer The buffer that is being checked.

- Returns:

  Returns a boolean indicating if the buffer is empty.

  - Return values:

TRUE	if the buffer is empty.
FALSE	if the buffer is not empty.

void UART_Process (void)

Processes all UART related tasks.

Returns:

Void.

void UART_processCommand (void)

Processes commands received by the UART module.

Command Helper Functions.

Returns:

Void.

*void UART_putNextChar (MON_FIFO * buffer, char ch)*

Pushes the specified character into the buffer.

- buffer The buffer used to store the character.
- ch The character to push into buffer.

- Returns:

  Void.

Variable Documentation

actualBaudRate

The configured UART baud rate.

cmdReady

The UART command receive flag.

cmdStr

The command string.

MON_COMMANDS

```
           Initial value:= {
                  {"HELP", " Display the list of commands avaliable. ", MON_GetHelp},
                  {"LIST", " Lists all WAV files. ", MON_GetFileList},
                  {"SET", " Sets the file to read. FORMAT: SET fileName.", MON_Set_File},
                  {"RESET", " Resets the file pointer to beginning of file.",
                  MON_Reset_File},
                  {"READ", " Reads numOfBytes from current file. Read from beginning of the
                  file if reset is set to 1. FORMAT: READ reset numOfBytes.",
                  MON_Read_File},
                  {"DAC", " Sets an output value on the DAC. MIN: 0, MAX: 65535. FORMAT: DAC
                  value. ", MON_TestDAC},
                  {"ZERO", " Sets all DAC outputs to zero. ", MON_ZeroDAC},
                  {"SIN", " Tests the DAC using a sin wave. ", MON_SinDAC},
                  "TONE", " Toggles on/off the Audio Timer. ", MON_Timer_ON_OFF},
                  {"PDG", " Get the current period set on timer 3. FORMAT: PDG.",
                  MON_Timer_Get_PS},
                  {"PDS", " Configures the timer period. FORMAT: PDS period .",
                  MON_Timer_Set_PS},
                  {"", "", NULL}
}
```

The list of commands.

numOfCmds

The number of commands.

rxBuffer

The UART receive buffer.

txBuffer

The UART transmit buffer.

11.5.22 UART.h File Reference

Data Structures

- struct COMMANDS
- **COMMANDS** data structure. struct **COMMANDSTR**
  COMMANDSTR data structure.

Typedefs

- typedef struct **COMMANDS COMMANDS**
  COMMANDS *data structure.*
- typedef struct COMMANDSTR COMMANDSTR
  COMMANDSTR *data structure.*

Functions

- void **UART_Init** (void)

Initialize the UART module.

- void **UART_Process** (void)

Processes all UART related tasks.

- void **MON_removeWhiteSpace** (const char *string)

Removes white spaces from a string.

- UINT16 MON_getStringLength (const char *string)

Gets the length of a string.

- **BOOL MON_stringsMatch** (const char *str1, const char *str2)

Checks if two strings matches.

- char **MON_lowerToUpper** (const char *ch)

Changes character to undercase.

- void **MON_SendStringNR** (const char *str)

Transmit a string.

- void **MON_SendString** (const char *str)

Transmit a string.

- void **MON_SendChar** (const char *character)

Transmit a character.

Detailed Description

Author:
Kue Yang
Date:
11/22/2016

Typedef Documentation

typedef struct COMMANDS COMMANDS

COMMANDS data structure.

The **COMMANDS** data structure is used to store a command with its corresponding command handler. The command handler is used in conjunction with the serial communication for processing a command that is received from the user.

typedef struct COMMANDSTR COMMANDSTR

COMMANDSTR data structure.

The command string data structure is used to store a command with its corresponding arguments.

Function Documentation

*UINT16 MON_getStringLength (const char * string)*

Gets the length of a string.

- string The string that is being modified.

- Returns:

  Returns the string length.

*char MON_lowerToUpper (const char * ch)*

Changes character to undercase.

Changes character to undercase.

- ch The character to undercase.

- Returns:

  The undercase character

*void MON_removeWhiteSpace (const char * string)*

Removes white spaces from a string.

- string The string that is being modified.

- Returns:

  Void.

*void MON_SendChar (const char * character)*

Transmit a character.

Add character to transmit fifo buffer.

- character The character to transmit.

- Returns:

  Void.

*void MON_SendString (const char * str)*

Transmit a string.

Adds string to transmit fifo buffer.

- str The string to transmit.

- Returns:

  Void.

*void MON_SendStringNR (const char * str)*

Transmit a string.

Transmit a string without a newline or return character.

- str The string to transmit.

- Returns:

  Void.

*BOOL MON_stringsMatch (const char * str1, const char * str2)*

Checks if two strings matches.

  - str1 The first string to compare.
  - str2 The second string to compare.

- Returns:

  Returns a boolean indicating if the strings matches.

- Return values:

TRUE	if both strings matches.
FALSE	if the strings does not match.

void UART_Init (void)

Initialize the UART module.

Returns:

Void

void UART_Process (void)

Processes all UART related tasks.

Returns:

Void.

11.5.23 WAVDEF.h File Reference

Macros

- #define WAV_HEADER_SIZE 44
- #define WAV_CHUNK_ID 0
- #define WAV_CHUNK_SIZE 4
- #define **WAV_FORMAT** 8
- #define WAV_CHUNK1_SUB_ID 12
- #define WAV_CHUNK1_SUB_SIZE 16
- #define WAV_AUDIO_FORMAT 20
- #define WAV_NUM_CHANNELS 22
- #define WAV_SAMPLE_RATE 24
- #define WAV_BYTE_RATE 28
- #define WAV_BLOCK_ALIGN 32
- #define WAV_BITS_PER_SAMPLE 34
- #define WAV_CHUNK2_SUB_ID 36
- #define WAV_CHUNK2_SUB_SIZE 40
- #define **WAV_DATA** 44
- #define **WAV_SUCCESS** 1
- #define WAV_HEADER_SIZE_ERROR 2
- #define WAV_CHUNK_ID_ERROR 3
- #define WAV_FORMAT_ERROR 4
- #define WAV_SUB_CHUNK1_ID_ERROR 5
- #define WAV_SUB_CHUNK1_SIZE_ERROR 6
- #define WAV_SUB_CHUNK2_ID_ERROR 7

Detailed Description

Author:
Kue Yang
Date:
11/22/2016

Macro Definition Documentation

#define WAV_AUDIO_FORMAT 20

Defines the index of the WAV audio format.

#define WAV_BITS_PER_SAMPLE 34

Defines the index of the WAV bits per sample.

#define WAV_BLOCK_ALIGN 32

Defines the index of the WAV block alignment.

#define WAV_BYTE_RATE 28

Defines the index of the WAV byte rate.

#define WAV_CHUNK1_SUB_ID 12

Defines the index of the WAV chunk1 sub ID.

#define WAV_CHUNK1_SUB_SIZE 16

Defines the index of the WAV chunk1 sub size.

#define WAV_CHUNK2_SUB_ID 36

Defines the index of the WAV chunk2 sub ID.

#define WAV_CHUNK2_SUB_SIZE 40

Defines the index of the WAV chunk2 sub size.

#define WAV_CHUNK_ID 0

Defines the index of the WAV chunk ID.

#define WAV_CHUNK_ID_ERROR 3

Defines the error code for invalid chunk ID.

#define WAV_CHUNK_SIZE 4

Defines the index of the WAV chunk size.

#define WAV_DATA 44

Defines the index of the WAV data.

#define WAV_FORMAT 8

Defines the index of the WAV format.

#define WAV_FORMAT_ERROR 4

Defines the error code for invalid WAV format.

#define WAV_HEADER_SIZE 44

Defines the size of a WAV file header.

#define WAV_HEADER_SIZE_ERROR 2

Defines the error code for the wrong header size.

#define WAV_NUM_CHANNELS 22

Defines the index of the WAV number of channels.

#define WAV_SAMPLE_RATE 24

Defines the index of the WAV sample rate.

#define WAV_SUB_CHUNK1_ID_ERROR 5

Defines the error code for invalid chunk1 ID.

#define WAV_SUB_CHUNK1_SIZE_ERROR 6

Defines the error code for invalid chunk1 size.

#define WAV_SUB_CHUNK2_ID_ERROR 7

Defines the error code for invalid chunk2 ID.

#define WAV_SUCCESS 1

Defines the success code when accessing a WAV header.