Assistive Aid for Playing the Ukulele by Persons with Duchenne Muscular Dystrophy

A Major Qualifying Project Report

Submitted to the Faculty

of the

Worcester Polytechnic Institute

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

in Electrical and Computer Engineering

By

Eric Lacroix

Alexander Sylvia

Ruxue Yang

and in Biomedical Engineering

By

Stephanie Arce

Krisha Nazareth

April 28, 2016

Approved by:
Professor Edward A. Clancy, Advisor
Electrical and Computer Engineering/Biomedical Engineering, WPI

Professor Stephen J. Bitar, Advisor
Electrical and Computer Engineering, WPI

Abstract

Duchenne Muscular Dystrophy (DMD) is a genetic degenerative muscle disease that occurs primarily in males. Since there is no cure, working towards improving the quality of life for those affected by DMD is important, especially due to the progressive nature of the disease. This project produced an assistive aid that allows a person with DMD to play the ukulele despite gradual muscle degeneration. A custom hand controller was built with sensors that react to changes in finger-tip force. These changes are processed using a microcontroller, and the microcontroller signals electronic actuators to depress ukulele strings, much like one's fingers would press the strings on the neck of a ukulele. The user is able to program the device to either play single notes or entire chords. The device works as an assistive aid because it eliminates the need for the user's left hand to move along the neck of the instrument in order to successfully play it, allowing for an easier way to play the instrument. As a result, the device may also benefit those with similar muscular deficits, including stroke patients.

Statement of Authorship

All team members contributed to the design and implementation of the device. All members also acted as main authors and editors of this paper. Sections written by each individual are listed below:

Eric Lacroix: Executive Summary, 2.1, 3.3.1, 3.3.3, 3.3.4, 4.2.1, 4.2.5, 4.2.6, 4.2.7, 5.6.2

Alex Sylvia: 2.5, 3.3.2, 4.2.3, 4.2.4, 4.3, 5.6.3, Conclusion, User Guide

Ruxue Yang: Introduction, 2.4, 2.6.2, 3.3.2, 5.5, 5.6.3, formatting

Stephanie Arce: 2.3, 2.6.3, 2.7, 3.1, 3.2, formatting

Krisha Nazareth: Abstract, 2.2, 2.6.1, 2.8, 3.2, 3.3.1(Clay Mold iterations), 3.4, 4.1, 4.2.2, 4.2.8, 5.1, 5.2, 5.3, 5.6.1

Benjamin Rogers contributed to this project during terms A and B 2015

Acknowledgements

The team would like to thank Professors Stephen Bitar and Edward (Ted) Clancy for their invaluable advice, guidance, and support as we worked to overcome several obstacles while creating this device.

The team would also like to thank Professor Gene Bogdanov, William Appleyard, Lisa Wall, and Tom Partington for their help and support in completing this project.

Finally, the team would like to thank the subject for whom the assistive ukulele was made for his cooperation and patience throughout the completion of the project.

Executive Summary

Approximately 1 of every 5,600 to 7,700 males is affected by Duchenne Muscular Dystrophy (DMD), a degenerative muscle disease caused by a genetic mutation in the DMD gene. Symptoms are apparent in children and many are often in a wheelchair by 12 years of age. Currently there is no cure for DMD, but improving the quality of life for people with DMD is still a necessity.

The goal of this project was to create an assistive aid that would allow patients with DMD to play a musical instrument: the ukulele. The ukulele was chosen based on conversations with the team's subject. The ukulele was chosen based of the subject's interest and inability to wrap his wrist around the ukulele neck. As a result, the team strived to create a working, ergonomic prototype for the subject that would give him the ability to finger the ukulele but still allows him to strum. The progression of DMD was taken into consideration and worked features into the prototype that would allow the subject to continue to use the musical device as the disease worsens.

Two main objectives were identified for the final design. The device needed to be user-friendly and produce the same sound as if an individual's fingers were holding down the strings of the ukulele. Replaceable parts were desirable in the prototype so as to be easily replaceable just in case a part of the prototype was not working properly. It was imperative that the sound produced by the prototype was similar to the sound of an individual holding the strings of a ukulele down with his own fingers and strumming. If the prototype failed to simulate a person playing an actual ukulele, the subject may not want to use the device.

There were also several constraints identified for the design of the prototype. Due to the nature of the disease, the subject was not able to exert a lot of energy over a long period of time. Therefore, the device needed to be designed so the subject did not overexert his muscles while playing. The team also needed to make sure that the strings of the ukulele were being depressed fully in order to produce an acceptable sound when that particular string was strummed. If this constraint was not taken into consideration, the sound produced by the ukulele would not be accurate. The team came to the conclusion that all major, minor, and diminished chords were able to be played within the first four frets of the ukulele. Therefore, the prototype was restricted to working within the limits of the first four frets on the ukulele.

To design the device, the team split the project into three parts: input hardware, microcontroller/software architecture, and output hardware. The input hardware consisted of a way for the user to interface with the microcontroller. The microcontroller/software architecture portion utilized software to detect an input from the user and select an output based on that input. The output hardware used the output of the microcontroller to actuate hardware over the neck of the ukulele to press the strings.

For input hardware, the 24V power supply was chosen because the device could potentially interface with the subject's 24V power chair. Force sensitive resistors (FSRs) were selected as input sensors to detect finger forces. FSRs were favorable over other options because they were cost effective and had the ability to change their outputs as the disease progressed in the subject. The FSRs were set up in a voltage divider configuration with a 51kΩ resistor and an operational amplifier acting as a unity gain buffer (see Figure 1). These FSRs were placed on an ergonomic clay mold that was shaped to fit the subject's left hand.

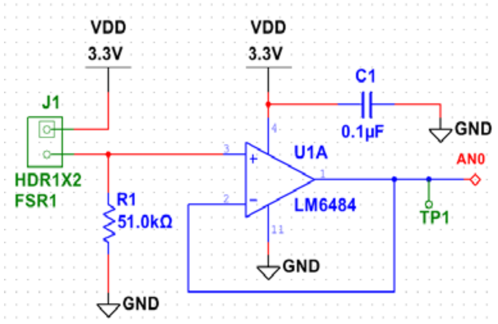


Figure 1: FSR Circuitry

The microcontroller chosen for this project was the PIC18F46K20 (PIC) for its price and ease of programming. Figure 2 shows the setup of the PIC for this project's application. Considering the traditional style of playing a ukulele is by strumming a chord for multiple measures, the change between frets is slow, allowing for the sampling rate for this application to be set to under 1 kHz. The force threshold is set by the microcontroller to detect an intentional actuation of the FSR. An algorithm was created to detect a combination of FSRs pressed and to output the proper chord. The combination of FSRs is mapped to a chord, and the solenoids corresponding to that chord could depress the appropriate strings. This algorithm also accepted the musical key the user wanted to play in and was able to change the chords based on the key selection. A USB interface was included in this application to switch between playing modes so that the user can play chords or individual notes on a string. Another feature of the USB

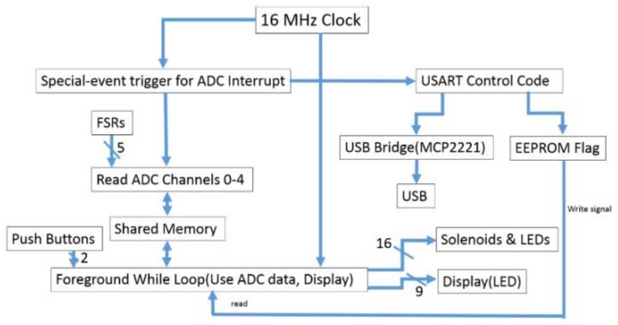


Figure 2: Software Architecture

was the ability to change the threshold of the FSRs so that they may be customized to the user’s force capabilities as the disease progresses.

The output hardware consisted of a switching circuit and solenoids, which were chosen because they supplied enough force to depress the string of the ukulele. The switching circuit, seen in Figure 3, utilized a power MOSFET to actuate the solenoids when the gate of the MOSFET received a high input. Light emitting diodes (LEDs) were also implemented in the design to indicate which solenoids were being actuated at any given time. With solenoids selected as the method of actuation, a solenoid stand needed to be made in order to hold all 16 solenoids. The solenoids were placed on the stand using a staggering method. In addition to the rest of the design, the team also included a way for the subject to select what key he would like to play in with two tactile buttons. To assist with the excess of wires, the team designed two printed circuit boards (PCBs) to simplify the number of wires.

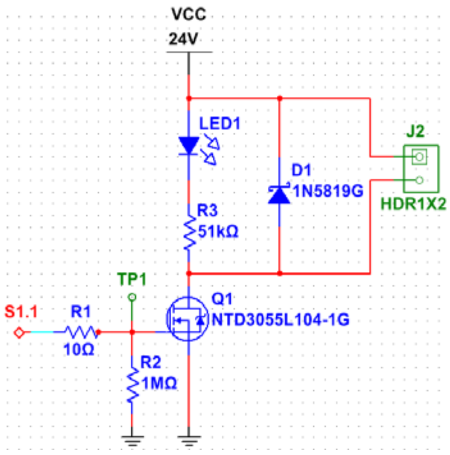


Figure 3: Switching Circuitry

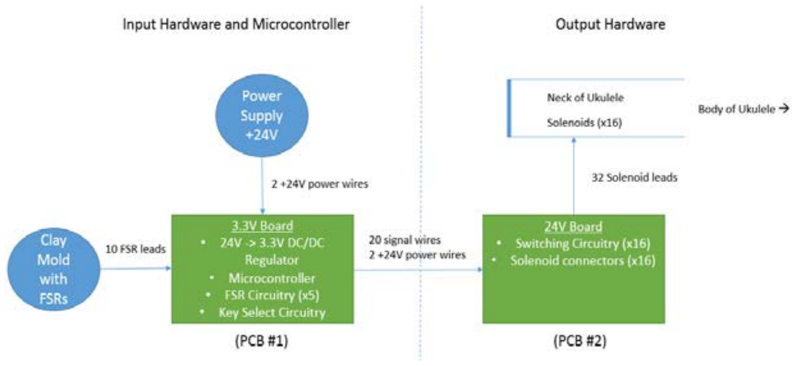


Figure 4: Global Architecture of the Prototype

Putting input hardware, microcontroller, and output hardware together, the team came up with a final design for the prototype. The global architecture in Figure 4 shows how all of the circuitry integrated together. When the user pressed an FSR with the fingers on his left hand, a chord would be actuated by the solenoids. Based on the combination of the FSRs that the subject presses, different chords were played.

Prior to assembling all of the circuitry and integrating them together, the team performed a series of tests on each circuit. These tests were used to ensure that the team was receiving the outputs that they were expecting. Test results were proven by observing the circuits at various points and seeing the signals produced on an oscilloscope. When the team felt confident that each circuit was working as designed, all of the circuitry was assembled together.

In order to meet the sound objective, the team needed to test the sound that the solenoids produced when they were actuated on the strings. In order to provide data to compare the solenoid sounds to, the team researched the theoretical frequency for the pitch of each note on the ukulele. In addition, the team used a tuning app on a smartphone to record data while using their fingers to press down on the strings. The solenoids were able to produce the same note as the theoretical notes for 14 out of the 16 solenoids. A portion of the sound results are shown in Table 1. All sound testing results can be seen in Appendix C.

Table 1: Sound Testing for the C String

C String					
Theoretical Pitch (Hz)	Note	Finger Pressed Pitch (Hz)	Finger Note	Solenoid Pressed Pitch (Hz)	Solenoid Note
277.18	C#/Db	283.5	C#	276	C#
293.66	D	298.9	D	294.1	D
311.13	D#/Eb	316.8	D#	309.9	D#
329.63	E	335.5	E	325.2	E

After the team tested the design all together and had a working prototype, they brought the device to the subject to perform subject testing as well as receive feedback from the subject. The subject assisted the team in appropriately setting thresholds for each individual FSR. The subject also provided feedback to the team, mentioning that continuously pressing an FSR could become tiresome. He recommended a software implementation of a "latching" method, where pressing an FSR once would actuate a chord until another FSR is pressed. This method would relieve the subject from having to continuously apply force to an FSR in order to play a chord. Though the team originally had the ukulele oriented so the subject was able to strum away from his body, the subject preferred a different orientation. He would rather strum from left to right with the neck of the ukulele perpendicular to his body. A picture of the preferred orientation is shown in Figure 5.

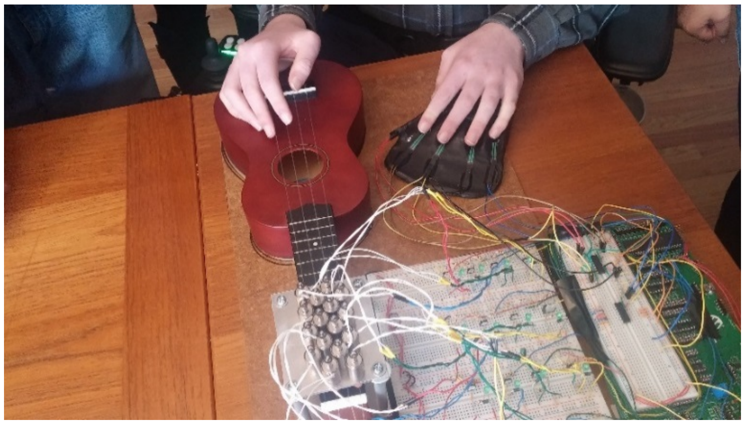


Figure 5: Subject's preferred orientation

Overall, the team was able to accomplish the objectives set at the start of the project. The team was successfully able to make the prototype user-friendly in a number of ways. There were additional features, such as the key select, that allowed the subject to pick which key he would like to play in. The clay mold was also shaped to the subject's left hand, allowing the subject to feel comfortable when playing the device. The team also effectively reproduced most of the same notes using the solenoids as they did with their own fingers.

In addition to completing the design objectives, the team designed a device adhering to the constraints set forth by the disease. The FSRs and microcontroller allowed the threshold of the sensors to be changed, which benefits the subject as the disease progresses. The team also implemented an actuation method that fit within the size limitations of the neck of the ukulele. The "latching" method gathered as a result from subject testing allowed the team to improve the usability of the device and prevented overexertion of the subject's muscles.

Though a working prototype was produced, the team felt as though there could still be some improvements made to the design. As the subject's muscles continue to degenerate, the clay mold will continuously need to be reshaped to the subject's hands. Additionally, the use of wireless technology could be implemented into the clay mold design in order to give the subject the freedom to place the controller in a more comfortable position.

In conclusion, the team was successfully able to produce a working assistive aid prototype. This device allowed the subject to play a multitude of chords based on a key selection input. The design catered to the subject's available range of motion and is designed to continue to do so. Other individuals with permanent or progressive muscle damage can also benefit from this device. Overall, the completion of the project assisted the subject in achieving his goals of creating music with a ukulele.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

Duchenne Muscular Dystrophy (DMD) is a degenerative muscle disease that results in premature death. This disease affects 1 in 5,600 to 7,700 male births. Patients diagnosed with DMD have a mutation in the gene located on the X chromosome, a gene that is responsible for producing a protein known as dystrophin. This mutated gene fails to produce functional dystrophin, which plays an important role in muscle regeneration. As a result, patients with DMD gradually lose strength in both voluntary and involuntary muscles. As the disease progresses, patients lose the ability to use their muscles. A patient's inability to use his muscles spreads from the patient's legs and pelvis to the patient's arms, neck and other areas, affecting his day-to-day activities and hobbies. The goal of this project is to help a patient with DMD who has a passion for music continue to explore his hobby of making and playing music. The team's subject and client is a patient with DMD who is currently in a wheelchair and has difficulty raising his arms. Despite his disability, the patient still plays music and tries to learn new instruments. The purpose of this project is to create an assistive aid that allows the subject to play the ukulele without struggling with the fact that he does not have enough strength to depress strings and, as a result, play chords.

This report discusses the nature of muscular diseases, the fundamentals of music, and musical assistive aids that have been conceptualized and implemented in the past. In addition, the report highlights the design objectives and constraints of the device set forth by the team, and covers the hardware and software design involved in the creation of the device. It also covers the implementation of the project, as well as the results obtained by testing the device, both with and without the subject's involvement. Finally, the report concludes by discussing the team's accomplishments, as well as its suggestions for implementations of the device.

2. Background

This section provides a foundation for understanding the design process for an ergonomic musical assistive device built for a patient with Duchenne Muscular Dystrophy (i.e., a patient with limited upper extremity function). This section also provides information regarding DMD, its causes, symptoms, and progression. It briefly explores populations such as post-stroke patients and Amyotrophic Lateral Sclerosis (ALS) patients, and how these populations could benefit from such a device as well. It will explain the scales of measurement used for both diagnostic purposes, as well as for measuring the severity of functional loss in both DMD and stroke patients. Since one of the main goals of this project was to create some sort of musical assistive device, musical instruments, as well as existing assistive musical devices will also be explored in this chapter, allowing for a better sense of how the subject's preferred music instruments work. It will also introduce what exists on the market in terms of what the team intends on creating. Finally, this chapter will provide information on options considered for the creation of different parts of the device (i.e. sensors and pre-existing schematics).

2.1 Duchenne Muscular Dystrophy

Duchenne Muscular Dystrophy is “a genetic disorder characterized by progressive muscle degeneration and weakness” [MDA, 2015a]. It is the most common form of muscular dystrophy, accounting for around 50% of all muscular dystrophy cases [NINDS, 2015]. A study from 2007 shows that approximately 1 of every 5,600 to 7,700 males are affected by DMD [NCBDDD, 2015]. It typically affects males because the disease is X-linked recessive, meaning that the gene responsible for DMD affects a particular gene on the X chromosome. Since men have one Y chromosome and one X chromosome, DMD is dominant in men whereas DMD is recessive in women because women have two X chromosomes. Women may have the defective gene on one X chromosome and not on the other, but will only be considered carriers and not exhibit any symptoms of DMD. This particular type of muscular dystrophy affects an individual’s ability to repair muscle, causing their muscles to grow weaker as time goes on.

In the typical muscle repair process, undifferentiated muscle cells called satellite cells work to repair the muscle. Studies have shown that satellite cells are “normally quiescent and are activated only in response to growth or muscle damage” [Grounds *et al.*, 2002]. Once these satellite cells are activated, proliferated and differentiated, they fuse together to form small muscle cells called myoblasts. The myoblasts then fuse together to create small muscle cells, or myotubes. The myotubes continue to differentiate and fuse together to form a fully functional muscle fiber [Grounds *et al.*, 2002]. It is important to note that satellite cells are not directly affected by a lack of dystrophin; however, since satellite cells attempt to divide indefinitely to repair a muscle fiber, when the fiber isn't repaired completely, the regenerative capacity of satellite cells is eventually exhausted as the dystrophy progresses [Boldrin *et al.*, 2015].

In DMD, muscles are unable to repair themselves due to a genetic mutation on the DMD gene that doesn’t allow for the production of functional dystrophin. This mutation is usually the deletion to part of the DMD gene, causing an early stop codon within the DNA [USNLM, 2015] Dystrophin is “part of a group of proteins that work together to strengthen muscle fibers and protect them from injury as the muscles contract and relax” [USNLM, 2015]. This protein connects each muscle cell’s structural framework with other proteins and molecules outside of the cell while also possibly playing a role in sending and receiving chemical signals in cell signaling. Dystrophin binds muscle membrane and helps maintain the structure of the muscle cells. Without it, muscles would not be able to operate properly and eventually would die after constantly suffering from progressive damage [USNLM, 2015].

Duchenne Muscular Dystrophy progresses at a rapid rate compared to other forms of muscular dystrophy. Symptoms such as enlarged calf muscles, which occur due to the calf muscle containing scar tissue, can start as soon as the child starts to walk, making the child appear clumsy since he is unable to keep from falling. Children may also begin to walk on their toes in order to maintain their balance. After their toddler years, children gradually lose the ability to walk and many are confined to a wheelchair by the age of 12. As the disease progresses, the muscles around the heart and respiratory organs weaken, causing damage that could be life threatening. DMD patients typically have a life span of 20 to 30 years [MDA, 2015b].

2.2 Stroke and ALS Patients

While the subject of this project was not a stroke or ALS patient, it was important to look at other populations that could benefit from the device being created. Not only did it add to the value of the intended device, but it also introduced alternative populations that could benefit from using this device. This section discusses stroke and ALS, their causes (if known), their clinical presentations, and how they compare with those of DMD patients.

2.2.1 Post-Stroke Patients

A stroke, which usually leads to the death of a portion of one's brain cells, can be caused by a variety of factors. Essentially, it is caused by an interruption of the necessary blood supply to the brain, which can be due to either a blockage in the artery that supplies blood to the brain [cdc.gov, 2013], more commonly known as an ischemic stroke, or a ruptured artery in the brain that causes a leakage in the brain and puts added pressure on its cells [cdc.gov, 2013], more commonly known as a hemorrhagic stroke. While there are several other causes of stroke, they can be considered derivatives of ischemic and hemorrhagic strokes. Regardless, a stroke can lead to damage in several parts of the brain, leading to impairments in the parts of the body that are influenced by those parts of the brain. For example, the frontal lobe is responsible for a person's "planned" movement, or a movement a person intends on making [Mcgrill, 2016]. Damage to the person's frontal lobe could lead to several complications in a person's "planned" movements. The complications could be anything from a slight delay in an intended movement, to an inability to perform that movement altogether, depending on the severity of the stroke.

In stroke patients, damage to the corticospinal tract can lead to upper extremity functional loss. This functional loss can be presented as paresis, loss of fractionated movement, abnormal muscle tone, and/or changes in somatosensation [Lang *et al.*, 2013]. Table 2 summarizes each of these functional losses, what they entail for the patient in terms of upper limb movement, and what parts of the brain or spinal cord are damaged (thus, resulting in these losses). Additional upper extremity impairments seen in post-stroke patients include reduced finger grip or extension, reduced elbow flexion or extension, and reduced shoulder abduction or adduction [Cascio *et al.*, 2014]. These impairments are similar to those afflicting DMD patients since they all limit the use of the patient's upper extremity.

Table 2: Common post-stroke upper extremity impairments

Symptom	Presentation in Patient	Associated Brain/Spinal Cord Damage
Paresis	-Muscles not activated in a timely manner -Appears as weakness and slower, less accurate movements	Primary motor cortex, non-primary cortical motor areas, corticospinal tract
Loss of fractionated movement	-Loss of ability to voluntarily move one portion independently of one another -Cannot "selectively" activate muscles	Corticospinal tract
Abnormal muscle tone	-Hypotonicity (reduced muscle tone) -Hypertonicity (increased muscle tone)	Basal ganglia
Changes in somatosensation	-Nervous system's ability to monitor and correct movement is lessened	Ascending somatosensory pathways, somatosensory cortical areas

Upper extremity impairments in post-stroke patients can be similar to the impairments faced by DMD patients as the disease progresses. For example, just like post-stroke patients, DMD patients can suffer from paresis as well [Cascio *et al.*, 2014]. In addition, DMD patients can suffer from pseudohypertrophy [MedlinePlus, 2016], which is analogous to a post-stroke patient's abnormal muscle tone in terms of the physical size of the affected limb changing. These similarities make the device being created useful for not just DMD patients, but post-stroke patients as well.

2.2.2 ALS Patients

Clinical presentations in ALS patients are similar to those seen in DMD and post-stroke patients. Like DMD, patients of ALS usually die due to respiratory failure. ALS is characterized by the degeneration of motor neurons that are found in voluntary muscles, and usually begins in the limbs (more specifically, the arms) [Gordon, 2013]. Some "first signs" include constantly dropping things, shortness of breath, and tripping on carpets [Scott *et al.*, 2007]. What sets ALS apart from stroke and DMD is that the exact cause of ALS is unknown. That is, while DMD is accepted to be caused by genetic mutation, only 10-15% of ALS cases are considered genetic. However, like DMD and post-stroke patients, ALS patients usually exhibit a loss of hand dexterity and demonstrate weakness when lifting their arms. In fact, doctors estimate a 50% loss of motor neurons in ALS patients by the time a diagnosis is performed [Scott *et al.*, 2007].

2.3 Scales and Measures for Strength and Function Evaluation

There are a wide range of tests that can be performed on patients to record the progression of degenerative diseases or measure recovery from muscle damage. Many of these tests require little to no equipment and are good for rough approximations. For the focus of this project, the team looked at tests that measured upper limb function and strength.

Table 3: A summary of different upper limb assessments for clinical populations

Test	Scale	Areas of Measurement	Materials	Clinical Populations
Manual Muscle Testing (MMT)	0 – 5 point scale based on clinician	Shoulder, Elbow, Finger	None	DMD Stroke
Hand – Held Dynamometers (HHD)	Newtons, kilograms, pounds	Shoulder, Elbow, Finger	Strain Gauge	DMD
Fugl-Meyer Assessment (FMA)	0 – 3 point scale based on clinician	Whole arm from shoulder to fingers	Plexor	Stroke
Action Research Arm Test (ARAT)	0 – 3 point scale based on clinician	Grasp, grip, pinch and gross movement	Balls, Blocks	Stroke
Nine – Hole Peg Tests	Time scale	Hand Dexterity	Pegs	Stroke
Accelerometers	Voltage	Arm	Piezoelectric resistors and software	Stroke

Table 3 shows several techniques used to measure upper limb strength, the most popular and oldest one being the Manual Muscle Test (MMT). It is widely used to gather qualitative data on muscle strength and functional abilities [Scott *et al.*, 2007], making it a relevant test for DMD patients. To conduct the test, a physician simply applies a small force on a part of the patient's limb and asks the patient to apply a force against the physician's palm that would resist that force [Scott *et al.,* 2007]. On a five-point scale, a physician determines a value between zero and five that indicates muscle strength, zero indicating no observable contraction and five representing “normal” muscle strength. Another possible test that may be used on DMD patients is the Hand-Held Dynamometer (HDD) which records quantitative data in Newtons, kilograms or pounds. This device works similarly to MMT, but the force pushed back by the patient can be measured by the device, rather than estimated by a physician. To conduct the test, a physician simply applies a small force on the muscles and asks the patient to apply a force against their palm [Scott *et al.*, 2007]. An HHD can allow for the testing of grip strength, giving physicians important information on various muscle groups working to produce that gripping motion.

Tests for stroke patients are characterized as performance tests to assess the improvement of motor function post therapy. The Action Research Arm Test (ARAT) is a test that measures four parts of the arm: grasp, grip, pinch and gross-movement. This is done in 19 individual performance tests. The ARAT is conducted on a four-point scale, in which three is "normal" performance and zero indicates no completed tasks. To conduct the test, a few materials are required, but training for the test is not required. Blocks are used to measure grasp, marbles to measure pinch, tubes to measure grip, and movement of one's hand in everyday activities to measure gross movement [Rehab Measure, 2015a]. For an unbiased test, the Nine–Hole Peg test is used to measure hand dexterity on a time scale. Patients recovering from stroke are told to move pegs from one place to another. A faster time means a better performance and greater motor recovery. The Fugl-Meyer test focuses on motor function, sensory function, balance, joint range of motion and joint pain. The test does require some material and the test is administered by a technical guide and is a popular test to be administered for post-stroke patients [Rehab Measures, 2015b].

A tool for testing is still under clinical investigation is the use of accelerometers. In this test, the whole arm is measured with a device that is similar to a wristwatch but has an accelerometer that can measure how much movement occurred in a specific time period [Lang *et al.,* 2013]. The sensor within the wristwatch-like device is a piezoelectric sensor that converts mechanical acceleration into electrical signals.

2.4 Musical Instruments

It is vital to understand the basics of music, as well as how some of the instruments that were considered for the project work. This information gave the team an idea as to what goes into creating a musical device. The following section explores the fundamentals of music and how some of the subject's favorite instruments work. It also explains how each of these instruments is played, which gave the team a foundation for creating the assistive device based on that instrument.

2.4.1 Fundamentals of Music

The intensity (loudness) of music is determined by the amplitude of the sound wave; that is, higher intensity indicates a larger amplitude. The frequency of sound waves determines the pitch of music and the higher the pitch, the higher the frequency. The universal way of assigning pitches is called twelve-tone equal temperament (12-ET). The 12-ET system divides the octave into 12 intervals, whose frequency spans are equal on a log scale. An interval can also be called a half-tone. The standard pitch in 12-ET is A440 with a frequency of 440 Hz, which means all other pitches are tuned based on that frequency. An octave of the 12-ET [howstuffworks.com, 2015] is as follows in Table 4.

Table 4. The corresponding frequencies for as each pitch increases with each note.

Note	**Frequency [Hz]**
C (Middle C)	261.6
C#	277.2
D	293.6
D#	311.1
E	329.6
F	349.2
F#	349.2
G	392.0
G#	415.3
A	440.0
A#	466.1
B	493.8
C	523.2

The letters in the list above are notes assigned to each pitch. Notes are commonly used in written music ranging from A to G. A scale is a sequence of notes ordered by pitch. There are many types of scales. The most common scales among them are the major scale, the natural minor scale, the harmonic minor scale and the melodic minor scale. A chord is a collection of nodes that create a harmonic sound together. The most common types of a chord are the major triad chord and the minor triad chord. A triad chord is a group of three notes where the triad is built upon the root note. To form a major triad, a major third and a perfect fifth should be added on top of the root. Similarly, a minor triad consists of a minor third and a perfect fifth added on top of the root. For a diminished triad a minor third and a diminished fifth would be added to the root. A major third is the note that is four half-tones above the root note and a minor third is three half-tones above the root. Furthermore, a perfect fifth is 7 half-tones above the root and a diminished fifth is 6 half-tones above the root [Kessler, 1945]. Considering the pattern between chords, an algorithm can be created to change chords depending on the key a user selects to play in.

2.4.2 How Instruments Work

The following section explains what each part of the guitar or ukulele and piano does. The guitar and ukulele parts described are the head, neck, strings, and body. The piano parts described are the keys.

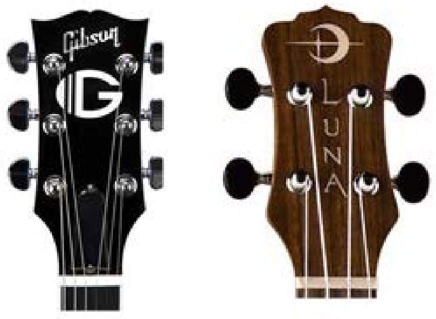


Figure 6: Guitar Head (left) and Ukulele Head (right)
[Capacicar, Lewington, 2016]

The head of a guitar or ukulele, depicted in Figure 6, contains tuning pegs, which adjust the pitch of each string by changing the tension of the string.

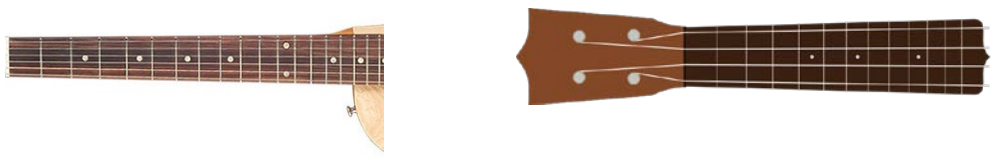


Figure 7: Guitar Neck (left) and Ukulele Neck (right)
[Ukulele Helper, 2015]

The neck of a guitar or ukulele, depicted in Figure 7, holds the frets. Frets are pieces of wood where players can push their fingers against to trap the string between the fingers and the frets. Frets can change the pitch by changing the vibrating length of the string. There are circular marks on the neck that help players to find chord positions.

On the head, neck, and body of a guitar or ukulele are strings, which are vibrating elements that produce sound. There are three factors that can affect the pitch (frequency) of the sound. They are: length of the string, weight of the string and the tension of the string. The heavier the string, the lower the pitch. In the same way, the shorter the string, the higher the pitch and the tighter the string, the higher the pitch.

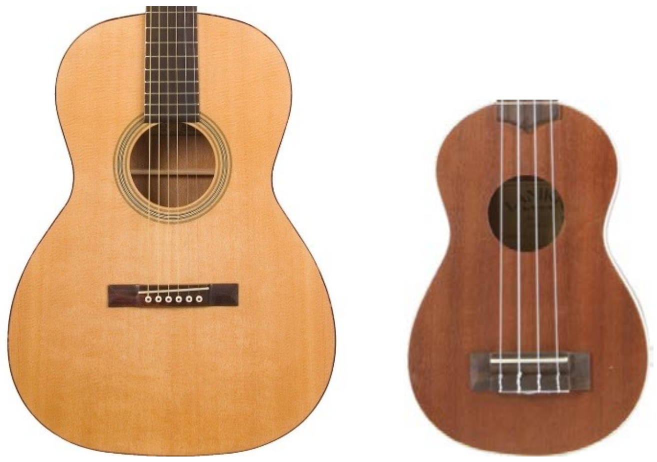


Figure 8: Guitar Body (left) and Ukulele Body (right)
[ZoZo music, 2012] [Gear Nuts, 2016]

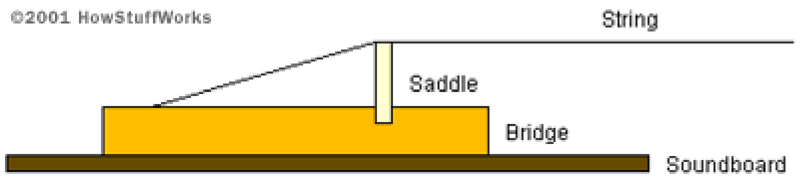


Figure 9: String, Saddle, Bridge and Soundboard
[HowStuffWorks, 2001]

The body of a guitar and the ukulele, depicted in Figure 8, contains several parts that are essential in making sound. On the top surface of the body is the soundboard, shown in Figure 9. The saddle and the bridge are two parts that connect the strings and the soundboard. When the string vibrates, the soundboard follows. The body then amplifies the vibration of the soundboard to make the sound louder. Finally the sound waves are projected through the sound hole to the eardrums.

The way an electric guitar or ukulele works is very similar to the acoustic ones. The head, neck and the string parts are the same. The only difference is the amplifying mechanism. Unlike amplifying the sound using a purely acoustic sound box, electric guitars or ukuleles use a magnetic pickup to convert the vibrations of the strings into vibrating current. The pickup is a bar magnet wrapped with wires. When the strings vibrate, the changes in the magnetic field produce a corresponding vibrating current that goes through the wires to a circuit that allows the player to change the tone and volume. By plugging in an amplifier, the current will be processed in such a way that it will be audible.

Musicians can play the guitar or ukulele by resting it on their lap and holding it against their chest. One hand holds the neck and presses the strings against the frets to create different chords or notes. The other hand rests on the body of the instrument and strums to produce sound. To play the guitar or ukulele, players need to have enough arm strength to be able to hold the instrument and enough wrist flexibility to slide along the neck or strum.

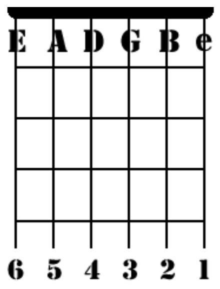


Figure 10: Guitar Tuning
[Matchim, 2016]

The guitar is tuned as shown above in Figure 10. From left to right, the notes are: E3, A3, D3, G3, B3, and E4. The ukulele is tuned as shown below in Figure 11. From left to right, the notes are: G4, C4, E4, and A4

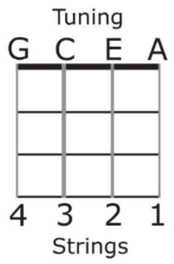


Figure 11: Ukulele Tuning
[Ukulele Lessons Today, 2016]

Another stringed instrument is the piano, but it works in an entirely different way than a guitar or a ukulele. Unlike playing strings on a guitar or ukulele, there are 88 keys, a portion of which is depicted in Figure 12, on a grand piano that one must press to play an individual note. A key is a wooden lever in which one end is much longer than the other. When the longer end is pressed, the smaller end lifts up and a hammer attached to a key hits a string inside a piano. At the same time, a damper, which controls the stopping of a string's vibration, is also raised up to release the string so it can produce sound. When the key is released, the damper falls back on to the string to stop the vibration.

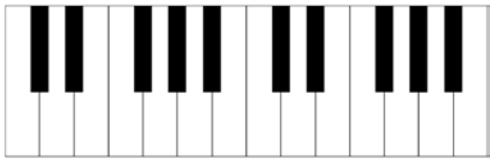
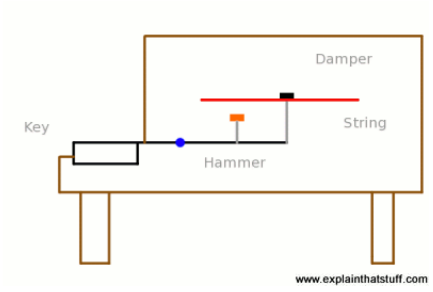


Figure 12: Piano Keyboard

[Clipart Panda, 2014]



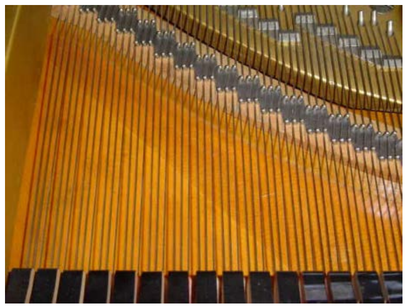


Figure 13: Inside the Piano and Piano Strings
[Woodford, 2008] [PianoNoise, 2016]

The strings in a piano, shown above in Figure 13, serve the same purpose as the strings in a guitar or ukulele; the strings vibrate to make sound waves. Notes on the left side of the piano (facing the back) have lower pitches, which make the strings longer than the ones on the right side of the piano. Thus, the grand piano is shaped in such a way to accommodate the longer and shorter strings.

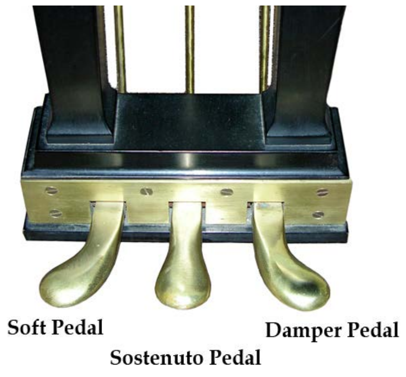


Figure 14: Piano Pedals
[Infinity Music Studio, 2012]

There are three different pedals on a grand piano, shown above in Figure 14. From left to right, there is the soft pedal, the sostenuto pedal, and the damper pedal. The soft pedal makes the sound softer by shifting the hammers to the right so they press against one string instead of three strings, which is how many strings they normally hit. The sostenuto pedal holds the deactivated dampers high, turning them on. The sostenuto pedal makes the notes that are played last longer.

The damper pedal raises all the dampers away from the string when pressed and is considered to be "the soul of the piano" because it allows for greater expressivity.

To play the piano, musicians need to sit in front of the instrument and place both hands on the keyboard. It requires certain arm strength to allow both hands to move on the keyboard and some finger strength to be able to hit the keys.

2.5 Musical Assistive Aids Prior Art

While there was a lot of research into assistive aids for those with DMD and stroke, there was limited information on any musical assistive aids for people with these diseases. However, this does not mean that there wasn't any research on the subject. There were a few musical devices that were created intentionally to be used by those with muscle deficits. Other devices were created to be used by people with typical muscles, but they may be able to be applied to people with muscular dystrophy or who experienced a stroke. These devices could help those who lost the ability to play music regain their passion for the art.

2.5.1 Existing Devices

Some musical assistive devices use computer software that links to a unique device to play music. One example that was created in association with the Muscular Dystrophy Association (MDA), called the Laser Band, shown in Figure 15, uses a device which contains four low-intensity lasers to play music [quest.mda.org, 2012]. If a user blocks a laser, then a specific sound will play according to which instrument was preprogrammed to the device. This device is mostly used to collaborate with professionally created music, but it can also be used to create any music one has in mind. It was designed with disabled people in mind, offering multiple means to operate the device if user mobility is limited.

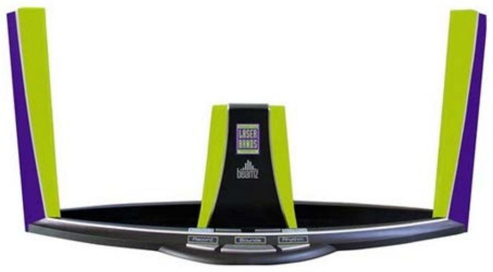


Figure 15: The Laser Band
[Quest, 2016]

The MDA was also involved with a similar device called the Jamboxx, presented in Figure 16. This device is a hands-free electrical harmonica-like device that also interacts with software to make and create music. The user must blow and slide the mouthpiece on the device to play the sounds that they want. Both devices support multiple digital instruments, making it easier to create the music one wants to make.

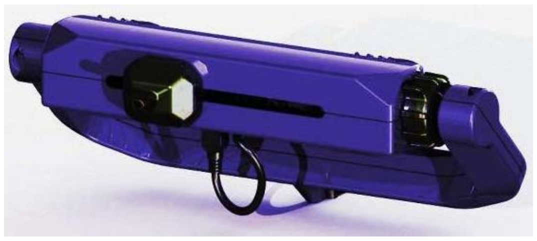


Figure 16: The Jamboxx
[Quest, 2016]

A musical device that was created to easily create music digitally, and not necessarily for disabled patients, consisted of interactive gloves created in collaboration with the singer and composer Imogen Heap. These gloves, demonstrated in Figure 17, were created to make it easier for her to produce music using Digital Audio Workstations (DAWs). The device has a lot of functionality depending on how someone moves their hands. For example, bending one finger can produce a different audio effect or sound than bending a different finger or even moving the entire hand up. She has created many iterations of the device with help from engineers and saw the device as something that will eventually change the standard for creating music in the future. While this device is not specifically for DMD or stroke patients, the device has the potential to be reimagined or reinvented for these patients [Pallister, 2014].

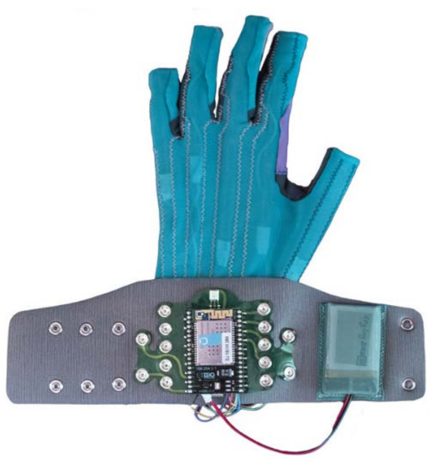


Figure 17: Imogen Heap's Electronic Gloves
[Pallister, 2014]

Other musical assistive devices are a combination of electrical and mechanical inventions. These devices are useful for a person with disabilities to play a specific instrument. Currently, there is a patent for a modular automated assistive guitar, shown in Figure 18, which places an acoustic guitar in a base station. The device uses mechanical strutting and fretting mechanisms that both use electrical controllers to actuate. This device makes it possible to move down and up the frets on the guitar with a knob-like device. The user then taps a pressure switch to strum the guitar. With this device, it is easier for patients with disabilities to play the guitar. It can also be adjusted to fit the needs of the user so that it is easier for them to use the device [White *et al.*, 2007].

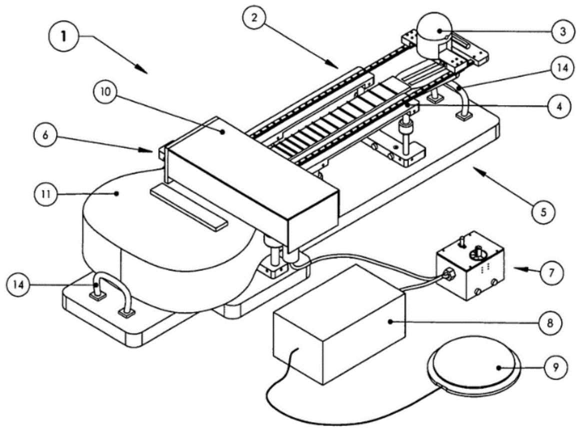


Figure 18: Patent Layout of the Modular Automated Assistive Guitar
[Schauer et al., 2003]

A different electro-mechanical device can be used to play the drums, depicted in Figure 19. It uses an electromagnet and a spring to propel a drumstick downwards and then back up. Using a few of these devices on one drum set can let a disabled person play the drums using an array of buttons or switches [Coton *et al.*, 2014].

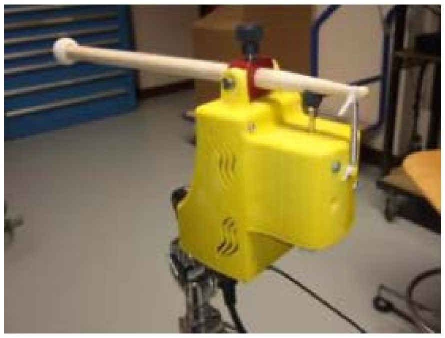
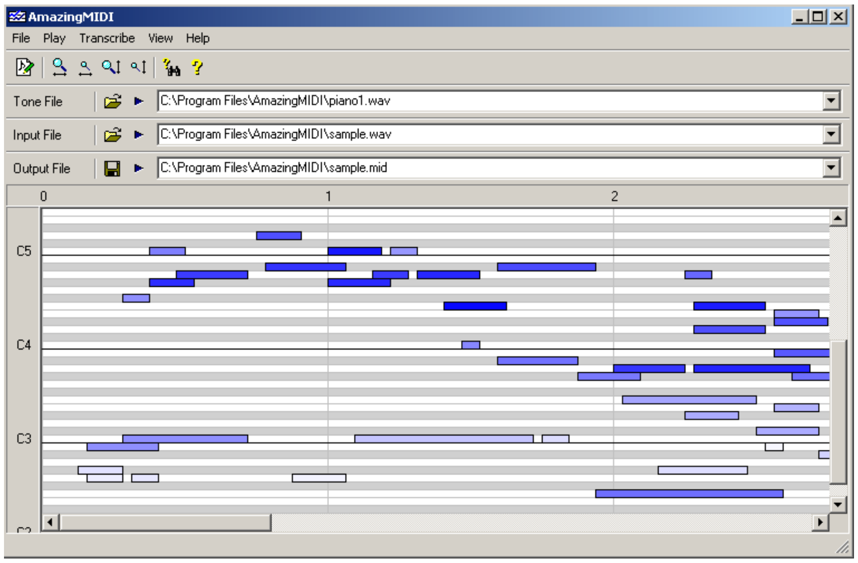


Figure 19: Electro-Mechanical Drum Stick
[Coton et al., 2014]

Some musical assistive devices were not made to play music, but rather to help patients with their therapy. One such device was used in a study to see if Musical Motor Feedback (MMF) could be used to improve the walking capabilities of stroke patients. The MMF device consisted of sensors embedded in the soles of shoes as well as a portable music player compatible with the MIDI (musical instrument digital interface) standard. The patients listened to music with headphones while they walked at their normal pace, and the music speed would be adjusted depending on the time between two heel strikes. Data were collected via Matlab to see if the MMF device was helping patients to walk better and faster. It was concluded in the study of this device that MMF was successful in increasing the walking pace of the patients [Schauer *et al.*, 2003].

2.5.2 Digital Music

Briefly discussed in the previous section were Musical Instrument Digital Interface (MIDI) and Digital Audio Workstations (DAW), which are powerful digital musical technologies that are used to easily create music with a computer. MIDI is a music technology that has been around for more than 30 years [midi.org, 2015]. An example of a GUI manipulating a MIDI file is shown in Figure 20. It can be analogous to a computer language that describes the process of how a computer plays music. Each piece of MIDI code, or MIDI message, describes what notes to be played, how long the note should be played, the tempo, the pitch, which instruments should be played, and the relative volume of each note or instrument. With MIDI, it is possible to fix, change, speed up, or slow down any kind of music, which makes it a great tool to create, play, or learn music. However, MIDI cannot play music on its own, and must have some sort of computer or controller to play a MIDI sequence. Many different devices such as cell phones, keyboards, and computers use MIDI to output music [MIDI Association, 2015]. An example of how a device is connected to musical instruments through MIDI software can be seen in Figure 21.



Tone File: C:\Program Files\AmazingMIDI\piano1.wav

Input File: C:\Program Files\AmazingMIDI\sample.wav

Output File: C:\Program Files\AmazingMIDI\sample.mid

Figure 20: Example of a MIDI File using AmazingMIDI Software
[MIDI Association, 2015]

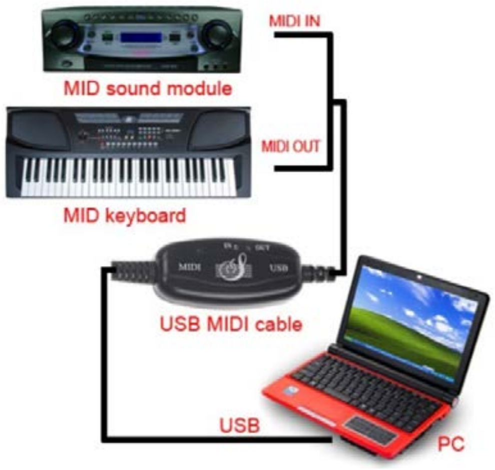


Figure 21: Connecting Musical Instruments through MIDI Software
[Sienoc, 2016]

A DAW, of which one example of a DAW interface is shown in Figure 22, is a computer software application which can be used to record or edit audio files. Within a DAW, a user could play digital instruments and mix them together to make a song or a track [sweetwater.com, 2015]. The user could even record his or her voice and add it to their song. Most DAW programs come with plug-ins such as distortion, synthesizers, compressors, and many others to alter the audio recordings they have made. Besides editing and adding effects to raw audio, a DAW can also be used to edit and change MIDI files. The MIDI file can be altered to change which instrument is being outputted. It can also be edited in the same way raw audio is edited with distortion, synthesizers, and other effects. Thus, MIDI and DAWs can go hand and hand, expanding what one can do with digital musical.

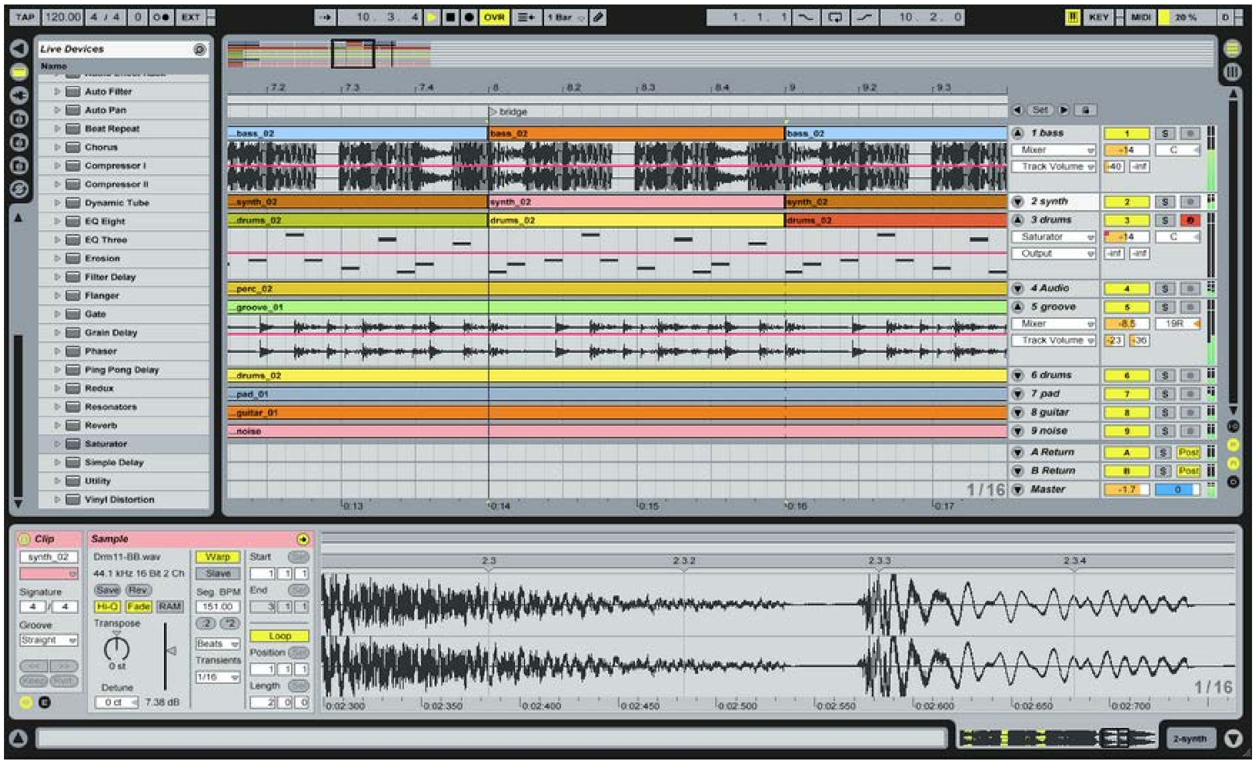


Figure 22: Example of a DAW, Ableton Live
[Sweetwater, 2015]

2.6 User Interface Options

The team recognized a need for a way for the user to interface with the device specifically for the hand and fingers to be used in a manner to indicate the desired fingering of a ukulele. To accommodate to this method, three options were considered: force-sensitive resistors, capacitive touch screens, and buttons.

2.6.1 Force-Sensitive Resistors

A force-sensitive resistor (FSR) is a device that has a resistance that fluctuates depending on the amount of pressure (or force per unit area) that is applied to it. It acts as a variable resistor in a circuit. The resistance of the FSR decreases as the pressure or force applied increases. An FSR typically contains four distinct layers (as seen in Figure 23) consisting of electrically insulating plastic; an active area consisting of conductors, connected to the leads on the tail; a plastic spacer, including an opening that is aligned with the active area and an air vent that runs through the tail; a flexible substrate coated with a thick polymer conductive film aligned with the active area- this polymer can be replaced by a layer of FSR ink [Interlink Electronics Inc., 2015].

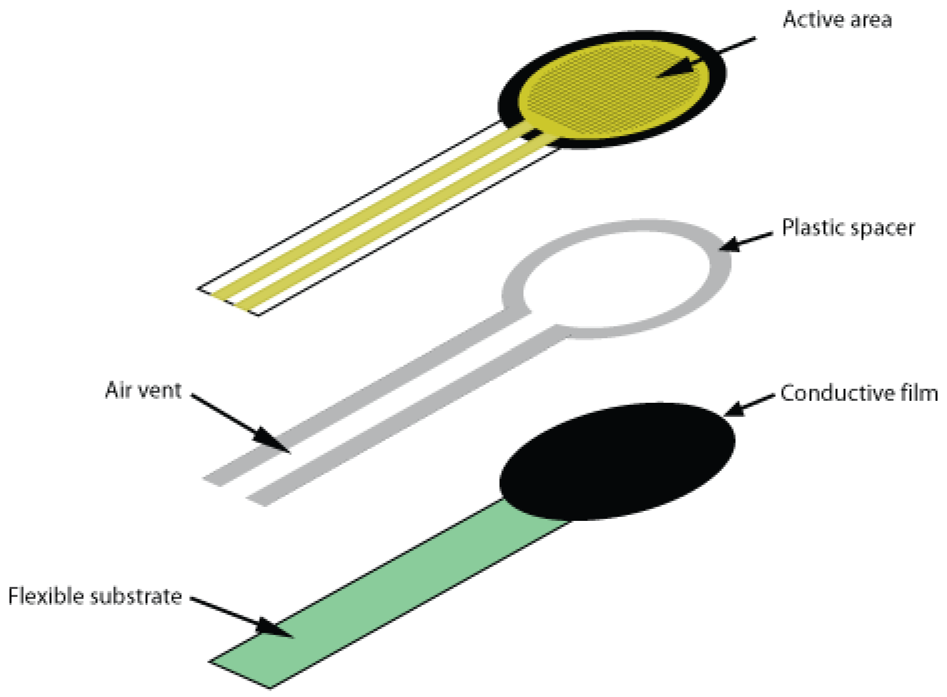


Figure 23: Layers of an FSR
[DigitalCommons, 2015]

When mounting an FSR, it is recommended that any curved surfaces be properly accounted for since bending the tail deforms the air vent and may apply a small force to the FSR. The tail is also fragile and the leads could easily break.

An FSR is commonly implemented in an electric circuit as a voltage divider configuration. That is, it is placed in series with another resistor and when pressure is placed on the FSR, a fraction of the input voltage is received at the output. FSRs can also be used to interact with a digital interface, as well as in a Schmitt Trigger Oscillator [Interlink Electronics Inc., 2015].

2.6.2 Capacitive Sensing

Capacitance is the ability of a body, such as a capacitor, to store an electrical charge. A parallel plate capacitor is a common form of a capacitor, of which the capacitance is calculated using the equation $C = Q / V$. The variable C here stands for the capacitance, Q is the charge stored, and V is the voltage across the capacitor. The capacitance of a parallel plate capacitor is determined by the area of the two plates A, the dielectric constant of the material between the plates ε_r, the electric constant ε_0, and the distance between the plates d. The equation to calculate the capacitance is: $C = \varepsilon_r * \varepsilon_0 * A/d$.

A basic capacitive sensor is anything that is conductive that detects anything else that is conductive, such as the human body, liquid, or a material that has a dielectric constant that differs from air. For a human-to-machine interface, a simple implementation of a metal powered by a low voltage can easily serve proximity detection purposes. In this case, the human body plays the role of ground compared to the powered metal. The electric field between the human body and the metal changes when the distance between the two changes. This phenomenon, of which an example can be seen in Figure 24, is how gesture change can be detected [Bhowmik, 2014].

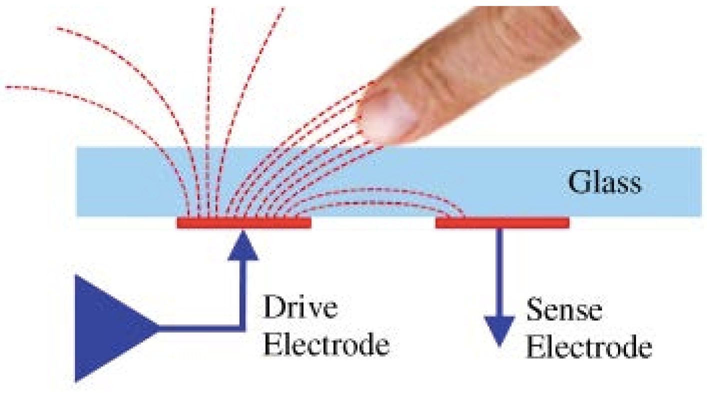


Figure 24: Capacitive Sensing Diagram
[Bhowmik, 2014]

Capacitive touch requires less accuracy than other capacitive sensing applications, such as liquid level sensing and material analysis. Capacitive touch usually requires a sensitivity of 10s to 100s fF, while other applications require a sensitivity smaller than 1 fF. There are two main types of capacitive touch technologies: surface capacitive touch and projected capacitive touch.

Surface capacitive touch, which can be seen in Figure 25, is commonly used for large size screens like an ATM. It contains a layer of thin wires on the four corners of the screen with another layer of insulators on top to protect humans. The corners are given a low voltage to create a uniform electric field. When a touch is performed, there will be current drawn from each corner. Then, a microcontroller can be used to measure the ratio of the current flow from each corner and then the location of the touch can be determined. Unfortunately measuring current flow from each corner does not support applications that require multi-touch interfaces [Bhowmik, 2014].

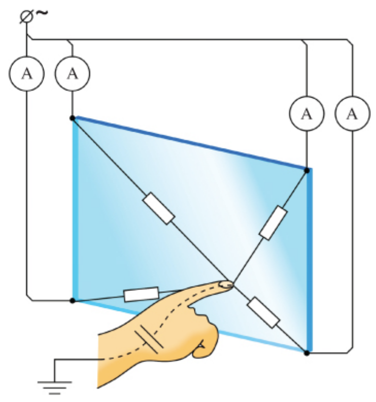


Figure 25: Surface Capacitive Sensing
[Bhowmik, 2014]

Projected capacitive touch technology, seen in Figure 26, supports multiple touch inputs at the same time, which is why it's largely being used for mobile technology like the iPhone and iPad. Projected capacitive touch screens use an X-Y grid of conductive material in between two insulators. During a touch, the electric field of the location of the touch changes due to the capacitance that forms between the finger and the grid. Then, the controller calculates the location of the touch on the X-Y grid based on the changing electrical characteristics caused by the touch.

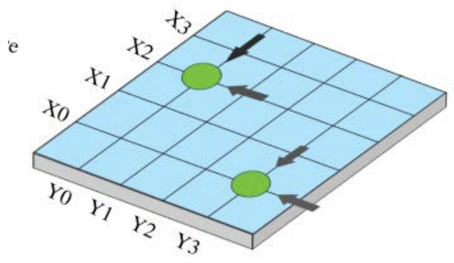


Figure 26: Projected Capacitive Sensing
[Bhowmik, 2014]

2.6.3 Buttons

One of the simplest options for implementing a controller sensor system is using buttons. Buttons operate by closing or opening an electrical connection when the mechanism of the button is depressed by a certain amount. The distance which the mechanism needs to move to register a button press varies significantly between different types of buttons and can be completely customized to suit the application. Some buttons are of the momentary-contact type which means the button will return to its original state when the force is released, others are persistent which means the button stays in the pressed state until the next press is released and returns it to the un-pressed state. Input from a momentary-contact switch can be interpreted into a persistent-type signal using a circuit or piece of software designed for the purpose. In terms of signal processing, a button input is handled as a digital signal. In either the pressed or un-pressed state the button is equivalent to either an open circuit or a short circuit. The voltage of the signal in the “high” and “low” states can be any values which meet the specifications of the device accepting the signal and also which do not exceed the maximum specifications of the button. The flexibility afforded by a button can be a considerable advantage to the ease of design for a device.

Buttons are best suited for applications where the amount of force applied by the user is not important or useful information for the application. The reliability and repeatability of buttons is also well suited for high volume and heavy duty uses, for example use as elevator call inputs.

2.7 Controller Circuitry

The team identified several options for the main input sensors of the controller. FSRs were favored (as described in section 2.1), so that component was selected to move forward with the design. To keep the device and design process efficient, the simplest possible circuit was chosen and built up in stages (as needed) to improve the performance of the sensor subsystem. Each design has its own advantages and disadvantages, those of which will be explained in the following section. In the following circuit designs, the FSR is usually denoted as the variable resistor R_S. Other components will be labeled accordingly and will be described through equations.

2.7.1 Pull-Up Resistor

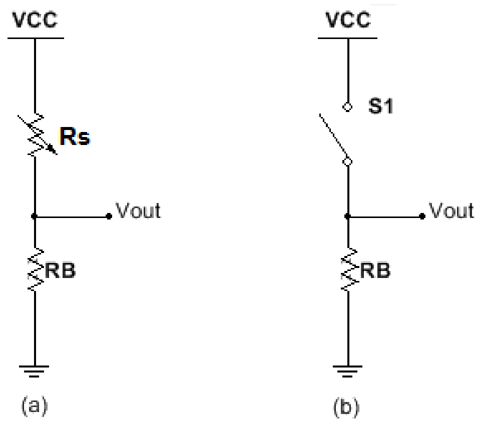


Figure 27: Pull Up Resistor Configurations

In the simplest case, the FSR could be used to be a pull up resistor. The resistance of the FSR ranges from infinity (zero force condition) to a known quantity of about 20-200 kΩ (high force condition). With no force, the FSR simply acts as an open switch because the resistance is so high. When there is no load, the output voltage V_{out} will simply be pulled to zero volts. When a force is applied to the FSR, a resistance R_{FSR} will be between 20 to 200kΩ. Figure 27 (a) and (b) shows this voltage divider configuration in which the output voltage will be:

$$V_{out} = \frac{R_B}{R_B + R_S} V_{CC}$$

The output voltage can be modified by adjusting the resistance $R_{B,}$ when V_{CC} is constant. The V_{CC} can be the voltage used to power the microcontroller and V_{out} can be determined to be the output voltage when the logic is high, as specified by the microcontroller. This configuration has the least amount of components and can be directly connected to the microcontroller. It can also be easily adjusted to fit the threshold of the microcontroller if the only variable to change is the value of R_B.

2.7.2 Inverting Amplifier

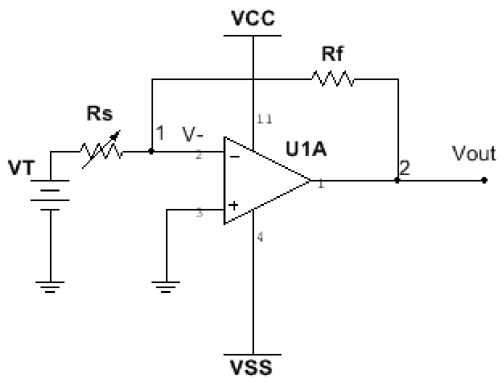


Figure 28: Simple Inverting Amplifier Circuit

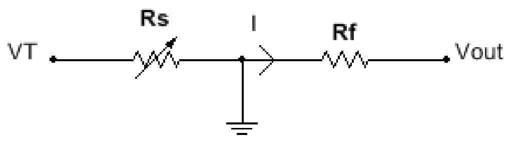


Figure 29: Model for Inverting Amplifier Circuit

The inverting amplifier circuit configuration, shown in Figure 28, is one of the simplest ways to amplify the output from a FSR once the 3.3V supply voltage V_T, has passed through it. The model in Figure 29 can be used to determine what the gain of this circuit should be. To find the gain, or the ratio of the output voltage to the input voltage, the equation for V_{out} must be derived first. Using Kirchhoff's Current Law (KCL) at node 1, the equation would be:

$$\frac{0V - V_T}{R_S} = \frac{V_{out} - 0V}{R_F}$$

This equation provides a way to solve for V_{out}:

$$V_{out} = -(\frac{V_T}{R_S})(R_F)$$

It can then be said that the gain of the circuit is:

$$Gain = G = \frac{V_{out}}{V_T} = -\frac{R_F}{R_S}$$

Since the gain can be changed by changing the resistor R_F, this circuit is easy to implement into the design. Of course, the gain would also change when the FSR changes resistance (i.e., when pressed), but this can be accounted for by measuring the FSR resistance when both pressed and not pressed using a digital multimeter. Through these resistance measurements and the gain equation above, a value for R_F can be determined in relation to the measured resistance of R_S to implement the desired gain.

Since there are fewer parts, this circuit is cheaper and easier to implement compared to the other circuits that were analyzed. However, although cost efficient and easy to create, it may not be the best design choice for our device. For example, this simple circuit may not account for the sensitivity of the FSR, which will most likely be a problem when trying to detect if the FSR has been pressed. Also, this circuit's output may be slightly harder to read into a microcontroller as it is a continuous analog signal. Therefore, the microcontroller that will receive the output signal would need an analog to digital converter so that a threshold voltage could be set in software.

2.7.3 Comparator Switch

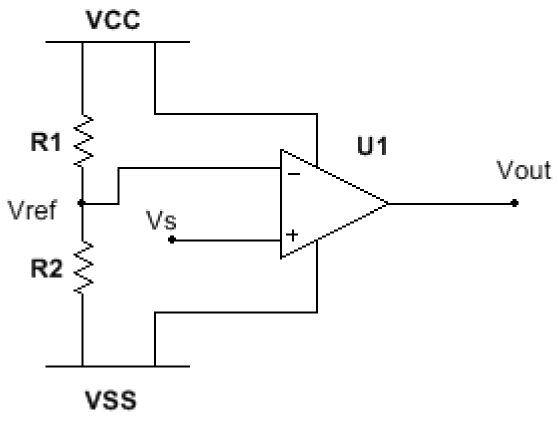


Figure 30: Simple Comparator Switch Circuit

The simple comparator circuit shown in Figure 30 is another simple circuit that may be used to detect whether the FSR is being pressed. This circuit can be described with the following statements:

- If resistor R_1 equals resistor $R_{2,}$ then the reference voltage V_{REF}, which is the voltage going into the negative side of the operational amplifier U_1, will be equal to half the input voltage $\frac{Vcc-Vss}{2}$. This relationship is shown in the following voltage divider equation:

$$V_{REF} = (V_{CC} - V_{SS})\left(\frac{R_2}{R_1 + R_2}\right) = V_{CC}\left(\frac{R}{2R}\right) = \frac{V_{CC}}{2}$$

- If the sensor voltage V_S is greater than V_{REF}, then the output voltage V_{OUT} will be equal to V_{CC}.
- If V_S is less than V_{REF} then V_{OUT} becomes V_{SS}.

With these statements, it can be seen that the comparator is useful for this application. A voltage divider can be set up so that when the FSR is pressed, the sensor voltage V_S will be greater than the reference voltage V_{REF} and output V_{CC}. When the FSR is not pressed, V_{OUT} should be V_{SS}. As this circuit is easy to implement and can be created using a comparator IC and a resistor, this circuit seems like a reasonable design to use for the controller circuit. However, this circuit may still not be the best choice as it does not account for the sensitivity of the FSR. If V_S is too small, then it may be too hard to design the circuit in such a way that V_{REF} would be smaller than V_S. This circuit may be most useful if the FSR output can first be amplified prior to the next stage.

2.7.4 Wheatstone Bridge

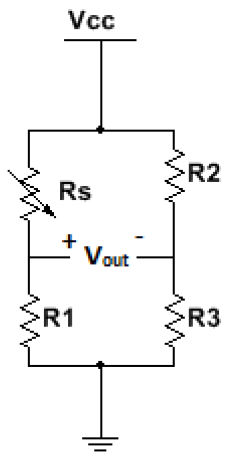


Figure 31: Wheatstone Bridge

The Wheatstone bridge is used as a means to measure the resistance of an unknown resistive element. From Figure 31, the Wheatstone bridge has two voltage dividers such that R_1 and R_S create one voltage divider and R_2 and R_3 create another. Both voltage dividers are connected by nodes at V_{cc} and ground. In order to measure $V_{out,}$ each output voltage of the voltage divider will be solved for separately.

$$V_{out+} = V_{CC}\left(\frac{R_1}{R_s + R_1}\right)$$

$$V_{out-} = V_{CC}\left(\frac{R_3}{R_2 + R_3}\right)$$

Since V_{out} is the difference between V_{out-} and V_{out+}, then V_{out} can be found by using the equation below:

$$V_{out} = V_{out+} - V_{out-}$$

$$V_{out} = V_{cc}\left(\frac{R_1}{R_S + R_1} - \frac{R_2}{R_2 + R_3}\right)$$

In order to have a balanced bridge, the output voltage must be set to zero when the variable resistor, R_S, is at its resting value. Since the voltage between the bridges is dependent upon the resistor ratio between the two voltage dividers, both sides of the bridge have to be matched in order to have zero at the output. V_{cc} and the resistance values can be chosen based on the equation for V_{out}, depending on the desired value for V_{out}.

$$\frac{R_1}{R_S + R_1} = \frac{R_2}{R_2 + R_3}$$

Solving for the variable resistor $R_{S,}$ the value of the variable resistor at rest is:

$$R_S = \frac{R_1 R_3}{R_2}$$

These derivations only remain true when the resistance between V_+ and V_- is infinite.

In this particular case, R_S is the FSR. It is known that the FSR ranges from infinity to 200kΩ when a pressure is applied. It may seem difficult to match resistances of infinity but adding a resistor in parallel with R_s will give a known value when the FSR resistance is infinity because the FSR will act as an open switch when no pressure is applied. As a result, when no load is applied, the resistance in parallel will be the ratio of $\frac{R_1R_3}{R_2}$. The resistance R_s can then be modified to be:

$$R_S = \frac{R_{FSR} \times \left(\frac{R_1R_3}{R_2}\right)}{R_{FSR} + \frac{R_1R_3}{R_2}}$$

The R_S value will then decrease when the resistance of the FSR, R_{FSR}, also decreases.

The output voltage will only be affected by the first half of the bridge where the ratio $\frac{R_1}{R_S+R_1}$ will increase so that the voltage at V_+ will increase. Since the voltage at V_ is not affected by the change in the R_S value, the voltage at V_- remains the same. From the V_{out} equation, if V_+ increases as R_S decreases and V_ remains the same, then V_{out} will also increase.

This design is geared more for the purpose of measuring small changes in resistance. This set up may also require more components that need to be carefully matched and an analog-to-digital converter to set a threshold. Despite requiring more parts, the overall design would still be relatively inexpensive.

2.7.5 Differential Amplifier

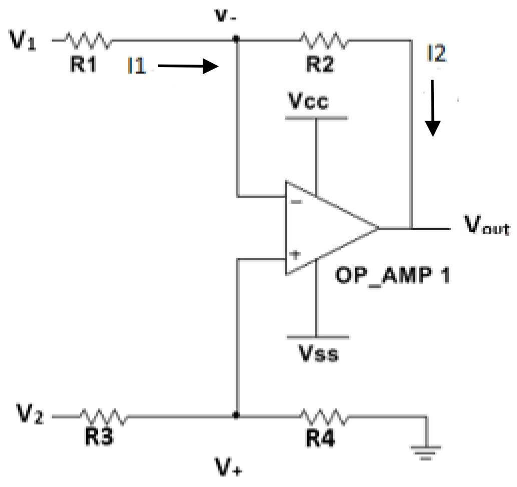


Figure 32: Differential Voltage Divider

The differential amplifier, as seen in Figure 32, is very useful in measuring the difference between two signals, since it subtracts their voltages. This op-amp stage could be useful, especially if it could measure the differential voltage of the Wheatstone bridge while also amplifying that voltage. In order to solve for the output voltage of the differential amplifier it is assumed that no current flows into the op-amp. The current across R_1 at V_1 would be:

$$I_1 = \frac{V_1 - V_-}{R_1}$$

Then the current across the feedback resistor R_2 would be:

$$I_2 = \frac{V_- - V_{out}}{R_2}$$

The current across resistor R_1 and R_2 at the negative terminal should be the same since no current goes into the op-amp. Setting both currents equal to each other, then V_{out} can be solved to be:

$$V_{out} = V_-\left(1 + \frac{R_2}{R_1}\right) - \frac{R_2}{R_1}V_1$$

The voltage at the positive terminal is found by a simple voltage divider equation:

$$V_+ = \frac{R_4}{R_3 + R_4}V_2$$

Using the law of superposition, V_{out} can be solved when V_1 is equal to zero and a set voltage V_2 is used. It is also assumed that the voltages at the positive and negative terminals are equal to each other so that the differential voltage is zero volts.

$$V_{out} = V_2\left(\frac{R_4}{R_3 + R_4}\right)\left(1 + \frac{R_2}{R_1}\right)$$

When V_2 is set to zero and V_1 is set to a specific voltage, V_{out} will be:

$$V_{out} = -V_1\left(\frac{R_2}{R_1}\right)$$

Taking the sum of the two equations, V_{out} will be:

$$V_{out} = -V_1\left(\frac{R_2}{R_1}\right) + V_2\left(\frac{R_4}{R_3 + R_4}\right)\left(1 + \frac{R_2}{R_1}\right)$$

To simplify the circuit, if R_3 is equal to R_1 and R_4 is equal to $R_{2,}$ then the simplified equation will be:

$$V_{out} = \frac{R_2}{R_1}(V_2 - V_1)$$

The gain will simply be R_2/R_1. This could be useful in measuring the differential voltage across the Wheatstone bridge but it would also apply a load to the Wheatstone bridge so that the output voltages do not necessarily represent a balanced bridge.

2.7.6 Instrumentation Amplifier

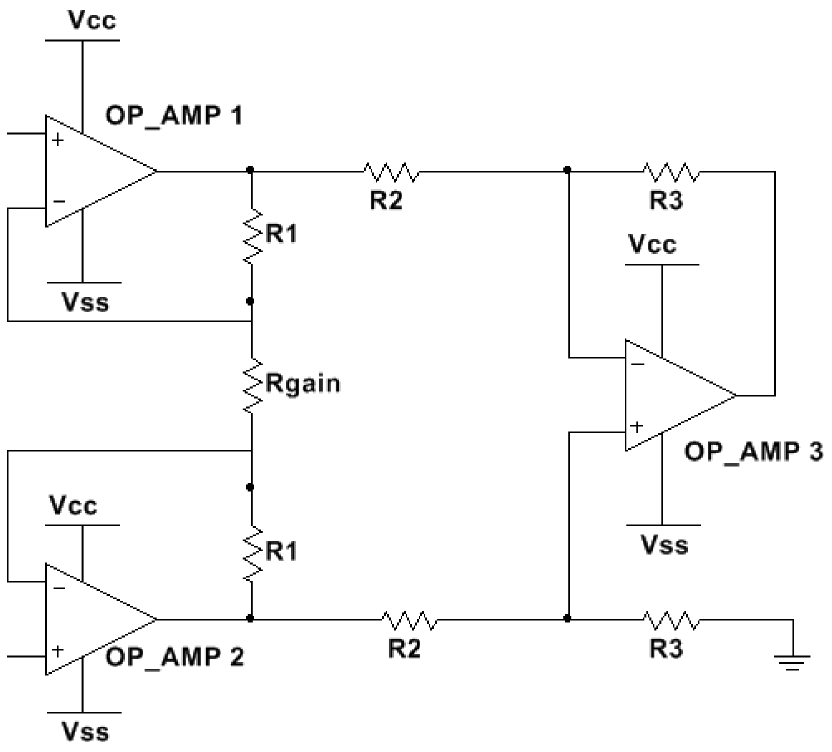


Figure 33: Instrumentation Amplifier

The instrumentation amplifier, shown in Figure 33, is a better configuration of a differential amplifier that applies a high impedance at the input so that the input voltage is measured more accurately. It serves as a means to amplify the differential voltage without applying a load to the sensor. The instrumentation amplifier can be seen as having two stages: the stage with a high input impedance or "buffer stage," and the stage with the differential amplifier.

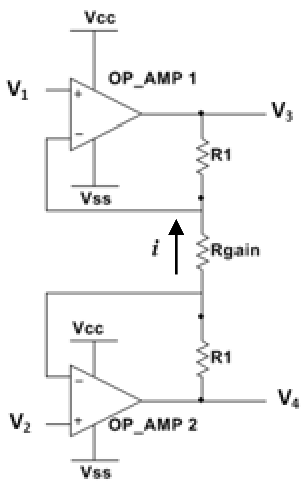


Figure 34: The Buffer Stage

Looking into the first stage of the instrumentation amplifier (seen in Figure 34) with the high input impedance, the current, *i*, will be the same through both R_1 and R_{gain} resistors since no current is allowed to enter the amplifier. Since the two op-amps are connected in a closed loop, the differential voltage at the input V_+ and V_- of the op-amp will be zero. This is so that for op-amp 1, if a voltage V_1 is applied to $V_{+,}$ then the voltage at V_- is also V_1 to allow for the differential voltage at the input to be zero. From this characteristic, the voltage drop across R_{gain} is:

$$V_2 - V_1 = i \times R_{gain}$$

The current through R_{gain} would then be:

$$i = \frac{V_2 - V_1}{R_{gain}}$$

Since the resistors are in series, the current through all the resistors are the same. The current can be derived to be:

$$i = \frac{V_4 - V_3}{R_1 + R_{gain} + R_1} = \frac{V_4 - V_3}{2R_1 + R_{gain}}$$

Setting both current equations equal to each other, the gain can be solved for the first stage of the instrumentation amplifier.

$$\frac{V_2 - V_1}{R_{gain}} = \frac{V_4 - V_3}{2R_1 + R_{gain}}$$

$$A_{v1} = \frac{V_{out}}{V_{in}} = \frac{V_4 - V_3}{V_2 - V_1}$$

$$A_{v1} = 1 + \frac{2R_1}{R_{gain}}$$

Looking into the second stage of the instrumentation amplifier, the configuration is the same as that of the differential amplifier. Therefore, it is already known that the gain will be:

$$A_{v2} = \frac{R_3}{R_2}$$

Taking the two gains from both stages, the total gain of the instrumentation amplifier will be the product of A_{V1} and A_{V2} so that the total gain is:

$$A_v = A_{v1} \times A_{v2}$$

$$A_{v=} \left(1 + \frac{2R_1}{R_{gain}}\right)\left(\frac{R_3}{R_2}\right)$$

To simplify the equation, let R_1 equal to R_2 and equal to $R_{3.}$ As a result, the gain equation can be modified to be:

$$A_v = \left(1 + \frac{2R_1}{R_{gain}}\right)$$

From the gain equation, V_{out} would then be:

$$V_{out} = (V_2 - V_1)\left(1 + \frac{2R_1}{R_{gain}}\right)$$

Having the gain be dependent upon one resistor value allows for a quick change in the gain. It is difficult to implement with individual resistors since resistors need to be matched precisely for the equations above to hold true. To compensate for this difficulty, there are commercially available IC chips that not only take up less space but will also more precisely match resistances.

2.7.7 Wheatstone Bridge with Instrumentation Amplifier

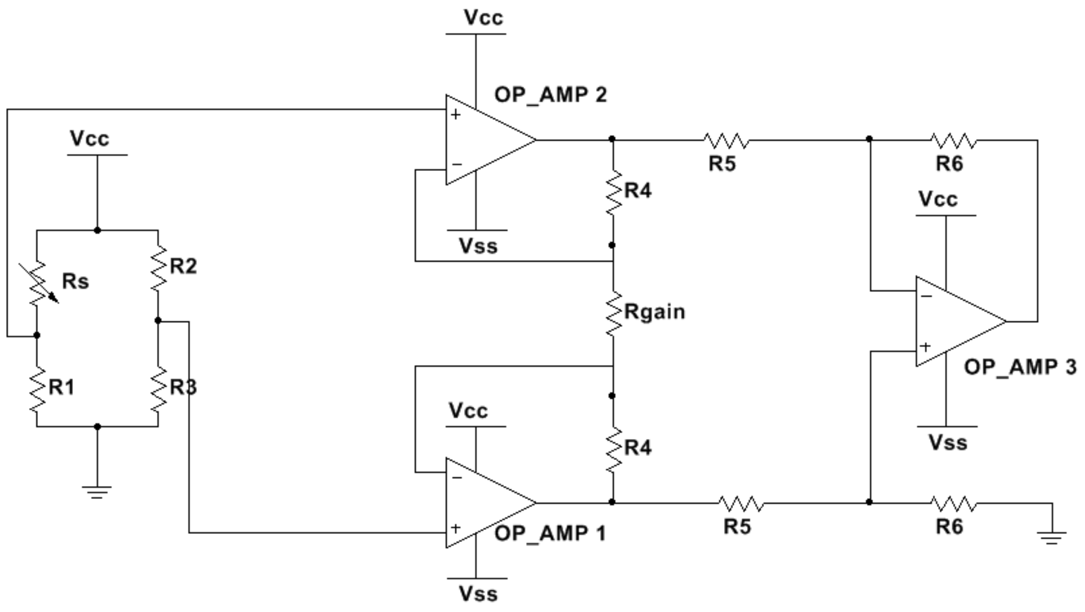


Figure 35: The Wheatstone Bridge and Instrumentation Amplifier Configuration

In the configuration shown in Figure 35, the Wheatstone bridge and instrumentation amplifier include both the advantages of an increase in sensitivity and a controlled gain. This circuit does have many advantages but it may do more than what is needed considering the FSR could already provide the needed sensitivity. This configuration would only provide more components and increase the cost and complexity.

2.8 Summary

In order to design an ergonomic musical device for a patient with a degenerative muscle disease, it is important to understand the disease, as well as how it presents itself in the patient. In addition, understanding music theory and how musical instruments work, as well as looking at existing musical assistive devices, provided the team with a foundation for creating a device that will allow its user to accurately play music. Furthermore knowledge in the types of sensors and circuits in different applications helped to decide the progression of the device.

3. Methodology

The following chapter discusses the steps taken to design and create the device. Due to the nature of the project, specific design criteria and constraints were considered in the design of the circuitry and hardware. Based on the criteria and constraints, the team was able to put together a working prototype.

3.1 Client Statements

Initially, the team was given a very broad statement in which the task was to design an ergonomic hand interface for a client with Duchenne Muscular Dystrophy (DMD). After researching DMD and conducting an interview with the client, the team was able to establish a more detailed, revised client statement. The initial client statement was as follows:

Initial Client Statement:

The client has a passion for playing drums and would like to be able to play the drums despite having DMD. The goal was to design and create a musical ergonomic hand interface that would allow the client to create and play music despite gradual muscle degeneration.

After interviewing the client, the team learned more about the client's interests and what he would like the device to do. The client's favorite instruments include the drums, the ukulele and the keyboard. However, the client already had alternative ways to play the drums and the keyboard. The client uses drum pads and the keyboard on an iPad app to play these instruments. However, the client finds it physically demanding to hold a ukulele properly. This piece of information narrowed the scope of the project to specifically designing an assistive aid to help the client play the ukulele. The client mentioned that he planned to take ukulele lessons and, as such, would like to be able to perform the fingerings of the ukulele.

From conducting further research on the progression of DMD, it was learned that the degeneration of muscles can be accelerated if muscles are used more often because once torn (due to exercising of the muscle), muscle fibers are not able to regenerate in DMD patients. Upon further testing, the client had mentioned that simply moving his arm across a piece of paper to write sentences is extremely tiring. Testing showed the range of motion the client had, allowing the team to revise the client statement.

Revised Client Statement:

The client has a passion for creating and experimenting with music and would like to continue to do so despite having Duchenne Muscular Dystrophy (DMD). The goal is to design and create an assistive aid that will allow the client to play the ukulele given his limited range of motion.

3.2 Design Objectives and Constraints

In order to design a successful device, objectives and constraints were outlined for the scope of this project. The following objectives and constraints are outlined below:

Objective 1: Mimic the fingerings of a ukulele.

- Being able to mimic the fingerings of a ukulele is necessary to consider because the client is considering taking ukulele lessons. This objective also contributes to the ease of use of the device since providing the client with a small learning curve when it comes to using the device will make the device easier to use.

Objective 2: Produce clean chord sounds

- The device should ultimately work to produce the same sound one would hear if one were to use one's fingers to strum strings to play a chord.

Objective 3: Be cost effective

- The device should be relatively cheap to create so that manufacturing costs can be kept low, making the device affordable. The cost must also stay within the team's allotted budget.

*Objective 4: Use parts that are easily replaceabl*e

- Considering that the client or the client's caregiver will be handling the device, should any part of the device malfunction, the part must be easily accessible and easy to fix.

Objective 5: Must be easy to set up and use

- The user and the caregiver should be able to set up the device without any trouble, and intuitively know how to turn the device on and off.
- The user should be able to know how to set up the ukulele and use it.

Constraint 1: Must not overexert muscles

- The range of motion required to use the device should be limited because if the client must not have to overexert his muscles.

Constraint 2: Must be safe

- This constraint takes priority over any other constraint. All electrical components that are in contact with the client should comply with the appropriate safety standards for electrical devices.

Constraint 3: Must sufficiently depress strings of the ukulele

- The team has learned that it is important that the actuation device depress a ukulele string fully in order to produce an acceptable sound when that string is strummed. As a result, any actuation should have enough force to fully depress a string to the neck of the ukulele.

Constraint 4: Must consider the size of the ukulele neck

- The ukulele neck is relatively small so the device that mimics the fingering of chords on the ukulele will need to fit the neck of the ukulele well enough to cover each of the four strings on the four frets being considered in this project. Only the first four frets are being considered for this device because the majority of the chords are played in these frets. Narrowing the scope of the project to the first four frets of the ukulele makes it easier for the team to adhere to this particular constraint.

3.3 General Architecture

Overall, the design has four main aspects: input hardware, microcontroller and software, output hardware, and additional features. The input hardware worked to provide a way in which the user interfaces with the rest of the device. The microcontroller and software were used to make decisions (such as which strings to press and on which frets) after receiving an input signal from the input hardware. The output hardware consisted of circuitry that was driven by the output of the microcontroller and includes the physical actuation used to press down on the appropriate string(s). Additional features included parts of the design that were not necessary for the user to have, but benefited the user by making the device more user-friendly.

3.3.1 Input Hardware

Power

An important aspect of the entire design to consider was how it was going to be powered. An AC to DC adapter was used from CUI Inc. that took the 120V AC signal from a wall outlet and converted it into a constant 24V DC voltage.

Other parts of the design (e.g. the microcontroller) were not capable of running off of a 24V supply. To compensate, a step down DC-to-DC regulator was required to bring the 24V down to a constant 3.3V output. At 3.3V, the microcontroller was able to run effectively. The voltage regulator was the OKI-78SR-3.3/1.5-W36-C from Murata Power Solutions Inc. The regulator is able to receive an input voltage from 7V to 36V and produce a constant 3.3V output. For added safety, a 5A fuse from Bel Fuse Inc. was added into the power system. The full power circuit diagram can be seen in Figure 36, where the header "J8" represents the 24V output from the AC to DC adapter, "F1" signifies the 5A fuse, and "U3" is the DC-to-DC voltage regulator.

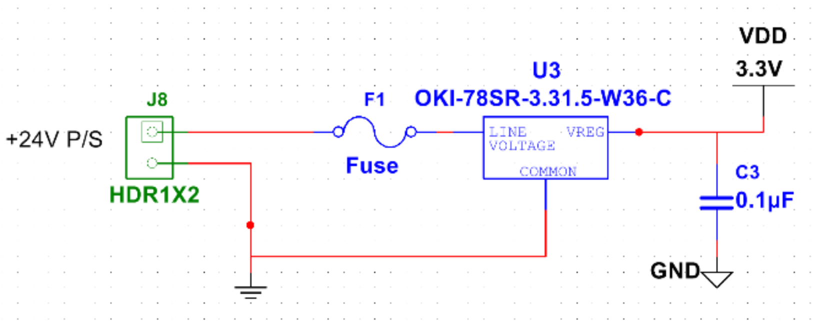


Figure 36: Power Circuitry

User Interface Input Sensors

Three user interface options were considered: push buttons, FSRs, and capacitive touch sensors. Each interface was examined for its ability to be ergonomic, replaceable, affordable, and adaptable to the progression of DMD.

Capacitive touch pads were considered because the subject was comfortable using a phone with a touchscreen. However, as DMD progresses, most patients' undergo wrist flexion contracture, a process in which the patient's hand curls inward at the wrist. As a result, a capacitive touch pad will not be able to adapt and fit into the user's hand. Compared to the other interface options, a capacitive touch pad was higher in price but would still offer the sensitivity required to avoid straining the user. However, capacitive touch pads cannot adapt to the progression of DMD and therefore, this user interface option was no longer considered as a viable option for this particular application.

Push buttons were the simplest type of user interface because they only had two states, on or off. They were inexpensive and could have been placed in any configuration that allowed the user to push down on. In addition, buttons were easily replaceable if one were to break. Despite these positive specifications, it was determined that, similar to a capacitive touchpad, a button could not adjust to differing force capabilities. Due to the fact that buttons cannot detect different forces then buttons cannot easily account for the progression of DMD, especially in the event that a patient undergoes wrist flexion contracture.

FSRs were ultimately used because while FSRs in this application simply acted as a button, the ability to detect the amount of force being applied may provide useful information. The progression of DMD affects each patient differently so the threshold of the FSR, a value that ultimately determines whether a press was accidental or intentional, can be lowered in software if a user cannot exert the same amount of force from the finger. This ability made FSRs ergonomic and highly adaptable to the progression of the disease, characteristics that capacitive touch pads and buttons seemed to lack. The FSRs were also cheap, replaceable, and could be made to be ergonomic through the use of thresholding (controlled through software).

Clay Mold Iterations

The creation of the clay mold began with the idea of having a highly ergonomic hand interface that would cater to the shape of the subject's hand. As a result, a computer mouse-like mold was created using blue polymer clay and was given to the subject, and a mold of the shape of his left hand was taken. Figure 37 shows the original blue mold made.

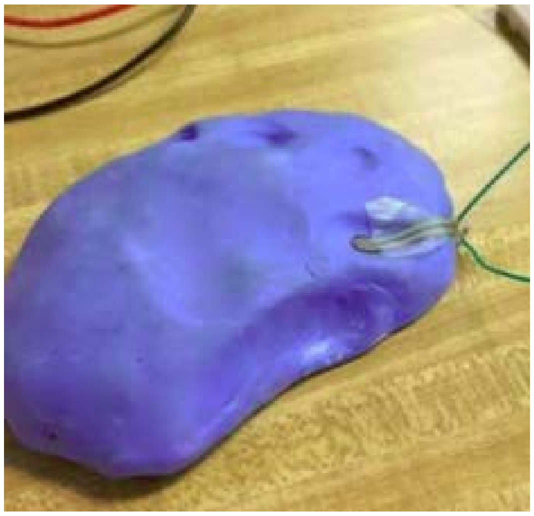


Figure 37: Original Clay Controller (blue)

Once it was determined that using a moldable substance like polymer clay to create the hand controller was a viable option for making a customizable user interface, the question arose as to whether or not FSRs would work well on hardened polymer clay. The FSRs were attached on the blue mold to determine whether or not they would provide a relatively clean signal in response to a change in force. Figure 37 shows the setup of the FSRs on the blue mold.

It was determined that the blue mold originally created did not have a smooth enough surface to allow the FSRs to provide a clean signal in response to changes in force. Pre-loading of the FSRs (due to the bending tensions exerted on the layers of the sensor's body) distorted the signal and, at times, no signal was detected at all. Thus, the testing of FSRs on baked polymer clay remained incomplete until another mold was created.

While in the process of creating an entire new mold, a smaller one was made so that tests for the FSRs may be continued. Figure 38 shows the finger mold made to test whether the FSRs can produce a clean signal.

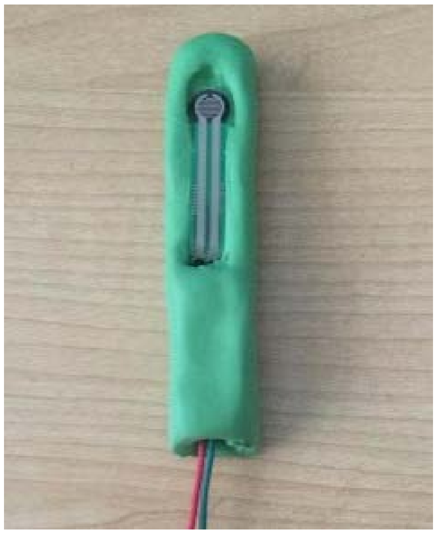


Figure 38: Finger Mold for Testing FSRs

Using the finger mold seen in Figure 38, the team determined that FSRs do indeed work well on baked polymer clay, provided that the surface of the clay is smooth.

A new, smoother controller had to be made while still accounting for the subject's natural finger placements. In order to create a smoother controller, a clay negative of the original mold was made. Plastic wrap was wrapped around the original mold in order to prevent the unbaked clay (negative mold) from sticking to the baked clay. This process can be seen in Figure 39.

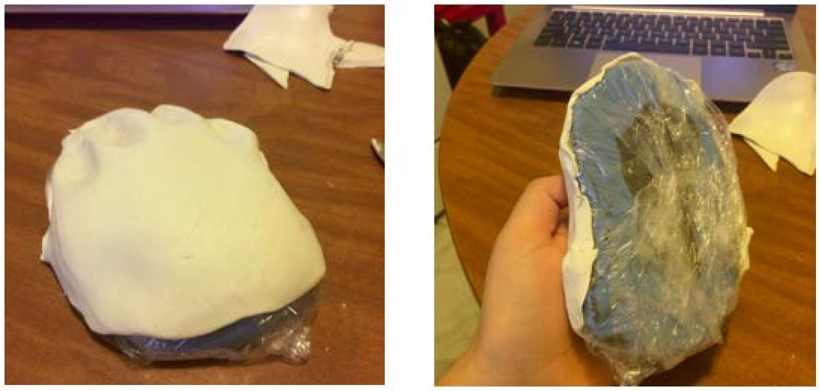


Figure 39: Molding the Negative Clay Mold (white)

After the negative unbaked mold was made from the original baked mold, more clay was added to the unbaked mold in order to fill it out. In addition, the mold was smoothed out in order to account for any bumps, bends, and unwanted ridges. The new mold was then baked to have it harden.

Through trial and error it was concluded that the best way to organize the FSRs would be to have them protrude off the front of the clay mold so that all the FSR wires could be collected and organized into a single bunch. This organization would also work with how the plans to organize the rest of the board, since both the driver circuit for the FSRs and the microcontroller would be placed relatively far away from the subject. Having the FSR wires come out of the back of the clay mold would make this difficult to implement, so the new iteration of the clay mold had to account for the body of the FSRs. The newest iteration can be seen in Figure 40, both of which show an extension made on the clay mold to hold the body of the FSRs.

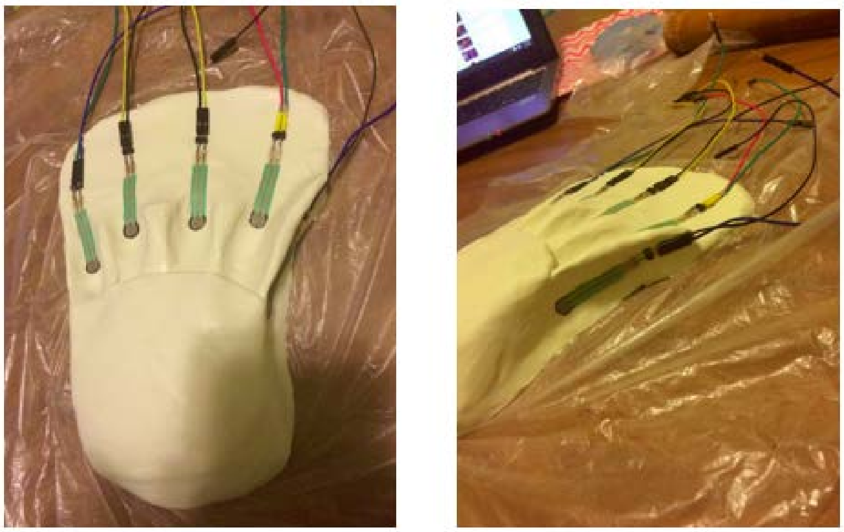


Figure 40: Top and Side View of New Mold

Prior to baking the new clay mold, the FSRs were placed on the mold prior so that there would be indentations in the mold to house the FSRs to allow for a better fit after the mold was baked. After the indentations were made, the FSRs were removed from the unbaked clay mold. The mold was then baked in order to create a hardened and permanent hand controller.

A polycarbonate thermoform plastic sheet was used to make a protective cover over the region of the mold containing the FSRs. The plastic sheets would allow for better protection of the body of the FSRs for long-term use. The team decided against using plastic to cover the entire clay mold in order to preserve the ergonomic feeling of the baked clay.

The polycarbonate sheet was cut to fit the area of the mold that was being focused on. It was then thermoformed using a heat gun, since placing the sheet in the oven would not allow for a thorough reconstruction of the plastic. The thermoformed plastic and the clay mold can be seen in Figure 41. The use of a heat gun allowed for the creation of miniscule deformities in the plastic, giving a more accurate shape for the controller.

Figure 41: Polycarbonate Plastic Cover for Clay Mold

FSR Circuitry

Once the FSRs were chosen as the preferred user interface input sensor, analog circuitry needed to be found to create an appropriate input to the microcontroller. The FSR was used as a pull-up resistor in series with a 51kΩ. When the FSR did not sense any force, it would provide infinite resistance. The infinite resistance would drive the positive input of a unity gain buffer to zero, which also caused the output of the buffer to be zero. When the FSR sensed a force, the resistance dropped. If the FSR was pushed with the maximum amount of force, the team measured the resistance to be approximately 20kΩ. The complete FSR circuitry can be seen in Figure 42, where "J1" represents a header connected to an FSR.

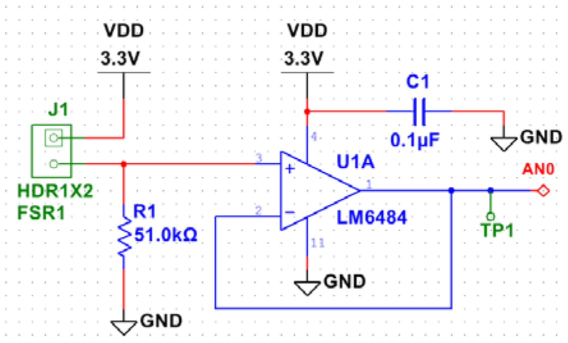


Figure 42: FSR Circuitry

A voltage follower configuration between the pull up resistors and the microcontroller was selected as a means to not load the microcontroller. If a voltage follower was not used to separate the two points then the microcontroller will not read the accurate voltage from the FSR. The buffer selected was an LMC6484 quad op amp. This op amp was ideal because it required little power, allowing it to be powered by the 3.3V supply coming from the voltage regulator. It was also a Rail-Input-Rail-Output op amp, which ensured that the output went from 0V to 3.3V. The output signal from the op amp matched the input signal when the FSR was pressed, which created an appropriate signal to be used as an input to the microcontroller. The capacitor, "C1", that is seen in Figure 42 was a decoupling capacitor from the 3.3V power supply to get rid of any possible noise that could have come from the power supply.

The selection of the 51kΩ resistor was not a trivial matter. Originally a 100kΩ resistor was selected. This meant that the output voltage (Vout) would be derived from the voltage divider equation:

$$Vout = \left(\frac{R2}{R_S + R2}\right) * Vin$$

where R_S was the changing resistance of the FSR, R2 was the fixed resistor in series with the FSR, and Vin was the 3.3V supply from the voltage regulator. With the FSR pressed at full force (R1 ≈ 20kΩ) and R2 = 100kΩ, the output voltage was 2.75V. When tested with the unity gain buffer, it was discovered that the FSR was too sensitive. It was desired to have the user to be able to rest his fingertips on the FSRs without causing an output from the buffer.

Due to the sensitivity of the FSR, the same test was attempted with a 51kΩ resistor, replacing the 100kΩ resistor. The output voltage of this test was 2.37V, about 86% of the output voltage with the 100kΩ resistor. When tested, this output voltage, the fingers were able to lightly rest on the FSR without causing the output of the unity gain buffer to be too high.

3.3.2 Microcontroller and Software Architecture

Microcontroller Options

The microcontroller can be considered a "central hub" since it read signals from the input sensors and wrote output signals to actuators. Overall, its purpose is to determine how the inputs will control the outputs. The microcontroller needed for this design requires several features that are common in many microcontrollers.

The original design required 16 outputs for the actuators and five inputs for the input sensors. Thus, a microcontroller with at least 21 general input and output pins was originally needed. Because the team was not designing anything that will require a lot of memory, memory size was not considered when selecting a microcontroller.

Other concerns that the team had in choosing a microcontroller involved ease of programming, debugging, and testing. The team wanted something that had an inexpensive development board to assist with the programming of the microcontroller and provide a board layout that made it easy to test individual inputs and outputs. The following table shows microcontrollers that were originally considered:

Table 5. Microcontroller Options

Microcontroller	Manufacturer	General Inputs/Ouputs	Price for 1 Device
MSP430G2553	Texas Instruments	24	$2.73
PIC18F23K20	Microchip	24	$1.70
PIC16F882	Microchip	24	$1.83
ATmega48	Atmel	23	$3.33
C8051F410	Silicon Labs	24	$6.02

Microcontroller Selection

As each microcontroller met the team's required specifications, a choice was made based on the lowest price, as well as ease of programming. At first, the team chose the PIC18F23K20 as it was the cheapest on the list, and programming this specific microcontroller was deemed easier than the others. Also, the development board for this device, the PICDEM Lab II, was relatively inexpensive compared to the development boards offered by other manufacturers.

As development of the device progressed, the team's design changed and required more inputs and outputs. As stated, the original design required five inputs for the FSRs and 16 outputs for the actuators. The team also decided that the device should be able to communicate with a computer through universal synchronous-asynchronous receiver/transmitter (USART) serial communication. USART requires one input pin for the receiver and one output pin for the transmitter. The team considered that the device would also require some sort of display so that the user would be able to tell which musical key he is in while playing. It was decided that a simple LED display would be used to complete this task since it is easy to implement. As there are seven keys, each of which can be flat or sharp and major or minor, there would need to be a total of nine LEDS, which corresponds to nine more outputs. There would be seven LEDs that correspond to the seven keys (A, B, C, D, E, F, G), one LED to correspond to whether the key was sharp or not, and one LED to determine if the key was major or minor. After finishing the initial design of the display, the team also realized that two more input buttons would be needed to give the user control over which musical key he would like to play in (i.e. major or minor). Thus, two more inputs would be required: one button to cycle through all the roots of the keys (A, A#, B, etc.), and another to select whether the key would be major or minor. With all of the new considerations, the microcontroller would now need at least 33 input/output pins in total.

At the point in time that the team realized more pins would be needed, the PICDEM Lab II development board was already being used for microcontroller testing. The team did not want to purchase a new development board that supported a larger chip from a different manufacturer. Instead, the team noticed that the PICDEM Lab II development board supported a bigger 40-pin microcontroller, called the PIC18F46K20, which had 35 general input/output pins. This bigger microcontroller could support the new requirements, was only $1.25 more expensive than the PIC18F23K20, and eliminated the need to buy a new development board. Therefore, the team purchased the PIC18F46K20 and incorporated it into the design of the device.

Software Architecture

A microcontroller must be embedded with code so that it will know how to control the pins. In the case of this device, the code told the microcontroller what to do with the analog data collected from the FSR inputs and then set different outputs to determine which solenoids should be actuated. This description is the simplest way to describe what the microcontroller is doing, and will be discussed further in this section.

Top-Level Design

The code can be viewed at a higher level by looking at the block diagram in Figure 43 below. From this diagram, one can see that the device is being run by a 16 MHz clock. This clock is then stepped down to about 1 kHz by using a special-event trigger which is controlled by a timer. The ADC channels read the input from the FSRs every 1 millisecond timer iteration. The main while loop of the code then takes the data points taken by the ADC through shared memory and uses these data points to write to the outputs to control the solenoids. The main loop was also used to turn on the key select LED display. The 1 kHz interrupt is also used to trigger the USART control code which controls the serial communication between the device and a computer. This serial communication is transmitted and received through a USB interface. The data that are received from the computer are stored into electrically erasable programmable read-only memory (EEPROM) so that the device remembers those data even after being powered off. These EEPROM stored data are used to save which threshold should be used as well as which playing mode the device is in.

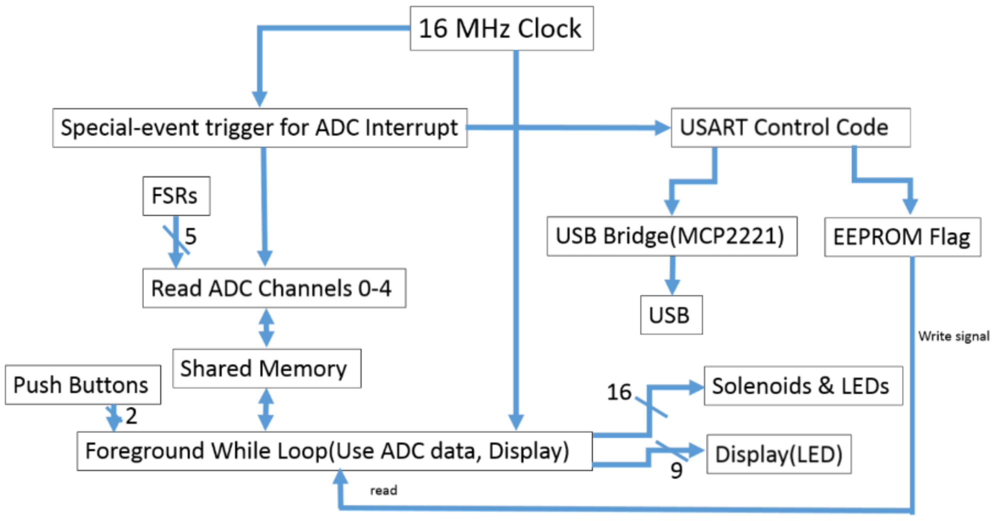


Figure 43: Top-level Block Diagram for the Software Architecture

Microcontroller Configuration and Initialization

Eight-bit PIC microcontrollers from Microchip are coded in the C programming language and use a chip-specific compiler called XC8. When programming PIC microcontrollers with the XC8 compiler, one first needs to set configuration bits to determine which features and settings of the microcontroller would be used. In this code, the configuration bits were mostly left at their default settings. However, a couple of changes were made. The watchdog timer and brown-out reset features were turned off in the code as they were not needed. The configuration bit labeled FOSC, which determines the oscillator the microcontroller would use, was set so that the microcontroller would use its internal clock. Using the internal oscillator would allow us to use two more general purpose input/output (GPIO) pins.

After setting up the configuration bits, the next step was to initialize different peripherals of the microcontroller. Initializing peripherals on a PIC microcontroller means setting specific bits in a register that correspond to the desired peripheral. An example of this bit setting in the C programming language can be seen below. This code configures the internal clock using the OSCCON and OSCTUNE registers. The selected bits configure the microcontroller to run as the primary clock running at 16 MHz. In this case, the phase-locked loop (PLL) was not needed to create a 16 MHz clock, so it was disabled.

```
OSCCONbits.IRCF = 0b111; // 16 MHz source

OSCCONbits.SCS = 0b00;   // primary clock

OSCTUNEbits.PLLEN = 0b0; // PLL disabled
```

Reading the data sheet on the chosen microcontroller is vital to determine which registers and bits are needed to use a specific peripheral. Multiple functions were created (as can be seen in Appendix A) to set these bits in their corresponding registers.

The first function, labeled *PIC_Init*, was used to disable all interrupts, set up the internal oscillator to 16 MHz, and configure each pin to be either an input or output. The next function, labeled *timer1_setup*, initializes timer one on the PIC chip. This function first disables the timer one peripheral interrupt and initializes the timer to run at ¼ of the 16 MHz clock. It then selects timer one as the clock source for the Capture/Compare/PWM (CCP) peripherals and starts the timer at 0. When the timer completes a one millisecond cycle, it will trigger the CCP, which is configured as a special event trigger. This special event trigger is in turn used to trigger the ADC interrupt, which is used to collect analog data from the FSRs and convert it into an integer. The ADC is initialized in the function *ADC_Init*. This function sets up five inputs as analog inputs so that analog signals could be received. This function also determines how the ADC register will receive and store captured data. In this case, the ADCON2 register is right justified, meaning that a full 10-bit conversion is used to determine the digital values. The ADC is also configured to have a 1 microsecond bit conversion rate (defined as T_{AD}). Finally, this function enables the ADC and peripheral interrupts.

The EEPROM must be initialized for first use of the device. The function, *EEPROM_Init* was created to check the contents of the set EEPROM addresses. If any of the addresses have values that cannot be used by the main code, this function will store in a default value to the EEPROM. This function checks both the addresses used for each FSR as well as the address used to store which play mode the user is in. This function should only ever run once in the devices lifetime, and is only there for first time initialization after the microcontroller is first programmed and removed from the development board.

Serial communication between the device and a computer was desired by the team and the advisors. This communication was desired so that the data going into the ADC could be monitored to make sure it was functioning correctly. Serial communication could also be used to create a kind of user interface for the subject. Using the software called PuTTY, a Secure Shell (SSH) and serial communication client, the serial output from the microcontroller could be monitored. As described earlier, USART was used as the method for serial communication. The USART was initialized in the *UART_Init* function. Using the XC8 compiler's legacy peripheral libraries, this function calls a secondary function called *OpenUSART* which configures the baud rate for the serial communication using two variables. The first variable is a character type which determines how the USART should be configured. This character type variable was set equal to certain C macros which tell the USART that a receiver interrupt would be used, configures the USART to asynchronous, eight bit mode, and runs the USART at a high speed. The second variable of integer type determines the actual baud rate that the USART should run at using the following equation:

$$Desired\ Rate = \frac{FOSC}{16(X+1)}$$

In this equation, FOSC is 16 MHz, the desired rate is 9600, and X is the variable that must be solved for to be used in the *OpenUSART* function. Thus, the second variable would be set to 103 to get a baud rate at 16 MHz. The *UART_Init* function further sets up the receiver interrupt for serial communication by resetting the receiver interrupt flag, making the receiver interrupt low priority, and then finally enabling the receiver interrupt.

ADC Interrupt

Once all the needed peripherals were initialized, the interrupts that would periodically run the ADC as well as the USART could be written. The ADC interrupt was the first interrupt to be written as it is the highest priority interrupt. As described above, this interrupt is periodically triggered every 1 millisecond to capture data that are being inputted into the analog pins.

The ADC interrupt first checks if the ADC interrupts are in fact enabled and if the ADC interrupt flag has actually been set, indicating that the ADC has been triggered. The interrupt will then create two local variables that are used to store the ADC data into a buffer as well as change the channel that the ADC is using to collect data. The first variable **iADC_samples* is a pointer which points to the buffer *ADC_samples* that stores all the data collected from the ADC. The second variable is a character that is used to switch between the different channels of the ADC. After these two local variables have been created, the interrupt resets the ADC interrupt flag in preparation of the next ADC interrupt. The interrupt must then check if the ADC is starting a new capture by using the global variable *SampleCount*. If this variable is equal to zero, then the ADC is presumed to be starting a new capture and an ADC data conversion should have been completed for the first channel. If this is indeed the case, then the pointer must point back to the start of the *ADC_samples* capture buffer. Since it is presumed that a data conversion has been completed on the first channel already, the *ADC_Channel* variable must also be set equal to one in this "if" statement. By completing this statement, the interrupt would select the second channel for data conversion.

If *SampleCount* is not equal to zero, indicating that a new capture has not started, then the interrupt must then set the proceeding ADC channel to prepare for the next conversion. Now, the ADC values that were found after a completed conversion can be read into the *ADC_samples* buffer for the selected channel. The next step would be to select the next channel and then check if this channel was the last channel to be converted. If the channel selected is the last channel, the first channel, channel zero, would need to be selected next.

Finally, the interrupt will check if the capture buffer is full by reading the *SampleCount* variable. If this variable is greater than or equal to the ADC sample size times the number of ADC Channels, then the capture buffer *ADC_samples* is full. In this case, *SampleCount* must be set back to zero so that the next iteration of interrupts will know to start a new capture. The first channel, channel zero, must also be selected to start a data conversion on the next capture. The ADC interrupt must then be disabled so that further ADC conversions will be stopped. This will also indicate that the capture buffer is full and that the data conversion cycle needs to start over.

The USART control code embedded into the ADC interrupt serves two main purposes. The first purpose is to provide an output stream of the data that the ADC is collecting. This data can be used to make sure the ADC is functioning properly. The second purpose is to provide a basic user interface to the user. Through USART, the user is able to select different thresholds for the force sensitive-resistors. These thresholds will let the user decide how hard or soft he would like to press on them. The user interface will also allow the user to determine which mode he would like the device to be in. Each mode determines what the FSRs do. For example, the default mode lets the user select different keys as well as what chords each FSR corresponds to. A second mode will put the device into single button-single solenoid mode. This means the one FSR will correspond to only one solenoid on one string. The fifth FSR will be used to determine which fret the other four FSRs will correspond to.

The USART control code first checks an index variable and makes sure that it is not greater than a set buffer size. This "if" statement is in place to make sure that the buffer, which is used to read user inputs from the computer and is called *rxbuffer*, is not full. If the buffer is full, there is an else statement in place that is used to clear the buffer of any data. If the buffer is not full, the next line of code is used to read data from the USART and place it into the receive buffer. The code then checks the contents of the buffer and will do one of three things based on the contents of the buffer. The user may select any three of these options.

Option 1, deemed the threshold selection option, has three subset options, each of which controls the threshold for an individual FSR. Option 1a[x], which is the light threshold option, is initiated if the contents of the buffer contain the characters "1", "a", a number between 1 and 5, and "enter", in this order. The [x] variable, or the number between 1 and 5, determines which FSR's threshold needs to be changed. The number 1 corresponded to the thumb, 2 corresponded to the index finger, 3 corresponded to the middle finger, 4 corresponded to the ring finger, and 5 corresponded to the pinky finger. To select a threshold, the complete string followed by the return key must be entered into the computer. For example, the string "1a1" followed by pressing enter will change the threshold for the thumb to a light threshold. Similarly, option 1b[x] will change the selected finger to a medium threshold, and option 1c[x] will change the selected finger to a hard threshold. When any of the options are detected, a specific flag is set which corresponds to the specified threshold. Then, a function called *USARTswitch* is called, which takes in both the [x] variable data point and a global threshold-specific variable. Based on what the [x] variable was, the function will pass by reference a number (which specifies the selected finger) into the threshold-specific variable. Both the set flags that correspond to a specific threshold and numbers that equate to fingers will eventually be read in the main while loop to determine what variables should be written into EEPROM. These variables are written into EEPROM to save the specified threshold after power-down. The code was written in such a roundabout way because writing to the EEPROM is a lengthy operation, and it would be very inefficient to do this operation in an interrupt.

Option 2 is triggered if the contents of the *rxbuffer* contains "2" followed by "enter". This option will either enable or disable the ADC values viewer option. This option simply sets a variable to either one or zero depending on the last value of the variable. When the variable is set to one, the USART will stream the collected ADC data to the computer from all 5 channels of the ADC. Once the variable has been set, the *rxbuffer* will clear and the interrupt will return.

The final option, option 3, has two subset options. This option is deemed the chord selection mode. This option is enabled if the rxbuffer contains a "3", an "a" or "b" and an "enter" character. If option 3a is selected, the device will go into "Smart Ukulele Mode", which is the regular play style mode. If option 3b is selected, the device will go into "Solo Mode", where each FSR corresponds to only one note. When either of these options are selected, a flag is set and a chord selection variable is set to either 1 or 0. The flag is so that code in the main loop will run. If the flag is set, the chord selection variable will be written into EEPROM. If the chord selection variable is 0, then the device is in the "Smart Ukulele Mode". If the variable is 1, the device is in "Solo Mode".

If none of the three described options are selected, the buffer will be scanned until an option is selected or the buffer becomes full. If either of these events happen, the buffer will be emptied, and then the new contents of the buffer will be scanned.

Main Code and Functions

Now that both of the functions to initialize the device and the interrupts had been written, the main code was written and analyzed. The first operation the main code should do is call all the initialization functions that were described in the previous paragraphs. Once initialization was complete, global interrupts were enabled, effectively starting the interrupts that were created. After both these tasks were complete, the main infinite while loop will run. In this loop, a few different functions are used. A couple of these functions determine how the ADC is used, but the majority of these functions lay out the algorithms that are used to determine the musical key and chord that the user would like to play.

The first ADC function called *ADC_Capturestate_t* simply determines whether the ADC is busy capturing more data. This function returns that the ADC is busy if the ADC interrupt is enabled. Otherwise, the function will return that the capture is done. This function is used at the beginning of the infinite loop before anything else is done. This ensures that the ADC is not busy and the data in the ADC capture buffer can be properly read. The next ADC function called *ADC_StartCapture* starts a new capture for the 5 ADC channels. This function is placed at the end of an iteration of the infinite loop. It first stops the ADC interrupts and then selects the first channel, channel zero, to run a data conversion. The function then synchronizes with the CCP ADC trigger by first clearing the ADC interrupt flag and then waiting for it to be set again. Finally, the function sets the *SampleCount* variable back to zero and re-enables the ADC interrupts. This function must stop the ADC interrupts in order to properly synchronize with the CCP special event trigger and timer. If the interrupts are not stopped, the interrupt risks desynchronization with the trigger, causing the code to become even less periodic. The *ADC_StartCapture* function causes the interrupt to become nonperiodic, which may cause problems if the code is updated in the future.

The first task the main while loop did after checking the ADC capture status was run a debouncing function to debounce the buttons that are used for key selection. This debouncing function, provided by Professor Gene Bogdenov, provides a sort of hysteresis to a button press. The function checks to make sure that the button has been read as pressed for at least 2 consecutive cycles before it is actually read as pressed. Likewise, the function checks if the button has been released for at least 5 consecutive cycles before it is actually read as released. As the buttons that were used were mechanical, it was vital that this debouncing code be implemented to ensure that the buttons were being read properly.

The next function the main while loop runs is the *USARTdatacheck* function. This function will first output messages to the user if the device is indeed connected to a computer. These messages will tell the user how to operate the USART. This function will then check if the flags were set in the interrupt. If any of these flags have been set, the function will write a value to the EEPROM that will correspond to a specific threshold for an individual finger. The function will then output to the user what threshold they selected for the selected finger. This function will also check if the flag that determines play mode selection has been set. If flag has been set, the function will write the selected mode to EEPROM and output to the user which mode they are in.

The main while loop will then initiate a couple of “for” loops to scan through all the data collected by the ADC. The first “for” loop is used to scan through the ADC channel, while the second “for” loop is to scan through the collected samples in each channel. In these embedded “for” loops, a function called *USART_ADCvalues* is called. This function will first determine if the ADC values viewer has been turned on by the interrupt. If it has been, the function will take in both “for” loop's indices and use them to output the ADC samples of each channel. After this function has been called, the FSR EEPROM addresses are read in order to determine which threshold should be used for each finger. Once the threshold has been decided, a function called *ADC_Read_Threshold* is called, which determines if the value in the ADC buffer is greater than the specified threshold hold. If the value is larger, the FSR input will be read, and then code to determine which solenoids should be actuated will run.

Mapping from FSR Samples to Chords

This part of the design involves two steps. First, the raw ADC samples are taken from the ADC buffer and converted into an integer which stores the state of all five input channels. This integer is called *raw_input_code* in the design and is often represented as a binary number. The least significant five bits of *raw_input_code* correspond to the five FSRs. The first least significant bit to the fifth least significant bit correspond to the FSRs pressed from the user's thumb to their pinky, respectively. When a bit is set to 1, it indicates that a corresponding FSR is being pressed. In the same way, 0 means the FSR is released. The way this design determines if an FSR is pressed, is by detecting the specific threshold that is determined by the user and written to EEPROM. If the

ADC sample is equal to or higher than the threshold, the bit of that FSR will be set to 1. When the ADC sample value drops under the threshold, the bit will be set back to 0. Once *raw_input_code* is set, a function named *debouncer* was called to prevent the device from actuating a transition chord. A transition chord in this project is defined as an unintended chord when transitioning from one chord to another. For instance, when the user wants to switch from a chord that can be actuated by pressing the thumb to another chord that requires the user to press down the index finger and the middle finger simultaneously, a chord corresponding to simply pressing down the index finger or middle finger could occur since it's nearly impossible for humans to press two fingers down at the exact same time. Therefore method *debouncer* is created to wait for a certain amount of time for the input to settle until it updates the value of a debounced input code called *debounced_input*.

Table 6. Chords in Major Keys

Finger	Thumb	Index	Middle	Ring	Pinky	Thumb & Index	Index & Middle	Index & Middle & Ring
Input_code Key	00001	00010	00100	01000	10000	00011	00110	01110
A Major	A Major	B Minor	C# Minor	D Major	E Major	F# Minor	G# Dim	G Major
A# Major	A# Major	C Minor	D Minor	D# Major	F Major	G Minor	A Dim	G# Major
B Major	B Major	C# Minor	D# Minor	E Major	F# Major	G# Minor	A# Dim	A Major
C Major	C Major	D Minor	E Minor	F Major	G Major	A Minor	B Dim	A# Major
C# Major	C# Major	D# Minor	E# Minor	F# Major	G# Major	A# Minor	C Dim	B Major
D Major	D Major	E Minor	F# Minor	G Major	A Major	B Minor	C# Dim	C Major
D# Major	D# Major	F Minor	G Minor	G# Major	A# Major	C Minor	D Dim	C# Major
E Major	E Major	F# Minor	G# Minor	A Major	B Major	C# Minor	D# Dim	D Major
F Major	F Major	G Minor	A Minor	A# Major	C Major	D Minor	E Dim	D# Major
F# Major	F# Major	G# Minor	A# Minor	B Major	C# Major	D# Minor	F Dim	E Major
G Major	G Major	A Minor	B Minor	C Major	D Major	E Minor	F# Dim	F Major
G# Major	G# Major	A# Minor	C Minor	C# Major	D# Major	F Minor	G Dim	F# Major

Table 7: Chords in Minor Keys

Finger	Thumb	Index	Middle	Ring	Pinky	Thumb & Index	Index & Middle	Index & Middle & Ring
Input_code Key	00001	00010	00100	01000	10000	00011	00110	01110
A Minor	A Minor	B Dim	C Major	D Minor	E Minor	F Major	G Major	E Major
A# Minor	A# Minor	C Dim	C# Major	D# Minor	E# Minor	F# Major	G# Major	F Major
B Minor	B Minor	C# Dim	D Major	E Minor	F# Minor	G Major	A Major	F# Major
C Minor	C Minor	D Dim	D# Major	F Minor	G Minor	G# Major	A# Major	G Major
C# Minor	C# Minor	D# Dim	E Major	F# Minor	G# Minor	A Major	B Major	G# Major
D Minor	D Minor	E Dim	F Major	G Minor	A Minor	A# Major	C Major	A Major
D# Minor	D# Minor	F Dim	F# Major	G# Minor	A# Minor	B Major	C# Major	A# Major
E Minor	E Minor	F# Dim	G Major	A Minor	B Minor	C Major	D Major	B Major
F Minor	F Minor	G Dim	G# Major	A# Minor	C Minor	C# Major	D# Major	C Major
F# Minor	F# Minor	G# Dim	A Major	B Minor	C# Minor	D Major	E Major	C# Major
G Minor	G Minor	A Dim	A# Major	C Minor	D Minor	D# Major	F Major	D Major
G# Minor	G# Minor	A# Dim	B Major	C# Minor	D# Minor	E Major	F# Major	D# Major

The second step of this process is to convert *input_code* into a chord. A chord will be represented as an array of notes. The code has an array named *notes* containing the string representations of all the notes. Each of these notes is a half tone away from its neighbor, and the sharp sign "#" is being used here to simplify the naming of notes. This array serves as a lookup table (shown in Table 6 and Table 7 above) that assigns an integer (the index) to each note. Notes in this design are represented by these integers. As a result, a chord is really an array of indices of notes. To get the chord based on *input_code*, the function *getChord*() is being used. This function uses switch cases to map from *input_code* to a chord. *getChord*() calls another function named *formChord*() to form the array of notes once the root of the chord *root* and the type of chord *type* is determined. *formChord()* takes in the root of a chord, and the type of a chord. There are three types: major, minor and diminished chords. In the variable *type*, 0 means major, 1 means minor and 2 means diminished. According to this information, as well as music theory, a chord is formed. In music theory, a major triad chord is 1-3-5, where 1 is the root note (explained in the background section 1.4.1), 3 is 4 half tones above the root, and 5 is 7 half tones above the root. Therefore, an array of a major chord could be represented as [root, root+4, root+7]. Using the same algorithm, a minor chord is [root, root+3, root+7] and a diminished chord is [root, root+3, root+6].

Mapping from Chords to Output Values

This part is done by calling the *chordToPin()* function. This function finds all the notes in a chord on each string of the ukulele and outputs correct values to pins that are connected to the solenoids and the LEDs. It loops through all four strings on the ukulele to find at least one note that belongs to the chord selected. The algorithm is currently only searching the notes within the first four frets of the ukulele because the hardware can only actuate in that range. When a note is found on a string, this function puts the fret location of that string into an array. The indices of this array correspond to strings on the ukulele. For instance, a value of 3 at the index of 2 means the third fret on the second string, or the C string, should be pushed down. If two or more notes are found on one string, the first one found will be kept. When the searching is done, the function *signalPin* is called to set the correct pins to high. This function sets the pins connected to the solenoids on the same string to the correct voltage level based on a table called *S_table*. This table contains information about which pins to set high or low based on the fret location. Appendix F contains a flowchart showing the main code for this device and explains the algorithm that processes the input to produce an output.

3.3.3 Output Hardware

Actuation Methods

In order to physically play specific notes or chords on a ukulele, the strings on the neck must be pressed down so that they touch the metal between the frets. This displacement shortens the portion of the string that vibrated and increases the frequency at which that string resonates. An increase in frequency causes that string to produce a sound that was higher in pitch when strummed. The further down the neck of the ukulele a string is pressed, the higher the pitch. In this section, methods of actuation and ways to simulate the human finger pushing on strings are investigated by the team.

Method 1: Direct Method

The first method considered in the project for pushing the strings was to have an actuator directly touch the string and physically push it down to the fret. One benefit to this method was that no modifications would need to be made to the ukulele. For this method to be effective, the team accounted for the amount of force it took for a string to be pushed down to the fret. Once this force was found, a solenoid could be selected.

When selecting a solenoid, a few parameters needed to be considered. The first was how much force can be exerted by the solenoid as it pushes down. Based on the force measurements that were taken (see Appendix A), the solenoid must be able to produce enough force to simulate the human hand pushing down on the string. If the solenoid did not push down with enough force, the string would not touch the fret and, when strummed, the string would not produce the expected sound. The force that the solenoid supplied was directly related to the amount of power that the solenoid needed. For example, some solenoids were rated at 24V but needed a nominal of 3W power. This meant that the solenoid could be actuated at less than 24V as long as the current was high enough to provide 3W of power. If the current was not high enough, then the solenoid would not receive the full 3W of power, which meant it did not press down with as much force.

The solenoid also had to have a small enough diameter in order to fit four across the width of the fret. A solenoid that was too large complicated the design of the stand and other methods of actuation may have had to be taken into consideration. Finding a solenoid that was small enough in diameter to fit across the width of the frets, as well as one that produced enough force, was needed in order for the direct method to work properly.

Method 2: Pull Method

Another method to get the strings to touch the frets was the pull method. This was a unique method in the sense that it required the ukulele to undergo some physical changes. The pull method used a pull solenoid rather than a push solenoid. Using a pull solenoid meant that when current ran through the coil, it caused a plunger to retract back into the coil as opposed to push out.

For the pull method to be successful, holes would be needed to be drilled into the neck of the ukulele, primarily near the two middle strings. The plunger of the solenoid must be lined up directly with the holes that were drilled into the ukulele. Each plunger would have to be attached to a hook that hovered over its own respective string. When the solenoid was actuated, the plunger would pull back, causing the hook to grab onto the string and pull it down. The two strings on the outside can utilize the same method, but holes in the ukulele were not needed. Instead, the solenoids would have to be placed just outside the ukulele and the hooks that pulled down still accomplished the same task.

The size of these solenoids also needed to be taken into account for the two inside strings, but the solenoids used to pull down the two outside strings were not required to be as small. The solenoids also needed to have a holding force that was large enough to keep the string pulled down on the fret. Similar to the direct method, if the force was not large enough, the string would not be held on the fret and would produce a different sound than what was expected.

Method 3: Lever Method

The third method that the team considered was the lever method. For this idea, push solenoids were used to push up on one end of a mechanical lever. The upward motion on one side caused a downward motion for the other end. This downward motion caused the lever to push down on the appropriate strings. Unlike the first two methods, the lever method did not have to use small solenoids. The solenoids could be placed on the side of the frets. As long as the levers were long enough and were corresponding to the correct string, the solenoids could have been placed anywhere in the design. One drawback to this method was that it would have been difficult to find levers that had enough force when the solenoid acted on it to fully push the string down to the fret.

All three methods were assessed and it was concluded that the first method, the direct method, was the easiest to implement for the design.

Actuation Options

After the direct method was selected, the team needed to decide how they wanted to proceed with pressing down on the strings. The team examined two different actuation options: a rack and pinion driven by a DC motor and a push solenoid.

Rack and Pinion

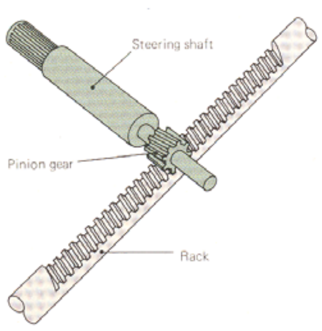


Figure 44: Rack and Pinion
[Lakshmi, 2008]

The rack and pinion option, shown in Figure 44, required the use of a DC or stepper motor in order to displace the rack enough to be on or off of the string. The possibility of using four racks per fret (one per string) and driving a DC or stepper motor driver circuit with the output of the microcontroller to depress the string down to the proper fret was investigated. The rack and pinion option required extensive knowledge of DC or stepper motors, which no one on the team possessed. Not only was the lack of knowledge on mechanical parts such as motors, but the use of motors was also a concern. The team did not want the motors to constantly be buzzing or making noise as the user played the device, altering the user's overall playing experience. Because of the aforementioned concerns, the team felt as though it was too complicated to implement the rack and pinion option and, instead, opted for a push solenoid.

Push Solenoid

As previously mentioned in the actuation method section, a push solenoid simply pushes a plunger away from the body of the solenoid when enough current flows through it. A push solenoid from Sparkfun was purchased for testing purposes. The team wanted to know if a push solenoid had enough force to simulate a human hand and press a string down onto a fret of a ukulele. A wooden test fixture was created, as show in in Figure 45, and testing proved to be a success.

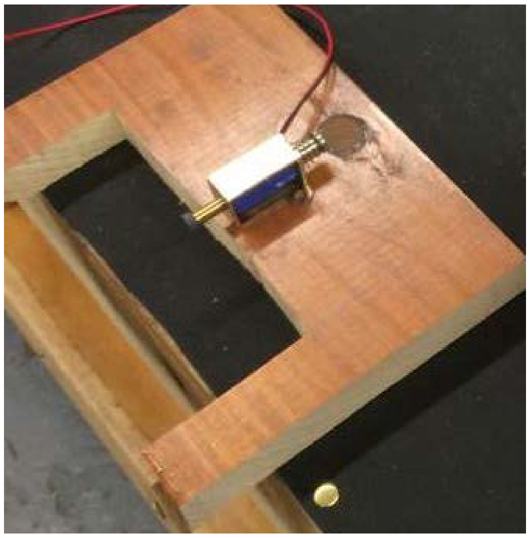


Figure 45: Wooden Test Fixture with Box Solenoid

The use of push solenoids was desired for a few reasons. The first reason was that, based on preliminary testing with the push solenoid, it had enough force to press a string down to the fret. The team also considered the amount of time given for the project and determined that push solenoids were the most time effective option due to their ease of implementation.

The TSP-45 push type solenoid from a company called Electro Mechanisms Inc. was chosen as a means to actuate on a ukulele string. The solenoids were rated for 24V and had a nominal power of 3W. The current needed to fully actuate one of these solenoids is approximately 125mA. They had a tubular body with a diameter of 0.5" with a smaller, threaded diameter of 0.25" closer to the plunger of the solenoid. The closer the plunger was to its target position, the more force the solenoid was able to actuate with. This solenoid can be seen in Figure 46.

Figure 46: 0.5'', 24V Solenoid
[Electro Mechanisms Inc, 2014]

Switching Circuitry

After finding the best method for actuating the strings, the team needed to come up with analog circuitry to connect the output of the microcontroller to the solenoids. It was decided that MOSFET switching circuits with an LED display would be used to show which solenoids are actuated at any given time. Figure 47 shows a schematic of one of the 16 switching circuits used to actuate the solenoids. The solenoids are represented by the headers in parallel with the diodes and the LEDs, as the leads of the solenoids were placed in the headers.

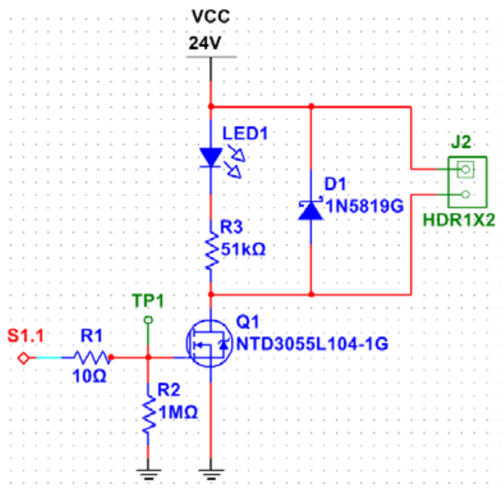


Figure 47: Switching Circuitry

First, the team had to find an appropriate MOSFET (Q1) that would work for the application. The MOSFET needed to be able to handle high voltage and high current because of the solenoid selection. The MOSFET also needed a gate threshold voltage less than 3.3V because the microcontroller outputted a 3.3V signal. To meet these requirements, the NTD3055L104-1G N-Channel Power MOSFET was selected from ON Semiconductor. This MOSFET was able to handle high currents and voltages while having a gate threshold voltage of 2.5V which is lower than the 3.3V signal that the microcontroller outputs. The source of the MOSFET was connected directly to analog ground and the drain was connected to multiple parts, including the solenoid (represented by the header in Figure 47), that were all connected to a 24V battery supply.

The two resistors (R1 and R2) before the gate of the MOSFET were important to the circuit. The 10Ω resistor limited the current going to the MOSFET when it was turned on. This resistor also limited any ringing that the gate may have seen. The 1MΩ resistor was placed at the gate and was connected directly to ground. The 1MΩ resistor ensured that the gate was not statically turning on and was off when there was no signal output from the microcontroller. Without this resistor, the MOSFET may have turned on unexpectedly, causing an improper solenoid to actuate and producing a less than pleasant sound from the ukulele.

The 24V supply from the battery was connected to three components of the switching circuit. The most important of these components was the solenoid. When the MOSFET was turned on, the solenoid presses down on the string of the ukulele that it corresponds with. The green LED from Cree Inc., LED1 in Figure 47, was an added feature to this design to assist with the user-friendliness of the device. When the solenoid actuated, the LED that was in parallel with the solenoid illuminated. The LED signified which solenoids were actuated at any given time. The placement of the LED also assisted with any troubleshooting if something were to go wrong with the switching circuit as a whole.

The Schottky diode, D1 in Figure 47, in parallel with the solenoid and the LED was a 1N5819 from Fairchild Semiconductor. It was reverse biased when the MOSFET turned on, but forward biased when the MOSFET was turned off. This specific diode was chosen because it had a fast switching speed, which was needed in this application. It also had the capability of handling high surge currents of up to 1A, which exceeded the amount of possible surge current necessary (125mA).

With all of these components put together, the switching circuit played a vital role in the final product. Without the switching circuit, the solenoids did not have a controlled way to actuate. This circuit was replicated 16 times, one for each solenoid, in order to ensure that each solenoid had the ability to be individually actuated when the user desired.

After the completion of the switching circuitry, the team had a full schematic design of the circuitry that was required to make the device work properly. See Appendix E for a full schematic overview.

Mechanical Hardware

In addition to electrical hardware like the analog circuits and microcontroller, the team also needed to include some mechanical hardware. This mechanical hardware, such as a board for the ukulele to be placed on and a solenoid stand to hold all 16 solenoids above their respective strings, was something that was outside of the scope of some of the knowledge of the team. However, with the help of many people, the team found a way to introduce some mechanical technology into the project.

Ukulele Board

When testing the push solenoid on the neck of the ukulele, the team observed that the ukulele as a whole was not staying in place. To solve this problem, the ukulele was secured to a wooden board.

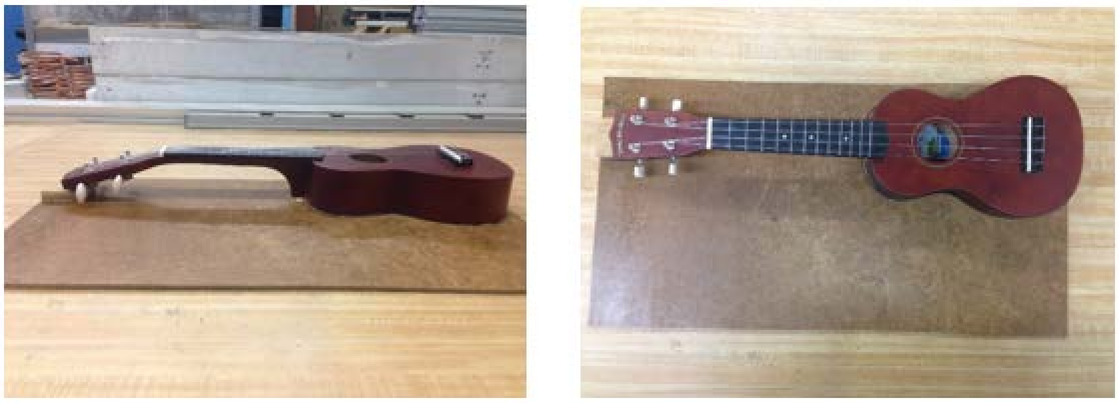


Figure 48: Side and Top View of the Ukulele Board

In order for the neck of the ukulele to be level, a 2” by 2” wooden block was placed at the furthest end from the body, or the headstock. This wooden block raised the neck of the ukulele to roughly the same height as the body. Both the ukulele and the wooden block were glued to the wooden mount using wood glue. After the team did this, they were able to properly test the push solenoid on the ukulele without the ukulele moving around. The ukulele board also provided a solid base for the stand that houses the solenoids.

Solenoid Stand

After selecting the 0.5” diameter solenoids, the team needed to find a way to house 16 of the 0.5” solenoids into the first four frets of the ukulele. The team originally came up with two possible designs for a solenoid stand. Figure 49 shows the orientation of the ukulele for the reader's reference.

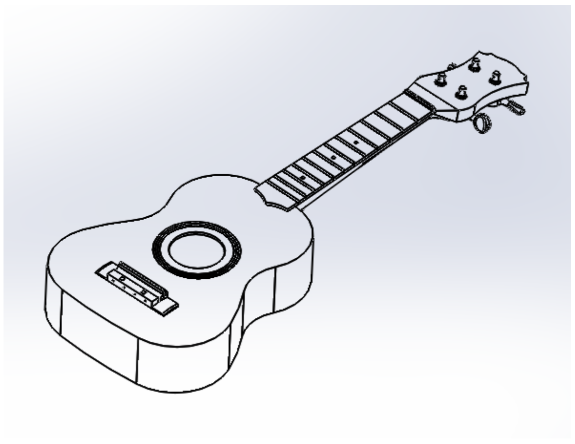


Figure 49: Orientation of the Ukulele

Design 1: Box Design

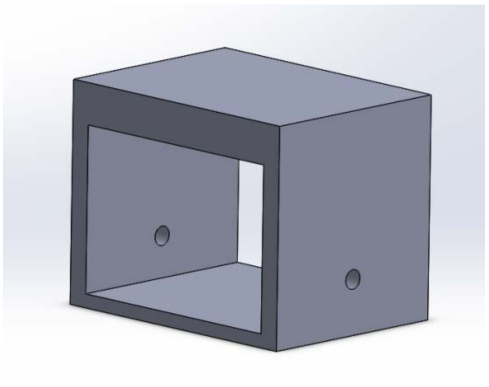


Figure 50: The Box Design

The first design, seen in Figure 50, considered for the solenoid stand was a simple box design. The box had two open sides that were designed to easily fit through the head and neck of the ukulele. With two open sides, the stand could be easily put on and taken off of the ukulele. The box was tall enough to place push solenoids above the neck of the ukulele. The box also had two holes drilled into either side that were designed to fit fasteners. These fasteners would have been used to secure the whole design to the neck of the ukulele. There would have been a sticky adhesive or small hole in the side of the neck to ensure that the fasteners were put in the same spot on the ukulele every time.

Design 2: Lego Design

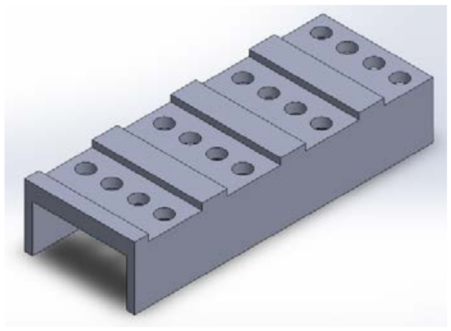


Figure 51: The Lego Design

The second possible design that the team came up with was called the “Lego” design. This design consisted of a box with holes drilled into the top. The box would be open on the bottom, allowing the device to slide right over the top of the neck. The four holes in a row, shown in Figure 51, represented areas where the plunger of the solenoids would be placed. The cutouts that are indented in the box were intended to hold each solenoid in place. This design was only compatible with the direct method as an actuation method. One of the limitations of this design was that the holes were really close together. The holes in the Lego design were slightly altered to compensate for larger diameter solenoids.

Ultimately, the team decided to combine both the box and Lego designs to create a solenoid stand that was effective with the 0.5” diameter solenoids that the team selected. With help from the technicians in the ECE and BME shops, an aluminum solenoid stand was created. It was able to hold all 16 solenoids directly above each respective string location. This allowed the direct method to occur, as the solenoids actuated directly on the strings.

To create the solenoid stand, the team first selected a scrap piece of aluminum from the ECE shop. It was a 1/8 inch thick piece of aluminum. The aluminum was cut into two equal pieces that were 3 ½ inches in length and 3 inches in width using a band saw. These dimensions were selected to accurately cover the length and width of the first four frets on the ukulele. One aluminum piece was used as the bottom of the stand and the other was used as the top.

After the team had the correctly sized aluminum, one hole was drilled into each corner of the aluminum pieces. These holes were measured to be ¼ inch by ¼ inch from the two sides of the piece that formed a corner. The holes were created using a 3/16 inch size drill bit. These holes were used to screw standoffs into the top and bottom pieces in order to hold them together. The holes on the bottom of the bottom piece were countersunk. This allowed a flat head screw to sit flush against the aluminum so that there are no exposed surfaces that catch on anything. It also ensures that the stand itself is level.

After the screw holes were drilled, the distance from the top piece to the bottom piece needed to be determined. This distance needed to be accurate because the solenoids must be high enough above each string where the solenoid would almost bottom out. The team determined that right before the solenoid bottomed out was where it would push with the maximum amount of force. If the solenoid was too close to the string, it would not push down with enough force and, when strummed, the string would sound muffled. Standoffs were used to different heights to test which height produced the best sound when the solenoid was actuated. The height of the standoffs, which were female on both ends, was 2 7/8 of an inch. The standoffs were then screwed into the bottom piece of aluminum with flat head screws.

The next task in creating the solenoid stand was to find exactly where the solenoids needed to be placed. The team used a technique referred to as the “staggering technique.” For this technique, the radius of the threaded part of the solenoid (1/4 inch) was measured to ensure the solenoids could fit close to each other. Then, one solenoid was placed on the fret and another was placed approximately halfway up the next string. This technique was used because the diameter of the body of the solenoid was larger than the desired diameter to fit all solenoids across the fret of the ukulele. In order to ensure that all of the bodies of the solenoids fit, this technique was used.

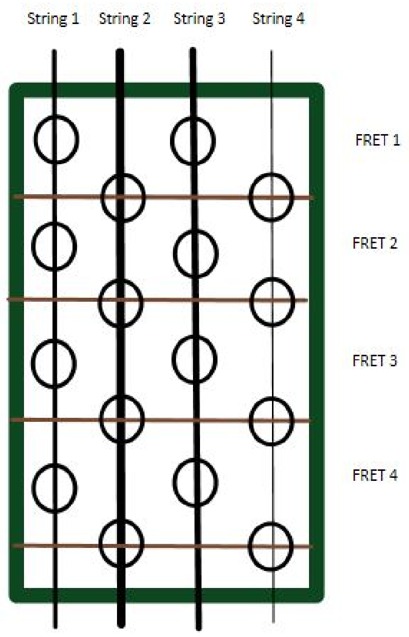


Figure 52: The Staggering Method

In Figure 52, the strings are drawn with accurate thickness to the strings on the ukulele and the circles represent where the holes for each solenoid was placed in the aluminum piece suspended above the ukulele neck. The solenoids were placed directly over the fret of the thickest string (string 2) because the thickest string required the most force in order for the string to touch the fret. If the string was over the fret, the distance required to push the string to the fret was less, thus requiring less force. Because string 2 needed to be on top of the fret, the other holes fell into place based on the staggering technique that was previously mentioned.

Using tape, a permanent marker, and white out, the team created a method to accurately drill the holes in the top aluminum piece. First, the first five frets of the ukulele were covered with clear packaging tape. Then, the frets, the strings, and the neck of the ukulele were outlined on the tape with a permanent marker. The marks gave the team a true understanding of where each string was in relation to the rest of the ukulele. Next, white out was used to cover the tip of the plunger on two solenoids. The solenoids were then positioned so their bodies were not interfering with each other and manually pushed the plungers down on the strings. The plungers left a white mark on taped strings, showing where they would be hitting. This process was repeated until there were 16 white marks on the tape.

After the marks were on the tape, the tape was taken off the ukulele and transferred onto the top piece of aluminum. The marked tape provided 16 locations where the plunger would hit the strings in the appropriate spots. A center punch tool was then used to create dents where the white marks were. These dents helped with the visualization of where the holes should go, as well as marking the point for when the drill press was used.

A drill press machine was used to drill all 16 holes where the dents were made. Using the drill press was faster and more precise than using a handheld drill. Smaller drill bit sizes were needed at first. Using them first created a small hole, which allowed the ¼ inch drill bit to easily drill a hole into the aluminum. This process was done for all 16 holes in the top aluminum piece.

The top of the solenoid stand was then secured to the standoffs that are screwed into the bottom aluminum piece. The overall solenoid stand is a box shape with four standoffs at each corner. It holds all 16 solenoids and when they are actuated, they press down on the appropriate strings with the appropriate amount of force. Figure 53 shows the side and top view of the solenoid stand.

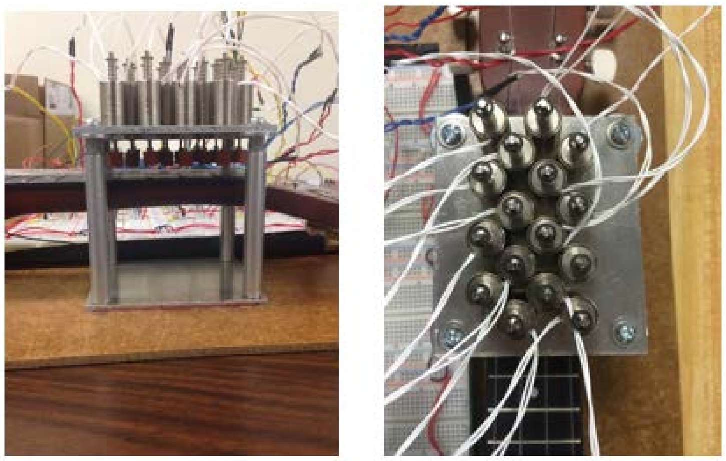


Figure 53: Side and Top View of the Solenoid Stand

3.3.4 Additional Features

In addition to the team building the design that used the input hardware, microcontroller, and output hardware, they also came up with a couple of additional features. These additional features are not absolutely necessary to the implementation of the design, but they help the design become more ergonomic and created a more user-friendly feel to the device.

Key Select Circuitry

Based on the software, the team felt as though it would be beneficial for the user to have the ability to select the key that they are in. This gave the user more control over what chords they were able to play. The team felt as though it made the device more ergonomic for the user.

The team implemented a two button design. Tactile switch buttons were used instead of FSRs for the key selection as these buttons would not be used as much as the FSRs. Once the key is selected, the user should stay in that key until they are done playing a song in that chosen key. For these reasons, a threshold was not required, so FSRs were not used.

When the first button was pressed, the key would increase a halftone. For example, a C Major Key would then change to a C# Major Key, then to D Major Key, etc. When the second button was pushed, it would determine if the key was major or minor. For example, the C Major Key became C Minor, C# Major became C# Minor, etc. The user knew what key he/she was in by seeing LEDs shown in Figure 54

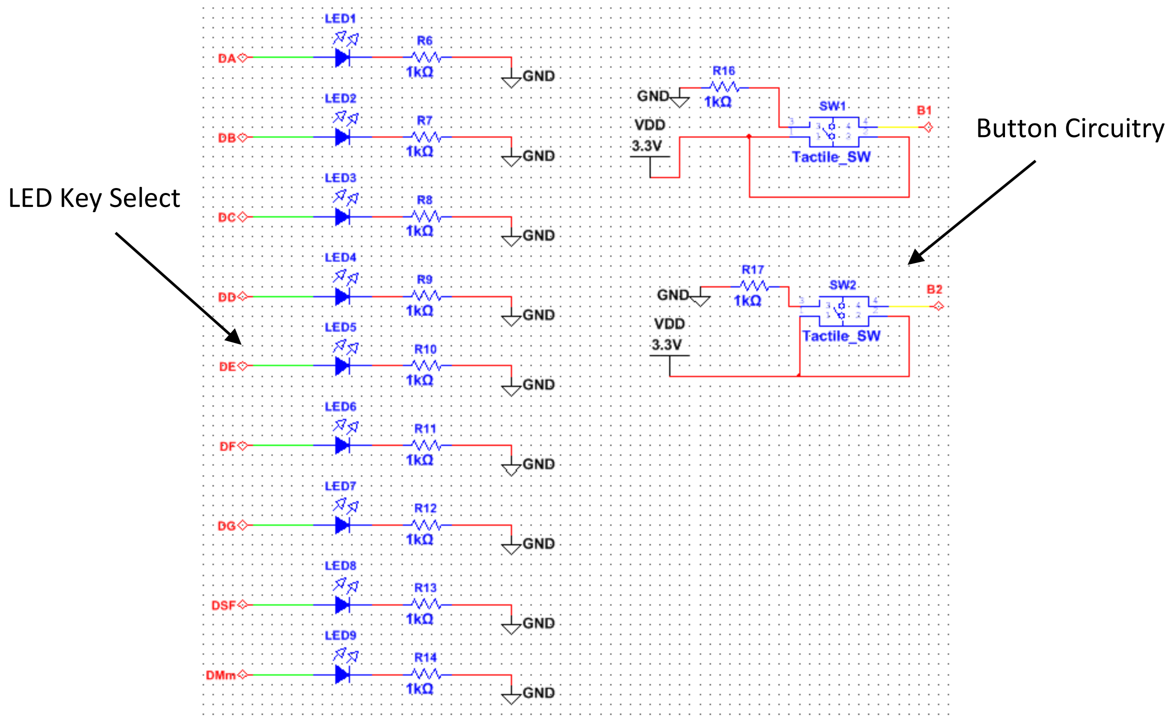


Figure 54: Key Select Circuitry

The data provided in Tables 6 and 7 represent each key and what chords are in the keys. These tables were important to understand because it helped the user select which key they would like to be in based on the chords that they required.

Printed Circuit Board Implementation

Although the team successfully designed the analog circuitry on breadboards, the team felt that the implementation of printed circuit boards, or PCBs, would benefit the user for a number of reasons. First, the user would not have to worry about wires falling out of the breadboards, as traces for these wires were placed within the PCB. The traces also limit the possibility of two wires accidentally touching and shorting the circuit, causing the circuitry to not work properly. The use of PCBs also confined the circuitry into a smaller area. The smaller circuitry allowed the ukulele to be more portable.

The team decided to design two PCBs: one for the input hardware, microcontroller, and key select circuitry and the other for the solenoid switching circuits. This decision was made because the input hardware ran mostly off of the 3.3V regulator while the output hardware was supplied with the 24V power supply. The two boards were connected with jumper wires, where the output signal from the microcontroller was transferred from one board to another. The 3.3V board was called PCB #1 and the 24V board was called PCB #2.

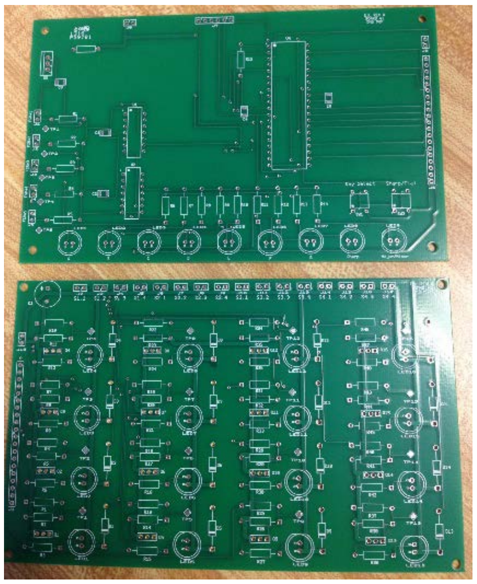


Figure 55: PCB #1 and #2

The first part of the PCB design process began with an accurate schematic drawing of the circuits. The team used Multisim to create schematics of the circuits mentioned in this section. After ensuring that all traces were properly wired to the appropriate parts, the team annotated the Multisim designs into Ultiboard, a PCB layout software. In Ultiboard, the team moved PCB footprints around a 6.5” x 4” board, placing parts that are connected close to each other. The team used this technique to avoid having long traces move across the PCB. Placing the connected parts close to each other also assisted with troubleshooting the PCBs.

The team set up connectors for the boards strategically. PCB #1 (shown in Figure 56) had eight connectors on it. The left edge contained five 2-pin headers. These headers were used to connect the FSRs to the FSR circuitry. They were placed on the left side because the user would be using the clay mold on his left hand. The 2-pin header on the top of the board was used to bring the 24V supply into the board. Even though PCB #1 was thought of as the low voltage board, the 24V power supply still needed to be stepped down by the regulator. The 2-pin header closest to the top right corner of the board is also the 24V supply. However, this header was used to transfer the power supply from PCB #1 to PCB #2 (shown in Figure 57). Finally, the 20-pin header on the right edge was used to transfer the output signals of the microcontroller from PCB #1 to PCB #2. The headers on the right were used to transfer signals and power because the team planned on PCB #2 being located to the right of PCB #1 and, to ensure a shorter connection, the connectors were placed as close to PCB #2 as possible. The key select circuitry was placed at the bottom of PCB #1 so the user would have easy access to the two tactile buttons. Each LED was labeled with its appropriate root name (A, B, C, etc.), as well as sharp and major.

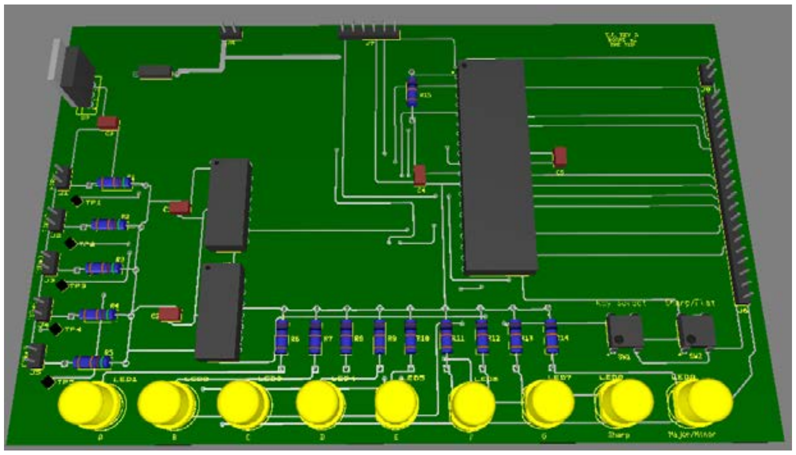


Figure 56: PCB #1 Ultiboard 3D View

For PCB #2, the team placed the 20-pin header on the left edge. The thought process behind PCB #2 was similar to that of the 20-pin header on the right edge of PCB #1. In addition to the 20-pin header placed on the left edge, PCB #2 also had a 2-pin connector for receiving the power supply from PCB #1. Along the top edge, 16 2-pin headers were placed. These connectors corresponded to a switching circuit. By plugging a solenoid into each connector, the solenoids were placed in parallel with the output of the switching circuits. The LEDs were placed on the board in a specific order. The LEDs on the bottom of PCB #2 correspond to the string closest to the user. Therefore, the user was able to easily understand which solenoids were being pressed just by looking at which LEDs were illuminated.

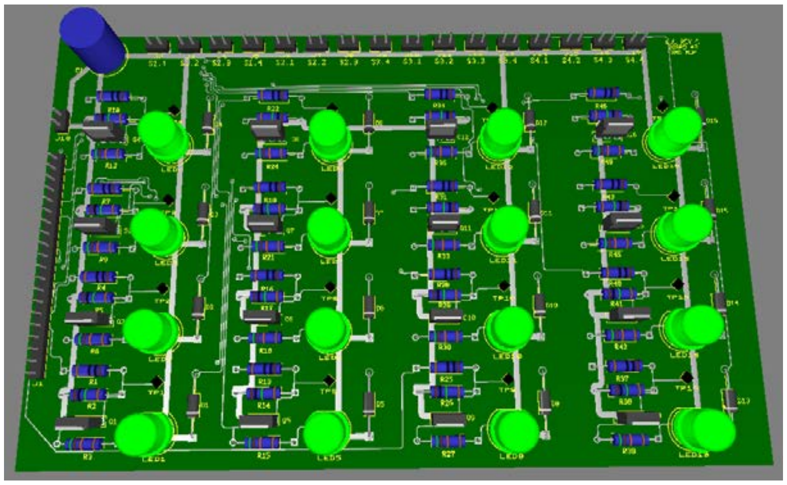


Figure 57: PCB #2 Ultiboard 3D View

Overall, the implementation of PCBs assisted the user in understanding what the device is doing. It also allowed the design to be safer with no fear of wires falling out or touching to short out the circuits.

3.4 Summary

This chapter described the steps taken by the team to create the device for the user. The team decided take a modular approach to creating the device, splitting it into three main parts (i.e. the input hardware, microcontroller and software, and the output hardware). The team was also able to create additional features for the device, such as a key select display. Finally, in order to package everything for the user, the team designed two PCBs that contained all the circuitry for the various parts of the device.

4. Implementation and Results

In order to perform a complete testing of the device, the parts of the device were tested in stages. Testing in stages allowed for easier debugging and gave the team an opportunity to observe which parts of the device needed more work. Next, the quality (i.e., frequency) of the musical notes produced using solenoids was tested and compared to the frequencies when pressing a string with a finger as well as a theoretical frequency for that musical specific musical note. In addition, several chords were tested to gauge for sound quality. Finally, the team tested and received feedback on the device from the subject.

4.1 Final Design

Figures 58 and 59 show an overall block diagram of the system and a picture of the setup of the device, respectively.

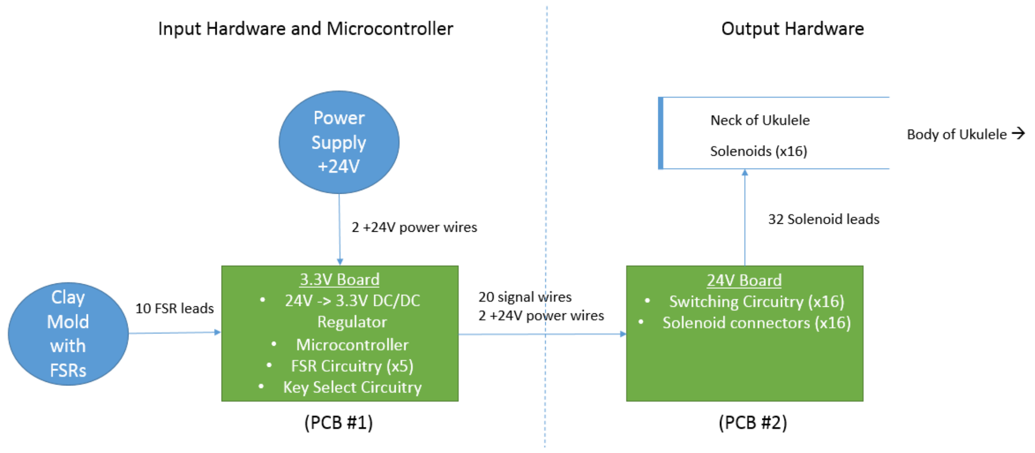


Figure 58: General Architecture of Final Design

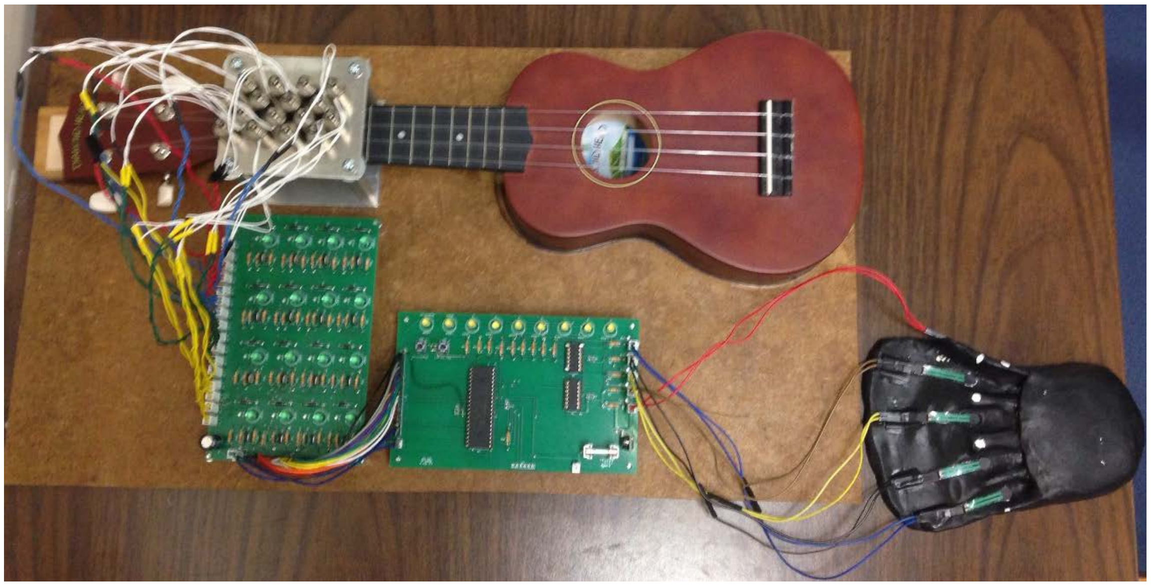


Figure 59: Full Device Layout

The device was separated into three parts: the hand input (black clay mold with FSRs), the electronics (two PCBs), and the solenoids with the ukulele. To use the device, the FSRs must be pressed. They can be pressed one at a time or in a combination. The input received from one or more FSRs will then be processed by a microcontroller. The microcontroller compared the voltage from the ADC to a threshold set by the user. This threshold was set based on the finger force capabilities of the user. If the voltage from the ADC was above the threshold, the microcontroller used an algorithm to correlate the user input to a corresponding chord. The microcontroller then outputted a signal to the correct switching circuits, actuating the solenoids and lighting the LEDs.

Figure 58 also shows the two PCBs that contain the electronics for the device. An advantage to having two PCBs was that it gave the team a bit more freedom to move the various parts of the device around. For example, PCB #2 contains the switching circuitry and as a result, was placed closer to the solenoids on the ukulele, whereas PCB #1 held the FSR circuitry, allowing for its placement to be near the clay mold. Using two PCBs allowed for a physical separation between the user interface and the output hardware, thus improving the safety of the device.

4.2 Team Testing

4.2.1 Power

The first part of the overall circuitry that the team tested was the power. It was important that the 3.3V regulator the team selected worked properly. The specifications for the voltage regulator stated that it functioned properly with input voltages from 7V-36V. The team inserted the regulator into a breadboard and supplied the input with 7V. With an oscilloscope reading the output voltage, the team observed that the regulator outputted a constant 3.3V. The team slowly increased the input voltage to 24V and noticed that the regulator outputted a constant 3.3V. This test proved that the voltage regulator worked properly. Thus, the team was able to move forward with testing the rest of the circuitry with the confidence that all parts were receiving the appropriate voltage levels.

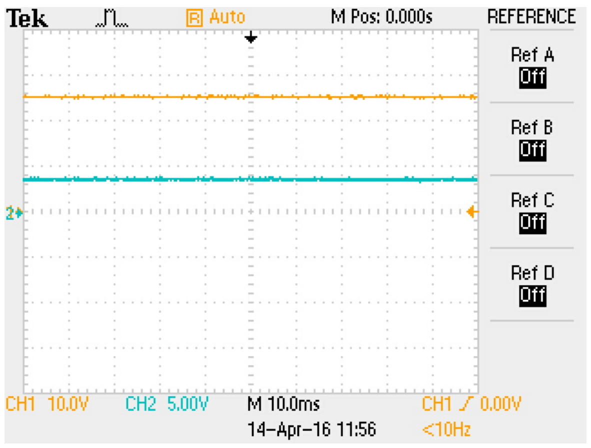


Figure 60: 24V Input (yellow) and 3.3V Output (blue)

Figure 60 shows an image of the oscilloscope. Channel 1, the yellow signal, had a voltage scale of 10V/div while Channel 2, the blue signal, was set to 5V/div. The time scale was set for 10ms/div for both channels. This oscilloscope reading showed the input signal into the voltage regulator (approximately 24V) and a constant output voltage (3.3V).

4.2.2 FSRs on the Clay Mold

The team found it important to test the FSRs on the clay mold. The team used epoxy to attach the FSRs to the clay mold in order to have a permanent solution (rather than having to use tape to adhere the FSRs onto the clay every time tests needed to be conducted). Therefore, it was important to understand the effects the epoxy would have on the FSRs and the clay mold, if any.

The team used five-minute epoxy and, after mixing equal parts resin and hardener, applied the mixture to the front of the body of the FSR. The epoxy was applied in this manner so the FSRs could be placed on the clay mold with the sensor side down, allowing for better protection of the FSR (the team found that the non-sensor side of the FSR was more durable, and decided that for long-term use, the non-sensor side of the FSR would last longer after repeated use).

The team had some difficulty attaching an FSR to the part of the clay mold that would cater to the user's thumb. Since the thumb FSR was on the side of the clay mold (as opposed to lying flat like the other fingers, as shown in Figure 40 on page 60), it was more difficult to epoxy. However, the team attached the thumb FSR and taped it down for approximately 24 hours until the FSR was freestanding with only epoxy.

The team tested and viewed the output of the FSR circuit on its own in order to observe the output waveform into the microcontroller. Figure 61 shows the results of this test. The time scale on the oscilloscope reading was 1second/div, allowing the team to hold an FSR down for different periods of time and see the output of the buffer. The voltage scale was set to 1V/div. Each peak represented the period of time when the FSRs were pressed. The peaks were around 3V, which was above the set threshold for the microcontroller.

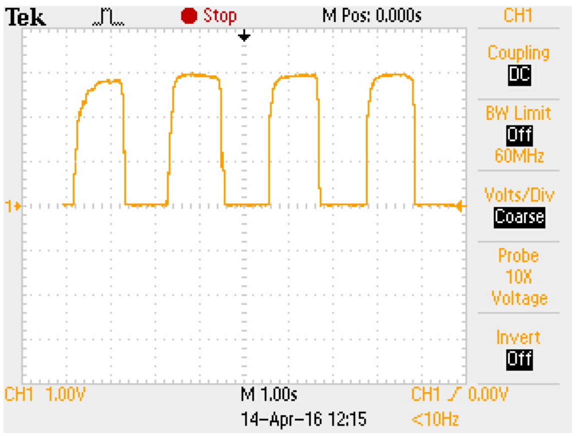


Figure 61: Buffer Output when FSR Pressed

4.2.3 Analog-to-Digital Converter

The microcontroller used five analog-to-digital converter (ADC) channels to read the inputs coming from the FSRs. Based on the digital values that are produced by the ADC and recorded into the microcontroller, corresponding outputs were set high (as close to the input voltage as possible) to actuate the solenoids. All testing with the microcontroller was done using the PICDEM Lab II development board with the PICkit 3 debugger tool. These tools helped the team integrate all the necessary components on one testing platform. The team tested the FSRs in three different ways using the specified tools.

The first test involved the use of serial communication with the USART. When an ADC value was captured in the code, it was stored in a buffer. This ADC data was then transmitted to the computer to be viewed by the user. This ADC data was useful to the user, as he could be able to make sure that the FSRs were functioning properly with it.

This USART communication was also used to make sure that the ADC was function properly. By inputting a sine wave into the input port that the FSR was connected to, the ADC collected individual samples from the sine wave and then streamed to the computer using the USART. The data was taken from the computer and inputted into an Excel sheet to make sure the input was the same as the output. Figure 62 shows the data plotted to create a sine wave. The sine wave had a frequency of about 1 Hz.

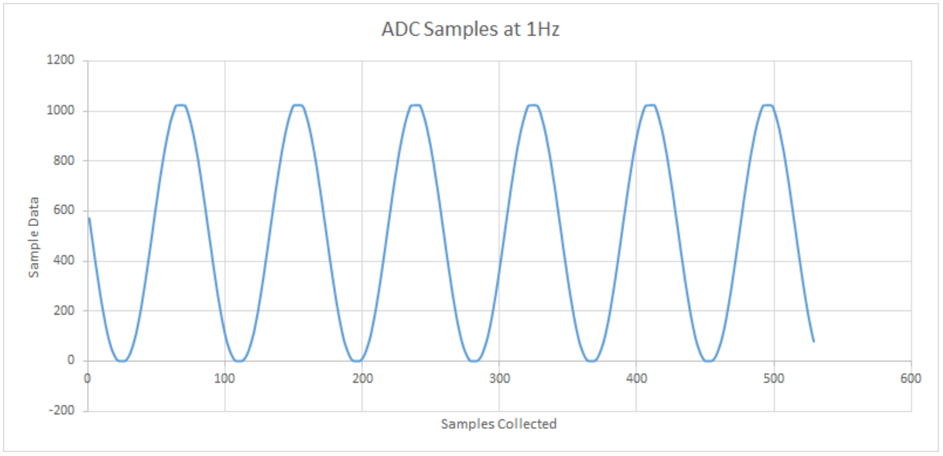


Figure 62: Sine Wave Test Results for ADC Channels

It is important to note that the ADC channels are not sampling periodically. However, the ADC was sampling at a rate that was fast enough (measured at about 450 Hz) for its application in the device being created. Once a sine wave was put into the microcontroller and the same sine wave was outputted, the team was able to confirm that the ADC was producing an output. Testing the ADC also allowed the team to confirm that the FSRs were being read properly.

4.2.4 FSRs with the Microcontroller

The second testing method involving the microcontroller was connecting the FSRs to five analog pins of the microcontroller and reading the outputs on an oscilloscope. Based on the team's code, the FSRs would only allow for an output to turn high when a certain threshold was met. This threshold was determined by reading into the debugger console for the digital values of different forces applied to the FSR, and was equal to approximately 1.6V. Once a force that the team felt was light enough for the subject to use was determined, the threshold was set in the code so that it could be used for testing. At first, the code was programmed in such a way that one FSR would only correspond to one output. An oscilloscope was used to make sure that the output was set high when the FSR was pressed and surpassed the threshold, as shown in Figure 63. The yellow waveform shows the input to the microcontroller from the FSRs and the blue waveform shows the output of the microcontroller.

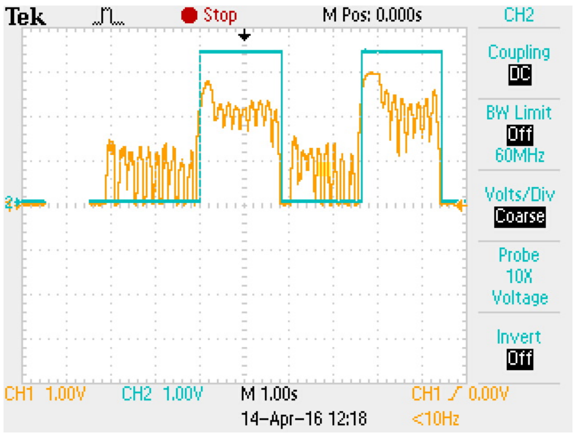


Figure 63: Buffer Output (yellow) and Microcontroller Output (blue)

Once the code was further developed, the team could check up to four outputs using four channels on the oscilloscope. At this point, one FSR was able to correspond to up to four solenoids.

Once the team was able to see four outputs go high using the oscilloscope, the team tested the next part of the design: the switching circuitry.

4.2.5 Switching Circuit

The next part of the circuitry that the team tested was the switching circuitry. The switching circuitry was isolated by using a function generator as an input and the dual power supply to provide 24V to the circuit. First, the team set up the function generator in the lab to output a 3.3V square wave with a period of five seconds. The square wave had an offset of 1.65V to produce a square wave from 0V to 3.3V. The function generator was set up to simulate the output of the microcontroller and can be seen in Figure 64.

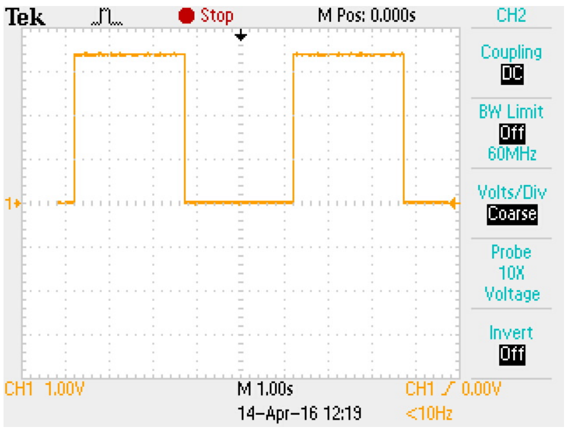


Figure 64: Input to Gate from Function Generator

When the function generator produced a signal of 3.3V, the MOSFET turned “on”. Current was then able to flow through the LED, illuminating it. The first few tests that were conducted were done so without the solenoids wired in parallel with the LED and the diode. Therefore, the LED acted as an indicator, where a lit LED symbolized an actuated circuit. Once the team observed the appropriate response from the LEDs, they introduced the solenoid in parallel with that LED. When the LED lit up and the solenoid actuated, the team was confident that the circuit was correct and implemented it into the rest of the design.

4.2.6 Key Select

The team needed a way to cycle through each of the 24 musical keys. Therefore, the team designed an LED display controlled by mechanical push buttons in order for the user to select which key they would like to play in.

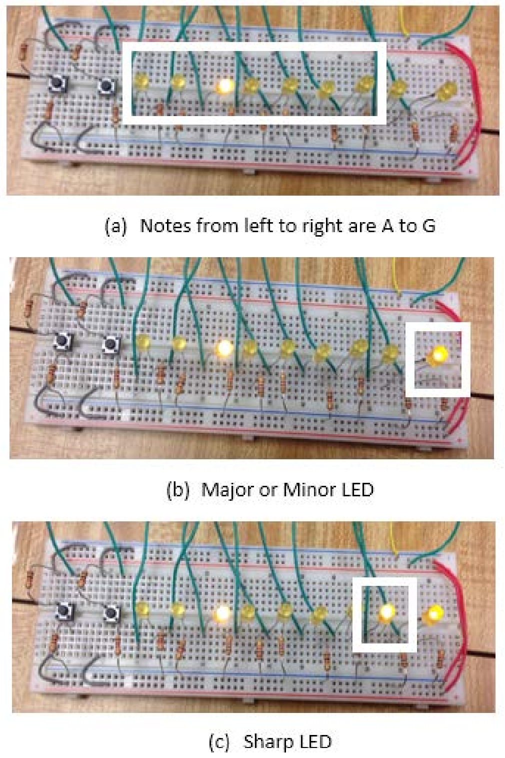


Figure 65: Key Select LED display

There were two buttons used to select a key. The first button selected the root note of the key (A, B, C, D, E, F, or G). The root note button also chose if the key was sharp. By default, the first root was C. If the root note button was pressed, the next key would be C#, then D, then D#, etc. The other button simply selected whether the key was major or minor.

To check if the buttons were working properly, the team included key select LEDs into the design. The LEDs were placed and labeled accordingly to accurately represent the key that the user was playing in. The first seven LEDs indicated the possible roots that could be played. The eighth LED represented whether the root was sharp and the ninth LED showed if it was a major or minor key. By default, the first key was C Major. Figure 65 shows the LED display of C Minor (a), C Major (b), and C# Major (c) keys respectively.

4.2.7 All Circuitry Together

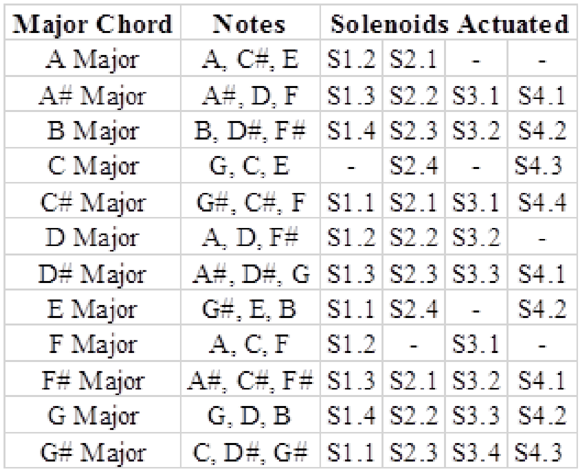
Up to this point, the team had successfully tested all of the circuitry needed for a prototype. These tests assured the team that all the circuits were working properly and that integrating them together would create a full working prototype. Similar to the switching circuit tests, the first iteration of testing was done without the solenoids, using the LEDs in the switching circuit as indicators.

The team connected all of the necessary parts. Each MOSFET gate was connected to its appropriate output pin from the microcontroller, allowing each switching circuit to have the ability to turn on based on the microcontroller outputs. Using an oscilloscope, the team observed the output voltage from the regulator (see Figure 60 on page 95), the output of the buffer circuits when an FSR was pressed (see **Error! Reference source not found.**61 on page 96), and the output signal from the microcontroller (see **Error! Reference source not found.**63 on page 98) to determine if they were all the correct outputs.

Once all of the electronics were powered and working appropriately, the key select LEDs turned on, showing the key C Major. The team then pressed the thumb FSR and observed LEDs at the output of the switching circuit turn on. These LEDs represented the chord C Major. The team tested each FSR and examined which LEDs were turning on, symbolizing an actuated solenoid. The team then used the key select buttons to cycle through each possible key and tested all of the chords in each key.

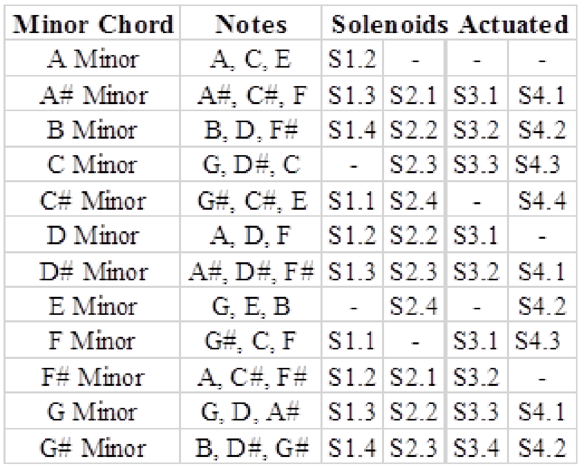
The team used a specific nomenclature to represent each position that a solenoid could actuate on. The S represents a solenoid being actuated. The first number refers to the string, with string 1 being the G string, string 2 being the C string, etc. The second number corresponds with the fret on that specific string. As an example, the phrase S2.4 represents the solenoid on the fourth fret of the second string. Table 8, Table 9, and Table 10 represent each major, minor, and diminished chord that the team tested with the entire circuitry and which solenoids they actuate. The dashes in the tables meant that no solenoids were actuated on that particular string.

Table 8: Major chords and their actuated solenoids



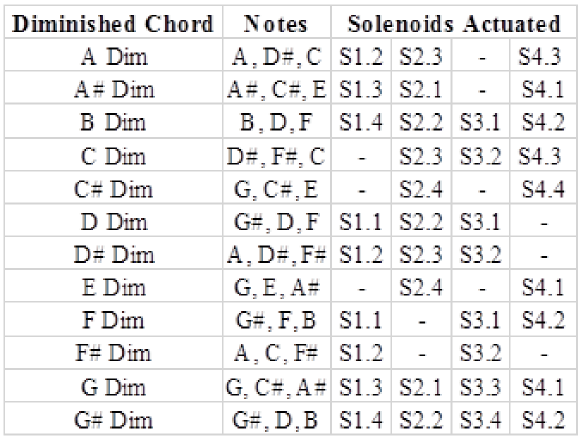
Major Chord	Notes	Solenoids Actuated			
A Major	A, C#, E	S1.2	S2.1	-	-
A# Major	A#, D, F	S1.3	S2.2	S3.1	S4.1
B Major	B, D#, F#	S1.4	S2.3	S3.2	S4.2
C Major	G, C, E	-	S2.4	-	S4.3
C# Major	G#, C#, F	S1.1	S2.1	S3.1	S4.4
D Major	A, D, F#	S1.2	S2.2	S3.2	-
D# Major	A#, D#, G	S1.3	S2.3	S3.3	S4.1
E Major	G#, E, B	S1.1	S2.4	-	S4.2
F Major	A, C, F	S1.2	-	S3.1	-
F# Major	A#, C#, F#	S1.3	S2.1	S3.2	S4.1
G Major	G, D, B	S1.4	S2.2	S3.3	S4.2
G# Major	C, D#, G#	S1.1	S2.3	S3.4	S4.3

Table 9: Minor chords and their actuated solenoids



Minor Chord	Notes	Solenoids Actuated			
A Minor	A, C, E	S1.2	-	-	-
A# Minor	A#, C#, F	S1.3	S2.1	S3.1	S4.1
B Minor	B, D, F#	S1.4	S2.2	S3.2	S4.2
C Minor	G, D#, C	-	S2.3	S3.3	S4.3
C# Minor	G#, C#, E	S1.1	S2.4	-	S4.4
D Minor	A, D, F	S1.2	S2.2	S3.1	-
D# Minor	A#, D#, F#	S1.3	S2.3	S3.2	S4.1
E Minor	G, E, B	-	S2.4	-	S4.2
F Minor	G#, C, F	S1.1	-	S3.1	S4.3
F# Minor	A, C#, F#	S1.2	S2.1	S3.2	-
G Minor	G, D, A#	S1.3	S2.2	S3.3	S4.1
G# Minor	B, D#, G#	S1.4	S2.3	S3.4	S4.2

Table 10: Diminished chords and their actuated solenoids



Diminished Chord	Notes	Solenoids Actuated			
A Dim	A, D#, C	S1.2	S2.3	-	S4.3
A# Dim	A#, C#, E	S1.3	S2.1	-	S4.1
B Dim	B, D, F	S1.4	S2.2	S3.1	S4.2
C Dim	D#, F#, C	-	S2.3	S3.2	S4.3
C# Dim	G, C#, E	-	S2.4	-	S4.4
D Dim	G#, D, F	S1.1	S2.2	S3.1	-
D# Dim	A, D#, F#	S1.2	S2.3	S3.2	-
E Dim	G, E, A#	-	S2.4	-	S4.1
F Dim	G#, F, B	S1.1	-	S3.1	S4.2
F# Dim	A, C, F#	S1.2	-	S3.2	-
G Dim	G, C#, A#	S1.3	S2.1	S3.3	S4.1
G# Dim	G#, D, B	S1.4	S2.2	S3.4	S4.2

4.2.8 Sound Testing

In order for the ukulele to produce a clean sound (i.e., a sound without buzzing or muting), a musician would need to push down on the strings of a fretboard with a good amount of force. Pushing a string down with a sufficient amount of force prevents the strings from buzzing as that string is strummed. A musician is capable of doing this with his fingers in several ways, depending on which string is being pressed. For example, a musician would use just the tips of his fingers in order to push down on strings in the middle, and would be able to use more than just the tips of his fingers in order to push down on the outer strings (since there is more room for his fingers to push down on those strings without affecting other strings). The solenoids in this device needed to mimic fingers pushing down on strings closely enough to produce a clean sound. Therefore, a test was completed with each solenoid to test the sound of individual notes.

Frequencies at which these sounds occurred were researched. Frequencies in music are the speed at which the strings resonate, determining the pitch of the sound. After finding the correct information, the team wanted to make sure that the sounds produced using solenoids would be comparable to the sound produced if one were to use his or her fingers. A guitar tuning application on a smartphone was used to record the sound and measure the pitch. Appendix C shows a table of these results. To summarize, the sounds produced using one's fingers were relatively close to the sound that is theoretically supposed to be produced. The frequencies found for these strings vary between the third and fourth octaves only.

Next, the team wanted to test whether the solenoids were able to produce a similar sound. Prior to testing, the team researched how well humans are able to hear differences in pitch. The results found varied depending on the amount of musical training and experience one may have. Nevertheless, it was determined that, on average, humans are able to hear differences in pitch of about 3Hz [Randall and Backus, 1970]. Therefore, the team determined whether a solenoid was pushing down on a string hard enough to produce a clean sound by looking at the frequency differences between the theoretical value researched, and the values measured by the guitar tuning app for any given note being played. Testing one solenoid at a time, the results of these tests are also shown in Appendix C.

Based on the results from the testing, all but two of the solenoids (i.e., 14 out of 16) produced the same notes as one would expect to find with those particular strings and frets. In addition, these notes were relatively close to the theoretical (within 3 Hz in terms of pitch differences). However, the two solenoids not producing the same note as the theoretical note were not pushing down with enough force. Though the sounds for these two solenoids were clean (no buzzing or muffled sound), the pitch of each of these sounds were outside of the 3 Hz range, resulting in the incorrect note.

The solenoids on fret 1 for the G string and the E string produced sounds that deviated from the theoretical for a couple of reasons. The tension of both strings at that particular placement of the solenoids was greater than anywhere else on the ukulele. Due to this placement, these solenoids required a greater amount of force in order to fully depress the string. Therefore, the team believed that the placement resulted in the solenoids needing a greater force. Because the solenoids could not provide that much force, the strings never depressed fully, causing an incorrect note to be played. Figure 66 shows the placement of the two solenoids close to the nut of the ukulele.

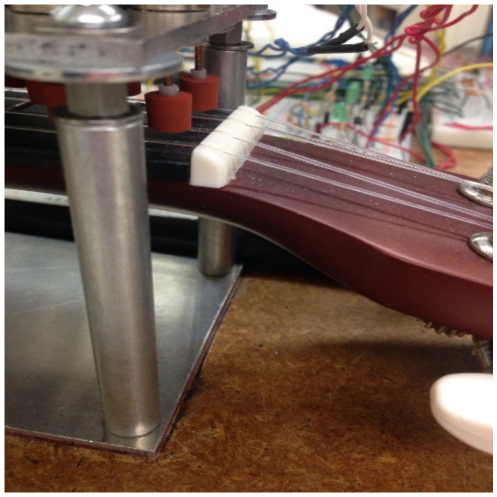


Figure 66: Two solenoids closest to the ukulele neck

In addition, although the solenoid on the fourth fret of the G string produced a correct note as well as a pitch that was close to the theoretical, it buzzed slightly as it was being strummed. This buzzing may have been due to the string not being depressed enough between the two borders that make up the fourth fret. However, the buzzing sound was barely audible, and was often masked when the note is played along with other strings.

4.3 Subject Testing

Once the team was able to produce a working prototype, it was brought to the subject for additional testing. The prototype provided the subject with a sense of how the final design would feel and what it would look like. In presenting the prototype to the subject, the team sought feedback on what could be changed or improved upon in order to make the device more comfortable for him. Particularly, the team was looking for information on the thresholding for each individual FSR, the design and comfortability of the clay mold, the most comfortable finger combinations to produce a chord, and the orientation of the ukulele in relation to his body.

When testing the thresholding for each FSR, the team had the subject press each FSR individually to test if he could exert enough force to actuate the solenoids. The force exerted on the FSR determined how high the voltage output would be. For example, if the FSR is pressed at full force, the output would reach the supply voltage rail. This output would then be connected to an ADC channel in the microcontroller that would convert the analog data into digital data.

Three constant thresholds were set in the code: a light threshold, a medium threshold, and a hard threshold. The light threshold equated to the digital value of 250, the medium threshold had the digital value of 500, and the hard threshold equaled the digital value of 750. To convert the digital value into an analog value, the resolution of the ADC (the smallest detectible change in voltage that the microcontroller can detect) and the voltage range of the microcontroller had to be known. The voltage range for signals that were measured were divided by 2^n parts where n was the number of bits the ADC works in. The team's microcontroller had a 10 bit ADC, resulting in a voltage range being divided by 1024 (2^{10}).

Converting the digital value to the analog value required knowledge of the voltage scale of the signal being measured. In the case of this project, the voltage range was 0V to 3.3V (the maximum and minimum ranges of the FSR voltage divider). These two parameters indicated that the smallest detectable change for the microcontroller with a voltage scale of 3.3V was 3.22mV. From the equation below, the analog value was calculated by simply multiplying the digital value by the smallest detectable change the microcontroller could detect. An example of converting the 500 digital value to an analog value is shown.

$$\frac{voltage\ range}{2^{n-bits\ of\ ADC}} \times Digital\ Value = Analog\ Value$$

$$\frac{3.3V - 0V}{2^{10}} \times 500 = 3.22mV \times 500 = 1.61V$$

Based on these equations, the light threshold was set at an analog value of 0.81V, the medium threshold was 1.61V, and the hard threshold equaled 2.42V.

When performing the test described above, the team asked the subject if the force he exerted was at a comfortable level or not. If the current FSR was tested at a comfortable level for him, the next FSR was then tested. If the FSR was not at a comfortable level to him, the team changed the threshold to a lower level via the USB interface. The subject believed that the medium threshold was acceptable for the index, middle, and ring fingers. However, the thumb and pinky fingers were difficult for him. To fix difficulty for the thumb, the team lowered the threshold from the medium threshold to the light threshold. When the subject tried the pinky finger at the light threshold, he still had difficulty pushing it down. However, he noted that it was more of the placement of the FSR not being in an optimal position for his hand and not necessarily the force that he needed to apply.

As briefly described above, the subject had some issues with pressing down on the FSRs due to their placement on the clay mold. Although he could press down on most of the FSRs with ease, the team observed that he had to move his hand out of the clay mold each time he wanted to press down on a different FSR. The subject then observed that even if he tried to keep his hand in the clay mold, he would need to lift up his other fingers when he tried to exert force with just one finger. He then told the team that he believed that if the FSRs can be repositioned on the mold, then this would not be as much of a problem. The team placed a white mark (see Figure 68) near the positions in which the subject thought would be the best place to put the FSR on the clay mold.

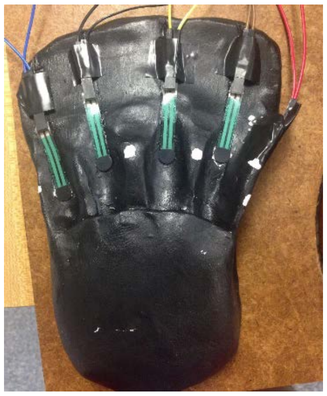


Figure 67: White Marks on Mold for FSR Placement

With the FSR placement testing completed, the team then moved on to asking the subject which finger combinations he desired to play all 8 chords for each key. As there are only 5 FSRs on the clay mold and 8 chords per key, a few FSRs needed to be pressed simultaneously so that the other three chords could be played. The original method to play the remaining three chords was to press down two FSRs at the same time in the following finger configurations: the thumb and the index finger, the thumb and the middle finger, and the thumb and the ring finger. The subject had trouble with the thumb and middle finger and the thumb and ring finger, but he found using the thumb and index finger was comfortable. He believed that the most comfortable way to play the remaining two chords was with the index and middle finger for one chord and with the index, middle, and ring finger all combined for the last chord. In accordance with what the subject suggested, these preferences were immediately implemented in the design.

The final aspect the team tested for was the orientation in which the ukulele was placed with respect to the subject. The team originally intended for the board with the ukulele mounted on it to be horizontal when facing the subject. This would allow for the subject to strum the strings by strumming away from his body. However, when placing the board near him on the table, the subject suggested that the ukulele board be placed in a vertical position facing him. This meant that the body of the ukulele was directly in front of him and he would strum left to right as opposed to away from and towards him. To better understand how the ukulele was positioned, see Figure 69.

While finishing up testing with the prototype, the subject told the team something that was previously considered. He told the team that constantly pressing down on the FSRs would probably become tiresome for him if he used the device for more than a few minutes. The device was designed in such a way that when the FSRs are pressed down, the solenoids were actuated. However, the moment the FSRs are released, the solenoids were reset to their original resting positon. The subject suggested a method that allowed him to press an FSR or FSR combination, release his finger from the FSRs and still have the solenoids actuate over the strings for a period of time. This method, which the team called the latching method, was implemented into the design in order to allow the subject to play the device for an extended period of time before becoming tired.

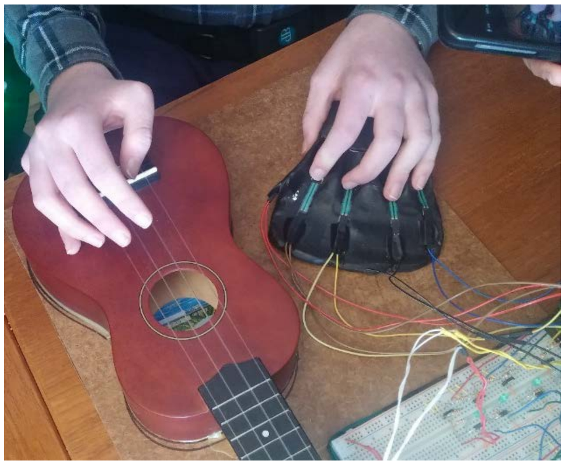


Figure 68: Preferred Orientation of Ukulele

5. Discussion

5.1 Accomplishments and Failures

Overall, the team was able to produce a working device in terms of accomplishing the objectives laid out at the start of this project. The goals the team had at the start of the project were to mimic the fingerings of a ukulele, produce clean chord sounds, create a cost effective device, use easily replaceable parts, and create a device that was easy to use. Since there are many different traditional ways of playing a chord on a ukulele, the team chose a variation of the most common fingerings. Though not all traditional, the device has a LED matrix in which each LED corresponds to a note on each fret of each string. This LED matrix allows the user to see which individual notes are being played when a chord is chosen using the FSRs on the clay controller. The LED matrix simplifies the learning curve the user may have to overcome in case the team implemented a way of playing a chord that may not be familiar to the user. The second objective the team worked towards was producing clean chord sounds. The team chose to use solenoids to depress the strings on the neck of the ukulele. This meant that in order to produce clean chord sounds, a solenoid needed to push down on a string with as much force as a finger is able to produce when performing the same task. According to the tests conducted to examine the pitch of each note, the majority of notes played using solenoids produced pitches that were comparable to pitches produced when playing with one's fingers. In addition, the device was made to be powered on and off easily, making it simple for both the subject and the caregiver to manage how the device is switched on, operated, and switched off.

While the team attempted to keep the cost of individual parts of the device low (less than ten dollars per piece for any component), the aggregate cost of the parts used in this device was much higher than the team would have liked due to the nature of the products available in the current market. A prominent example of a part needed for the device that turned out to be costly due to its very specific nature was the type of solenoid used to depress the strings of the ukulele. The solenoid needed for this device needed to be small enough so that 16 solenoids would fit well on the first four frets of the ukulele, as well as produce enough force to depress a string on the ukulele without resulting in a buzzed or muted sound when that string is strummed. Due to these limitations, the team had to look for a very specific type of solenoid (i.e. one that was small and produced enough force), narrowing the options for solenoids available to the team. As a result of the specific nature of the solenoids used in this device, the price of the solenoids was higher than expected. Purchasing sixteen solenoids was costly ($400 in total), so the team was unable to satisfy the goal of having a cost effective device. In addition, the solenoids are not easily replaceable since they were purchased from a very specific manufacturer. Thus, if one solenoid were to stop working, a user would have difficulty replacing it. The team hopes that, in the future, the market for miniature solenoids grows to support applications and devices such as the one created for this project.

The team also implemented the device well in terms of adhering to the constraints set forth by the limitations of DMD. The constraints the team kept in mind while creating this device were that the device should not overexert the user’s muscles and must be electrically safe to use. In addition, constraints were placed on the creation of the device itself to ensure that the device worked well post-implementation. The device-specific constraints set forth by the team were that any actuation method chosen by the team had to be chosen such that it would depress the strings with enough force, and that the method of actuation would effectively cover each string on each fret. The clay controller created to house the FSRs was molded to fit the shape of the subject’s hand so that the controller (a part of the device that the subject would come in direct contact with) would be comfortable to use. In addition, in order to prevent the user from overexerting his muscles, FSRs were used in order to sense the user’s intended inputs since FSRs have an adjustable threshold. Adjusting the threshold will prove to be useful as the disease progresses, and can be adjusted in software accordingly. The clay controller and FSRs were chosen so as to not overexert the user’s muscles. In terms of electrical safety, fuses have been implemented to protect the user from dangerously high currents.

The team adhered to device-specific constraints fairly well in that the solenoids chosen are able to push down on a string with the force needed to produce a good sound when strummed. However, due to the large diameter of the solenoids, the two solenoids on the first fret of the neck of the ukulele are placed such that they are suspended above an area of high tension in the strings. More specifically, these two solenoids are placed close to the nut of the ukulele because of the staggering method used by the team to place all 16 solenoids on four frets to ensure that they all fit. As a result of this placement, the two solenoids are unable to push down with enough force to depress the strings above which they are placed. In keeping with the second device-specific constraint, all 16 solenoids were made to fit on the first four frets of the ukulele using the staggering method. However, due to the staggering method, there are four solenoids that are placed directly on the line of the frets to which they belong. The positioning of these solenoids posed a problem because when actuated on their respective frets, the string produced a buzzing sound when strummed. This buzzing sound is only apparent when a note is played on its own, rather than when played with other notes to form a chord. However, to get most of the solenoids to depress the strings and all of the solenoids to fit in the limited space on the ukulele in order to adhere to the constraints, the staggering method was necessary.

5.2 Providing Safety

In order to account for the user's safety while using the device, the team has implemented several ways of making the device safe. As mentioned earlier in this chapter, fuses were used as a form of protection against high currents. However, the solenoids on the stand and the two PCBs that held the circuitry for the device will be covered using clear, plastic boxes. Clear boxes are being used so that the user will not be able to touch the circuitry involved in the device, but will still be able to see if something isn't working within the device, allowing for easier debugging by a professional. Encasing the circuitry and solenoids will provide general protection around anything that shouldn't be handled directly by a user of the device.

5.3 Societal Impacts

The device created by the team was originally intended to be only used by the subject on whom the device was tested. However, the design of the device is fluid enough as an assistive aid to cater to the needs of any population suffering from permanent or progressive muscle disease. Stroke patients, as well as patients with other forms of muscular dystrophy may also benefit from a device such as this, since it limits the amount of effort needed on the user's end. The device was created so that few pieces of the device would require direct contact with the user. The piece that does require direct contact with the user is the clay hand controller, which is intended to be custom made to fit the user's grip, and contains FSRs that allow for modifiable thresholding. The FSRs are also cheap and can be easily replaced if one were to break. These specific characteristics of the device make it a viable assistive aid that works to improve the quality of life for patients who suffer from a broad spectrum of muscular disorders.

5.4 Subject Feedback

Once the team finished with the testing of the prototype, they wanted to visit the subject to have him test it himself. The purpose behind the visit was to ensure that the subject was able to use the prototype to play the ukulele. After playing for a certain period of time, the subject provided the team with positive and negative feedback on the prototype.

5.4.1 Code Latching

Due to the nature of DMD, the subject stated that he had a difficult time holding down the FSRs. The fingers on his left hand grew tired after pressing the sensors for too long. He suggested a technique that the team called "code latching".

Code latching is a method that allowed the subject to press an FSR once and the appropriate chord would hold its position. The subject could then release the FSR and strum the same chord. He would no longer have to apply constant force to the FSR in order to actuate the solenoids. Instead, he could have the ability to press an FSR once and have that chord "remembered" by the solenoids. To change chords, he could then push on another FSR, and the same "remembering" technique would be applied. The code latching technique would allow the subject to play the device for a longer period of time before taking a break. The team felt that this was important enough feedback from the subject to immediately implement it into the first prototype.

The latching code was implemented towards the end of the project. When an FSR was pressed, the chord selected by the FSR remained actuated for 30 seconds if no other chord was selected. As this code was implemented by using a counter, the counter would reset if a different chord has been selected. The team found that this duration of time of latching was sufficient for the subject.

5.4.2 FSRs

One comment that the subject had was the difficulty in finding the appropriate part of the FSR to press down upon. He suggested that the FSRs be moved to more accurately depict where his fingers would be placed on the clay mold. The team then worked to mark the clay mold where the fingers on the subject's left hand rested naturally. Moving the FSRs would help the subject not think about pressing directly on the FSR, but rather just applying force with each finger and knowing that the FSRs are placed in the correct spot.

An observation that the team noticed was that the FSR heads could be considered too small for the subject's fingertips. If the FSRs had larger heads, the subject would not have to strain so much to find the appropriate spot to press. A larger surface that senses force would improve the design of the clay mold because it would give the subject a larger area to press.

5.4.3 Solenoid Sound

A negative comment that the subject mentioned was the clicking noise that the solenoids produced when they were actuated. There were two clicking noises that involved the solenoids. The first clicking sound was internal to the solenoid when current flows through and caused the solenoid to bottom out. Therefore, the team could not do anything to completely rid the design of that particular clicking sound. However, the team examined the idea of masking the first clicking noise by covering the solenoids with a clear, protective box.

The second and smaller clicking noise occurred when the solenoids were released from actuation. This sound, also internal to the solenoid, was caused by the release of the plunger back up into the body of the solenoid. It only occurred when the solenoids were fully actuated from their original position. To prevent this, the team came up with the idea of presetting the solenoids so the plunger was never at the original position. The team thought this could be done by selecting a clear, protective box that is placed over the solenoids and stops the plunger from reaching its original position. A felt material could line the inside of the top of the box in order to decrease the sound of the plunger hitting the box.

Overall, the subject enjoyed using the prototype and provided critical feedback to the team. Some of the comments that the subject had were able to be easily implemented into the existing prototype while others will have to be fixed in later iterations of the design.

5.5 Manufacturability and Cost

If this device is to be manufactured, it would be rather challenging to reproduce the hand clay mold due to the uniqueness of the shape. In this project, the clay mold was specifically created to fit the subject's hand so he could rest his hand comfortably on the mold and be able to press his fingers down without reaching from his natural resting hand position. To make a hand mold that can be used by everyone with DMD, further research on hand shapes would have to be done. Furthermore, the solenoids that were used to press down the strings could be changed as described in previous sections. The team could not find solenoids that both had enough force to produce a good sound and were small enough to fit on the instrument. As there was no market for such specific solenoids, this was not surprising to the team. If these specific solenoids ever develop a market, they should be used to improve the device. The base price of one device is shown in the table below:

Table 11: Base price of the device

	Component	Price of 1 Unit	Total Units	Total
Hand Mold	Clay	$3	3oz	$9
	FSR	$5.95	5	$29.75
PCB	PCB Board	$33	2	$66
	Microcontroller	$2.95	1	$2.95
	LED	$0.25	25	$6.25
	Resistors	$0.15	-	~$10
	Voltage Regulator	$5	1	$5
	Buttons	$2.5	2	$5
	Operation Amplifiers	$3	2	$6
	MOSFETs	$0.63	16	$10
	Diode	$0.5	16	$8
	Connector	$0.25	25	$6.25
Output Hardware	Solenoid	$25	16	$400
Total				$564.20

The total amount of money the team spent on making the prototype was $564.20. This number could be significantly lowered by replacing the solenoids with cheaper, but effective actuators.

5.6 Limitations and Future Work

5.6.1 Hand Controller Modifications

As DMD progresses, a patient's wrist can be seen as gradually being locked in flexion. This phenomenon is called wrist flexion contracture, and can be seen in Figure 70. Flexion contracture occurs as a result of muscles being shortened. This shortening of muscles (in this case, in the hand and wrist) pulls the joint into a permanent flexion position. Since the subject's hand controller is constructed to fit the subject's hand, the design of the hand controller may need to change as the subject's wrist flexion contracture becomes more prominent.

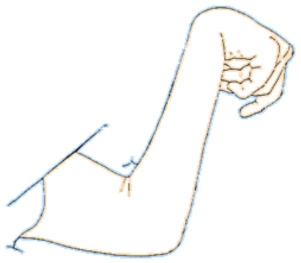


Figure 69: Wrist Flexion Contracture

[Howard, 2016]

A possible design for a modified hand controller that could account for wrist flexion contracture can be seen in Figure 71. This figure is an example of using unbaked polymer clay to create an amorphous shape to fit the subject's hand. Using clay to fit the natural shape of the subject's hand will allow for a customizable and comfortable fit. In addition, small FSRs (or even sensitive buttons) can be used to sense an input, provided that either is placed on a surface without major curvature.

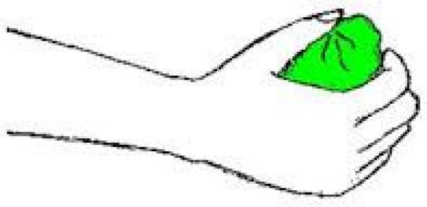


Figure 70: Modified Hand Controller Design

[The Physical Therapy Source, 2006]

Another possible device can be added to this project that could account for the inevitable challenges the subject may face would be an automated strumming device. This strumming device could be designed to strum the ukulele when the subject sends a "strum" signal. The device can have several options that allow the subject to choose a strumming pattern, or a pattern consisting of a group of strumming actions that are arranged differently to present different rhythms. These options would further automate the device, having the only input from the subject be the chord progressions of the subject's choice.

5.6.2 Solenoid Size and Force

As mentioned previously in the Methodology chapter of this paper, the team used a staggering method to ensure that all of the solenoids were pressing on the appropriate strings. The method the team used involved staggering the location of the solenoids across the inter-fret space. However, this staggering method caused some problems with the sound that the ukulele was producing when strummed. Some of the solenoids were placed over the frets on each string. This placement of the solenoids caused a muffled or buzzed sound when the solenoid was actuated on the string and that string was strummed. The sound concern could be controlled by using solenoids that are more forceful and have a smaller diameter. However, the team was unable to find small enough solenoids to fit the neck of the ukulele and still produce a great enough force to push down on the strings (based on measurements recorded in Appendix A).

The solenoids that were selected were 0.5” in diameter. These specific solenoids were chosen because the force of 8.34N supplied by the 0.5” diameter solenoids was enough to press down the strings. The team also discussed purchasing 0.35” diameter solenoids but, after testing the force supplied by the 0.35” solenoid, the team determined that the force of 5.56N on the smaller solenoids was not sufficient to press down on the strings to produce a good sound (i.e., no buzzing or muting) when strummed. Both solenoid types can be seen in Figure 72

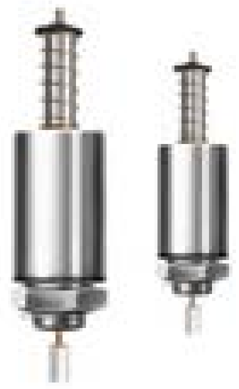


Figure 71: 0.5'' Solenoid (left) and 0.35'' Solenoid (right)
[Electro Mechanisms Inc., 2016]

Due to the 0.5” solenoids being too large and the 0.35” diameter solenoids not providing enough force, 0.5” solenoids were chosen. Ultimately, the staggering method was chosen to fit the larger solenoids in order to have enough force pressing down on each string. If the smaller solenoids were chosen, staggering would not be needed and the problem of location would be resolved, but the strings may not have been able to be pressed down with enough force, replicating the unwanted muted or buzzed sound. The best case would have been to use 0.35” diameter solenoids that provided the 8.34N force that the 0.5” diameter solenoids. Smaller solenoids could mean all four solenoids could fit across the frets uniformly, thus pressing down on the ideal location within each fret. The combination of a smaller solenoid that presses with more force would solve the problem of having a muffled or buzzed sound when a string is strummed when a solenoid is actuated on that string.

Even though the solenoids are not ideal, 14 out of 16 of them worked well in terms of the pitch produced. The pitch of a string is determined by the frequency at which the string vibrates when it is strummed. The higher the frequency, the higher the pitch. Appendix C shows the results of a pitch test that was conducted to compare the sound produced when using one's fingers to press down on a string and using solenoids to press down on a string. It was clear that most of the solenoids produced the same pitch as one would hear if one were to use his fingers to push down on the strings, showing that the actuated solenoids were comparable to a human finger pressing down on a string. The only difference is a slight buzz or a muffled sound that can be eradicated by either getting solenoids that push down on the string with more force or by using smaller solenoids with the same amount of force to remove the staggering.

Future implementations of the device could also include a method to actuate the strings on the ukulele using fewer solenoids. An algorithm could be created to achieve all desired notes and chords with less than 16 solenoids, since the same notes occur on multiple frets. Using fewer solenoids could alleviate the problem of having to space 16 solenoids on the first four frets of the ukulele.

5.6.3 Microcontroller Options and Code

The microcontroller choice and design have several key areas that can be altered in order to improve the overall design. While some of the design choices made worked well for the intended application given the time constraints imposed on the team, other choices could be altered for better performance. Alternatives that should be considered can be divided into three categories: microcontroller choice, USB versus wireless integration, and some aspects of the code design.

While the microcontroller that chose had enough GPIO pins and more than enough memory, it may not have been the best choice overall. For one, the PIC18F46K20 through-put package is large, having a 40 pin chip. Being more than 0.25" wide and over 1.25" long, this microcontroller is quite large and a lot of space is needed to fit it on a PCB. If the PCB for the device needs to be smaller in the future, it is recommended that a surface mounted package be used to decrease the size of the microcontroller. Another aspect of the microcontroller to consider is that every GPIO pin is being used on the chip besides the MCLR pin, which is being held at V_{DD} to stop the chip from being reset. As a result there is no room on the chip to add any more features. If there is a desire to add more features to the overall device, considering a microcontroller with more pins is highly recommended.

Currently, the team's device is using a USART device to transmit data from the chip to the computer. The USART device, called the MCP2221 breakout board, converts the data transmitted from the microchip into data that is readable by the computer, and vice versa. The MCP2221 has three available GPIO pins that were not used efficiently in the overall design. For example, these pins can be used to confirm if data are actually being sent or received to and from the computer. These pins can also be used as many other functions, all of which are described in the MCP2221 Breakout Board User's Guide. If the USB protocol is still a desired feature in future implementations of the team's device, it is recommended that these pins be used more effectively, rather than leaving them open and unused.

Rather than wired USB, it may be better to use a wireless interface for this device to create more maneuverability for the subject's hand controller. Since the entire device needs to be comfortable for the subject, a wireless hand controller may be a better choice so that he or she can move the controller wherever it is needed. If it is desired to integrate wireless features into this device, it is recommended that a microcontroller that supports Bluetooth be used. Another option would be to purchase a device that can convert serial communications into wireless signals, such as the Bluefruit LE by Adafruit.

Aside from the physical aspects of the microcontroller, there are parts of the overall code design that can be improved upon. A major update that is required for better microcontroller programming is to configure the ADC interrupt as periodic. Currently, the ADC is not periodic; the interrupt is being disabled in the main “while” loop. Since the code is using a CCP special event trigger, disabling the interrupts was required to synchronize with the 1ms timer. If the interrupts are not stopped to sync every time the timer has completed a cycle, then the ADC trigger will desynchronize with the timer, causing a loss in ADC sample gathering. Although there may be a better way to trigger the ADC interrupt, it was not found in the scope of this project. It is recommended that future iterations of this project make the ADC interrupt periodic so that the code is more stable.

The algorithm that the program uses to map a chord to a combination of solenoids to actuate can also be improved. The algorithm uses a unique method to determine which solenoid should be pushed down, instead of referring to a lookup table where all the traditional fingering of the chords is mapped. The uniqueness of the algorithm can present a learning curve for the subject. However, it was deemed necessary by the team to create a unique algorithm in order to conserve memory on the microcontroller. This algorithm currently works well with a majority of the chords in all major and minor keys, except one (the "A" diminished chord). The team has since upgraded the microchip as the project continued, providing more memory to store all the fingering information of all the chords. In this case, a traditional lookup table might be the better choice as the basis for the mapping algorithm since it would not only remove the learning curve set forth by the team's current algorithm, but also allow for a more flexible program and make debugging easier.

6. Conclusion

The overall goal of this project was to create an ergonomic hand interface to make it possible for a client with Duchenne Muscular Dystrophy to play the ukulele. The goal was met, as the final design of the project was able to provide the client with a comfortable and easy way to play the ukulele. The clay hand controller was formed to fit the client's resting hand posture, and the FSRs were placed in such a way that the user could easily push down on them with minimal force. Pressing down on one of these FSRs would actuate several solenoids, which were placed over the ukulele strings. When a solenoid is pressed down onto a string and that string is strummed, a note is formed. Several notes combined together produce a chord. The device worked such that pressing down on one FSR would output an entire chord. It was then the responsibility of the user to strum the strings of the ukulele to produce the desired sound.

Without the created device, the client's limited range of motion would prevent him from effectively playing the ukulele. As he does not have the finger dexterity or strength to press down on individual notes on the ukulele, he cannot press down on the required notes all at once to produce a chord. The final device helped the client to create chords in the specific musical key he desired. Thus, he was able to play music and chords that he would otherwise be unable to play.

Although the completed device met the overall goal of the project, there are still some areas that can be improved to make the device better for the user. If the limitations and recommendations are resolved, then this device may be able to be expanded to a wider audience and could potentially be used as an assistive aid to more than just the Duchenne Muscular Dystrophy community.

References

Bhowmik, Achintya K. *Interactive Displays: Natural Human-interface Technologies*. John Wiley & Sons, 2014. Print.

Boldrin, Luisa, Peter S. Zammit, and Jennifer E. Morgan. "Satellite Cells from Dystrophic Muscle Retain Regenerative Capacity." *Stem Cell Research* 14.1 (2015): 20-29. Web.

Capacicar. "1961 SG GUITAR-HEAD" *Capacicar-capacity of your car.* Web. 13 Apr. 2016.

Cascio, Christian M. Lo, Tsogyal D. Latshang, Malcolm Kohler, Thomas Fehr, and Konrad E. Bloch. "Severe Metabolic Acidosis in Adult Patients with Duchenne Muscular Dystrophy." *Respiration* 87.6 (2014): 499-503. Web.

Coton, Justine, Marcel De Gois Pinto, Julien Veytizou, and Guillaume Thomann. "Design for Disability: Integration of Human Factor for the Design of an Electro-mechanical Drum Stick System." *Procedia CIRP* 21 (2014): 111-16. Web.

"Types of Stroke." *Centers for Disease Control and Prevention*. Centers for Disease Control and Prevention, 2013. Web. 13 Apr. 2016. <http://www.cdc.gov/stroke/types_of_stroke.htm>.

"Data & Statistics." *Centers for Disease Control and Prevention*. Centers for Disease Control and Prevention, 2016. Web. 13 Apr. 2016.

"DMD." *Genetics Home Reference*. Web. 13 Apr. 2016. <http://ghr.nlm.nih.gov/gene/DMD>.

"Diseases - DMD." *Muscular Dystrophy Association*. 2015. Web. 13 Apr. 2016. <https://www.mda.org/disease/duchenne-muscular-dystrophy>.

"Diseases - DMD - Signs & Symptoms." *Muscular Dystrophy Association*. 2015. Web. 13 Apr. 2016. <https://www.mda.org/disease/duchenne-muscular-dystrophy/signs-and-symptoms>.

"Duchenne Muscular Dystrophy: MedlinePlus Medical Encyclopedia." *U.S National Library of Medicine*. U.S. National Library of Medicine. Web. 13 Apr. 2016. <https://www.nlm.nih.gov/medlineplus/ency/article/000705.htm>.

"FSR 400." Interlink Electronics, Inc. 2015. Web. 25 Apr. 2016. <http://www.interlinkelectronics.com/FSR400.php>.

"Getting Started with MIDI -." *Getting Started with MIDI* -. Web. 13 Apr. 2016. <http://www.midi.org/aboutmidi/index.php>.

Gordon, Paul. "Amyotrophic Lateral Sclerosis: An Update for 2013 Clinical Features, Pathophysiology, Management and Therapeutic Trials." *Aging and Disease A&D* 04.05 (2013): 295-310. Web.

Grounds, M. D., J. D. White, N. Rosenthal, and M. A. Bogoyevitch. "The Role of Stem Cells in Skeletal and Cardiac Muscle Repair." *Journal of Histochemistry & Cytochemistry* 50.5 (2002): 589-610. Web.

"How Acoustic Guitars Work." *HowStuffWorks*. 2000. Web. 13 Apr. 2016. <http://entertainment.howstuffworks.com/guitar3.htm>.

Kessler, Hubert. *Fundamentals of Music Theory*. Ann Arbor, MI: Edwards, 1945. Print.

Lakshmi Anand K. "Rack and pinion gearbox" Web. 13 Apr. 2016

Lang, Catherine E., Marghuretta D. Bland, Ryan R. Bailey, Sydney Y. Schaefer, and Rebecca L. Birkenmeier. "Assessment of Upper Extremity Impairment, Function, and Activity after Stroke: Foundations for Clinical Decision Making." *Journal of Hand Therapy* 26.2 (2013): 104-15. Web.

Lewington. "Ukulele High Tide Tenor." *Luna guitars*. Web. 13 Apr. 2016.

"Muscular Dystrophy: Hope Through Research." *Muscular Dystrophy: Hope Through Research*.

Web. 13 Apr. 2016. <http://www.ninds.nih.gov/disorders/md/detail_md.htm>.

Pallister. "These Gloves Will change the Way We Make Music", Says Imogen Heap." *Dezeen These Gloves Will Change the Way We Make Music Says Imogen Heap Comments*. 2014. Web. 13 Apr. 2016. <http://www.dezeen.com/2014/03/31/imogen-heap-gloves-mini-frontiers-movie/>.

Randall, J. K., and John Backus. "The Acoustical Foundations of Music." *Perspectives of New Music* 8.2 (1970): 149. Web.

"Rehab Measures - Action Research Arm Test." *The Rehabilitation Measures Database*. Web. 3 Apr. 2016. <http://www.rehabmeasures.org/Lists/RehabMeasures/DispForm.aspx?ID=951>.

"Rehab Measures - Fugl-Meyer Assessment of Motor Recovery After..." *The Rehabilitation Measures Database*. Web. 13 Apr. 2016.

Schauer, Michael, and Karl-Heinz Mauritz. "Musical Motor Feedback (MMF) in Walking Hemiparetic Stroke Patients: Randomized Trials of Gait Improvement." *Clin Rehabil Clinical Rehabilitation* 17.7 (2003): 713-22. Web.

Scott, Elaine, and Susan J. Mawson. "Measurement in Duchenne Muscular Dystrophy: Considerations in the Development of a Neuromuscular Assessment Tool." *Developmental Medicine & Child Neurology* 48.6 (2007): 540-44. Web.

"Search: Magicmakingmusic." *Search: Magicmakingmusic*. Web. 13 Apr. 2016. <http://quest.mda.org/article/magic-making-music>.

"The Brain from Top to Bottom." *The Brain from Top to Bottom* Web. 13 Apr. 2016. <http://thebrain.mcgill.ca/flash/d/d_06/d_06_cr/d_06_cr_mou/d_06_cr_mou.html>. <http://www.rehabmeasures.org/lists/rehabmeasures/dispform.aspx?ID=908>.

“The Ukulele Helper.” *The Ukulele Helper*. 2015. Web. 13 Apr. 2016.

"Which Software DAW Is Right for Me? | Sweetwater.com." *Which Software DAW Is Right for Me? | Sweetwater.com*. Web. 13 Apr. 2016. <http://www.sweetwater.com/feature/daw/daw_defined.php>.

White, C. K., A. M. Qureshi, V. Singh, J. E. Krager, J. E. Porlier, K. L. Wood, and R. H. Crawford. Modular Automated Assistive Guitar. Patent. 23 Oct. 2007. Print.

Appendix A

The following tables show the results of the force measurements taken on the ukulele to find the force required to push down on the strings to create acceptable pitch.

Fret Number	Length (cm)	Width (cm)	Area (cm^2)	String 1 Force Pull Length (N,cm)	String 2 (N, cm)	String 3 (N, cm)	String 4 (N, cm)
Nut	--	3.3	--	--	--	--	--
1	1.9	3.4	6.37	4.14, 0.2	2.95, 0.1	2.95, 0.1	3.38, 0.1
2	1.8	3.5	6.21	3.00, 0.3	3.54, 0.3	2.47, 0.2	3.02, 0.2
3	1.7	3.6	6.04	2.92, 0.4	2.46, 0.3	2.01, 0.3	1.24, 0.3
4	1.6	3.7	5.84	2.45, 0.4	2.04, 0.3	1.53, 0.3	2.90, 0.5

Ukulele measurements for string being pulled up to create an acceptable pitch

Fret Number	Length (cm)	Width (cm)	Area (cm^2)	String 1 Force, Push Length (N,cm)	String 2 (N, cm)	String 3 (N, cm)	String 4 (N, cm)
Nut	--	3.3	--	--	--	--	--
1	1.9	3.4	6.37	4.63, 0.2	4.44, 0.1	3.79, 0.1	3.84, 0.1
2	1.8	3.5	6.21	3.17, 0.3	3.75, 0.3	2.94, 0.2	3.10, 0.2
3	1.7	3.6	6.04	2.32, 0.4	2.89, 0.3	2.54, 0.3	1.99, 0.3
4	1.6	3.7	5.84	2.34, 0.4	3.10, 0.3	1.87, 0.3	2.63, 0.5

Ukulele measurements for the strings being pushed down to make an acceptable pitch

Appendix B

Header Files

Config.h:

```
/*
 * File:   config.h
 * Author: DMD Hand
 * Chip: PIC18F46K20
 *
 * NOTE: Code using XC8 compiler with legacy peripheral libraries
 */

#ifndef CONFIG_H
#define CONFIG_H

// CONFIG1H
#pragma config FOSC = INTIO67   // Oscillator Selection bits (Internal oscillator block, port function on RA6 and
RA7)
#pragma config FCMEN = OFF      // Fail-Safe Clock Monitor Enable bit (Fail-Safe Clock Monitor disabled)
#pragma config IESO = OFF       // Internal/External Oscillator Switchover bit (Oscillator Switchover mode
disabled)

// CONFIG2L
#pragma config PWRT = OFF       // Power-up Timer Enable bit (PWRT disabled)
#pragma config BOREN = OFF      // Brown-out Reset Enable bits (Brown-out Reset disabled in hardware and
software)
#pragma config BORV = 18        // Brown Out Reset Voltage bits (VBOR set to 1.8 V nominal)

// CONFIG2H
#pragma config WDTEN = OFF      // Watchdog Timer Enable bit (WDT is controlled by SWDTEN bit of the
WDTCON register)
#pragma config WDTPS = 32768    // Watchdog Timer Postscale Select bits (1:32768)

// CONFIG3H
#pragma config CCP2MX = PORTC   // CCP2 MUX bit (CCP2 input/output is multiplexed with RC1)
#pragma config PBADEN = ON      // PORTB A/D Enable bit (PORTB<4:0> pins are configured as analog input
channels on Reset)
#pragma config LPT1OSC = OFF    // Low-Power Timer1 Oscillator Enable bit (Timer1 configured for higher
power operation)
#pragma config HFOFST = ON      // HFINTOSC Fast Start-up (HFINTOSC starts clocking the CPU without
waiting for the oscillator to stablize.)
#pragma config MCLRE = ON       // MCLR Pin Enable bit (MCLR pin enabled; RE3 input pin disabled)

// CONFIG4L
#pragma config STVREN = ON      // Stack Full/Underflow Reset Enable bit (Stack full/underflow will cause
Reset)
#pragma config LVP = OFF        // Single-Supply ICSP Disabled
#pragma config XINST = OFF      // Extended Instruction Set Enable bit (Instruction set extension and Indexed
Addressing mode disabled (Legacy mode))

// CONFIG5L
#pragma config CP0 = OFF        // Code Protection Block 0 (Block 0 (000200-000FFFh) not code-protected)
#pragma config CP1 = OFF        // Code Protection Block 1 (Block 1 (001000-001FFFh) not code-protected)

// CONFIG5H
#pragma config CPB = OFF        // Boot Block Code Protection bit (Boot block (000000-0001FFh) not code-
protected)
#pragma config CPD = OFF        // Data EEPROM Code Protection bit (Data EEPROM not code-protected)

// CONFIG6L
#pragma config WRT0 = OFF       // Write Protection Block 0 (Block 0 (000200-000FFFh) not write-protected)
#pragma config WRT1 = OFF       // Write Protection Block 1 (Block 1 (001000-001FFFh) not write-protected)

// CONFIG6H
#pragma config WRTC = OFF       // Configuration Register Write Protection bit (Configuration registers (300000-
3000FFh) not write-protected)
#pragma config WRTB = OFF       // Boot Block Write Protection bit (Boot Block (000000-0001FFh) not write-
protected)
#pragma config WRTD = OFF       // Data EEPROM Write Protection bit (Data EEPROM not write-protected)

// CONFIG7L
#pragma config EBTR0 = OFF      // Table Read Protection Block 0 (Block 0 (000200-000FFFh) not protected from
table reads executed in other blocks)
#pragma config EBTR1 = OFF      // Table Read Protection Block 1 (Block 1 (001000-001FFFh) not protected from
table reads executed in other blocks)

// CONFIG7H
#pragma config EBTRB = OFF      // Boot Block Table Read Protection bit (Boot Block (000000-0001FFh) not
protected from table reads executed in other blocks)

#endif /* CONFIG_H */
```

Buttons.h (Provided by Professor Gene Bogdanov):

```
/*
 * buttons.h
 *
 *  Created on: Aug 12, 2012
 *      Author: Gene Bogdanov
 */

#ifndef BUTTONS_H_
#define BUTTONS_H_

// Buttons functions

#define BUTTON_COUNT 2    // number of buttons
#define BUTTON_SAMPLES_RELEASED 5 // number of samples before a button is considered released
#define BUTTON_SAMPLES_PRESSED 2 // number of samples before a button is considered pressed
// counter value indicating button pressed state
#define BUTTON_PRESSED_STATE (BUTTON_SAMPLES_RELEASED*BUTTON_SAMPLES_PRESSED)

extern volatile unsigned long g_ulButtons; // debounced button state, one per bit in the lowest bits

// update the debounced button state in the global variable g_ulButtons
// the input argument is a bitmap of raw button state from the hardware
void ButtonDebounce(unsigned long ulButtons);

#endif /* BUTTONS_H_ */
```

Chords.h:

```
/*
 * File:   chords.h
 * Author: DMD Hand
 * Chip: PIC18F46K20
 *
 * NOTE: Code using XC8 compiler with legacy peripheral libraries
 */

#ifndef CHORDS_H
#define CHORDS_H

void getChord(int input_code, char** chord);
void formChord(int root, int type, char** chord);
int chordToPin(char** chord);
void signalPin(int, int);
void start_clear();
void display_clear();
void getNote(int input_code);
int debouncer(int input_code, int debounced_input);
void displayKey();
void updateInput(int debounced_input) ;

#endif /* CHORDS_H */
```

CustomADC.h:

```
/*
 * File:   customADC.h
 * Author: DMD Hand
 * Chip: PIC18F46K20
 *
 * NOTE: Code using XC8 compiler with legacy peripheral libraries
 */

#ifndef CUSTOMADC_H
#define CUSTOMADC_H

typedef enum ADC_CaptureState {kADC_CaptureDone = 0, kADC_CaptureBusy} ADC_CaptureState_t;

void ADC_StartCapture(void);
ADC_CaptureState_t ADC_CaptureStatus(void);
void ADC_Read_Threshold(unsigned int x, unsigned int y, unsigned int z);

#endif /* CUSTOMADC_H */
```

CustomUSART.h:

```
/*
 * File:   customUSART.h
 * Author: DMD Hand
 * Chip: PIC18F46K20
 *
 * NOTE: Code using XC8 compiler with legacy peripheral libraries
 */

#ifndef CUSTOMUSART_H
#define CUSTOMUSART_H

void USARTdatacheck(void);
void USART_ADCvalues(unsigned int x, unsigned int y);
void USARTswitch(unsigned char buffer, unsigned int *finger);

#endif /* CUSTOMUSART_H */
```

Initialize.h:

```
/*
 * File:   initialize.h
 * Author: DMD Hand
 * Chip: PIC18F46K20
 *
 * NOTE: Code using XC8 compiler with legacy peripheral libraries
 */

#ifndef INITIALIZE_H
#define INITIALIZE_H

void PIC_Init(void);
void timer1_setup(void);
void ADC_Init(void);
void UART_Init(void);
void EEPROM_Init(void);

#endif /* INITIALIZE_H */
```

Variables.h:

```
/*
 * File:   variables.h
 * Author: DMD Hand
 * Chip: PIC18F46K20
 *
 * NOTE: Code using XC8 compiler with legacy peripheral libraries
 */

#include <xc.h>
#include "defines.h"

#ifndef VARIABLES_H
#define VARIABLES_H

// external Global Variables
extern const unsigned char ADC_InputSelect[5];

extern unsigned int ADC_samples[ADC_SAMPLE_SIZE][ADC_CHANNELS];
extern unsigned char SampleCount;
extern unsigned char ADCcompval[];
extern unsigned char ADCstring[4];
extern unsigned char rxbuffer[RX_BUFFER_SIZE];
extern const unsigned char StartMessage1[];
extern const unsigned char StartMessage2[];
extern const unsigned char option1[];
extern const unsigned char option1_2[];
extern const unsigned char option1_3[];
extern const unsigned char option2[];
extern const unsigned char option3[];
extern const unsigned char option3_2[];
extern const unsigned char selection1_1[];
extern const unsigned char selection1_2[];
extern const unsigned char selection1_3[];
extern const unsigned char selection1_4[];
extern const unsigned char selection1_5[];
extern const unsigned char selection2_1[];
extern const unsigned char selection2_2[];
extern const unsigned char selection2_3[];
extern const unsigned char selection2_4[];
extern const unsigned char selection2_5[];
extern const unsigned char selection3_1[];
extern const unsigned char selection3_2[];
extern const unsigned char selection3_3[];
extern const unsigned char selection3_4[];
extern const unsigned char selection3_5[];
extern const unsigned char mode1[];
extern const unsigned char mode2[];
extern const unsigned char error[];
extern const unsigned char newline[];
extern unsigned int q; // FOR USART
extern unsigned int ADC_sem; // ADC Viewer Semaphore
extern unsigned int thread_flag_1; // threshold selection flag (Light)
extern unsigned int thread_flag_2; // threshold selection flag (Medium)
extern unsigned int thread_flag_3; // threshold selection flag (Hard)
extern unsigned int g_threshold_1;
extern unsigned int g_threshold_2;
extern unsigned int g_threshold_3;
extern unsigned char chord_sel; // chord selection bit
extern unsigned int chord_flag; // chord selection flag
extern unsigned int afinger; // light-threshold finger selection
extern unsigned int bfinger; // medium-threshold finger selection
extern unsigned int cfinger; // hard-threshold finger selection
extern unsigned int error_flag;
extern unsigned char eeprom_fsrs[ADC_CHANNELS]; // Set to arbitrary address
extern unsigned char eeprom_chord; // Set to arbitrary address
extern unsigned char readthresh;
extern unsigned char readchord;
extern char UART1Config;
extern char baud;
extern unsigned int USART_flag; // Flag to determine if the USART messages have already been sent

extern int key;
extern int input_code;
extern int major;
extern int debounced_input;


#endif /* VARIABLES_H */
```

Defines.h:

```
/*
 * File:   defines.h
 * Author: DMD Hand
 * Chip: PIC18F46K20
 *
 * NOTE: Code using XC8 compiler with legacy peripheral libraries
 */

#ifndef DEFINES_H
#define DEFINES_H

// Defines
#define _XTAL_FREQ 16000000L // 16 MHz system clock
```

```
#define FCYC (_XTAL_FREQ/4L) // instruction clock frequency
#define ADC_TRIGGER_TIME ((FCYC)/1000L) // Number of Timer1 counts in 1.0ms
#define ADC_SAMPLE_SIZE 1
#define ADC_CHANNELS (sizeof(ADC_InputSelect)/sizeof(*ADC_InputSelect))
#define USE_AND_MASKS // For UART configuration
#define RX_BUFFER_SIZE 20

#endif /* DEFINES_H */
```

<u>Source Files</u>

Main.c:

```
/*
 * File:   main.c
 * Author: DMD Hand
 * Chip: PIC18F46K20
 *
 * NOTE: Code using XC8 compiler with legacy peripheral libraries
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <xc.h>
#include <string.h>
#include <plib/usart.h>
#include <plib/EEP.h>
#include "config.h"
#include "variables.h"
#include "defines.h"
#include "initialize.h"
#include "customADC.h"
#include "customUSART.h"
#include "chords.h"
#include "buttons.h"

/*** Interrupts ***/
void interrupt ADC_ISR()
{
  if(PIE1bits.ADIE)
    {
        if(PIR1bits.ADIF)
        {
            static unsigned int *iADC_samples = (unsigned int *)ADC_samples;
            static unsigned char ADC_Channel = 1;
            PIR1bits.ADIF = 0;
            // Check if starting a new capture
            if(SampleCount == 0)
            {
                // point to start of capture buffer
                iADC_samples = (unsigned int *)ADC_samples;
                // When the SampleCount is zero the ADC is presumed to have
                // completed a data conversion for the first channel
                ADC_Channel = 1; // select second channel for next conversion
            }
            // Set ADC channel for next conversion
            ADCON0 = ADC_InputSelect[ADC_Channel];
            // Read ADC value for completed conversion
            *iADC_samples++ = (unsigned int)(ADRESH)<<8 | ADRESL;
            // Select next channel to convert
            if(++ADC_Channel >= ADC_CHANNELS)
            {
                ADC_Channel = 0;
            }
            // Check if capture buffer is full
            if((++SampleCount) >= ADC_SAMPLE_SIZE * ADC_CHANNELS)
            {
                SampleCount  = 0;
                // Select first channel to convert on next capture start
                ADCON0 = ADC_InputSelect[0];

                // Disable the ADC interrupt to halt conversions
                // and indicate the capture buffer is full.
                PIE1bits.ADIE = 0;
            }

            // USART Control
            if (q < RX_BUFFER_SIZE) {
                rxbuffer[q] = ReadUSART(); //read the byte from rx register
                // Option 1 - Threshold Selection Mode
                // Option 1a - Light Threshold
                if (rxbuffer[q - 3] == 0x31 && (rxbuffer[q - 2] == 0x61 || rxbuffer[q - 2] == 0x41) && rxbuffer[q] ==
0x0D) //check for 1a[x] and then return key
                {
                   thread_flag_1 = 1;
                   USARTswitch(rxbuffer[q-1], &afinger);
                   for (; q > 0; q--)
                      rxbuffer[q] = 0x00; //clear the array
                   q = 0;
                   return;
                }
                // Option 1b - Medium Threshold
                if (rxbuffer[q - 3] == 0x31 && (rxbuffer[q - 2] == 0x62 || rxbuffer[q - 2] == 0x42) && rxbuffer[q] ==
0x0D) //check for 1b[x] and then return key
                {
                   thread_flag_2 = 1;
                   USARTswitch(rxbuffer[q-1], &bfinger);
                   for (; q > 0; q--)
                      rxbuffer[q] = 0x00; //clear the array
                   q = 0;
                   return;
                }
                // Option 1c - Hard Threshold
                if (rxbuffer[q - 3] == 0x31 && (rxbuffer[q - 2] == 0x63 || rxbuffer[q - 2] == 0x43) && rxbuffer[q] ==
0x0D) //check for 1c[x] and then return key
                {
                   thread_flag_3 = 1;
                   USARTswitch(rxbuffer[q-1], &cfinger);
                   for (; q > 0; q--)
                      rxbuffer[q] = 0x00; //clear the array
                   q = 0;
                   return;
```

```
            }
            // Option 2 - ADC Values Viewer
            if (rxbuffer[q - 1] == 0x32 && rxbuffer[q] == 0x0D) //check for 2 and then return key
            {
                if (ADC_sem == 0)
                    ADC_sem = 1;
                else
                    ADC_sem = 0;

                for (; q > 0; q--)
                    rxbuffer[q] = 0x00; //clear the array
                q = 0;
                return;
            }// Option 3 - Chord Selection Mode
            if (rxbuffer[q - 2] == 0x33 && (rxbuffer[q - 1] == 0x61 || rxbuffer[q - 1] == 0x41) && rxbuffer[q] ==
0x0D) //check for 3a and then return key
            {
                chord_flag = 1;
                chord_sel = 0x00;
                for (; q > 0; q--)
                    rxbuffer[q] = 0x00; //clear the array
                q = 0;
                return;
            }
            if (rxbuffer[q - 2] == 0x33 && (rxbuffer[q - 1] == 0x62 || rxbuffer[q - 1] == 0x42) && rxbuffer[q] ==
0x0D) //check for 3b and then return key
            {
                chord_flag = 1;
                chord_sel = 0x01;
                for (; q > 0; q--)
                    rxbuffer[q] = 0x00; //clear the array
                q = 0;
                return;
            }
            q++;
        } else {
            for (; q > 0; q--)
                rxbuffer[q] = 0x00; //clear the array
            q = 0;
            return;
        }
    }
  }
}

/*** Main ***/
int main() {
    // Initializations
    PIC_Init(); // Initialize PIC
    EEPROM_Init(); // Check value in EEPROM
    ADC_Init(); // Initialize ADC
    timer1_setup(); // Start Timer1 to be synched with ADC
    UART_Init(); // Initialize UART to 9600 baud

    // Enable Global Interrupts
    INTCONbits.GIE = 1;
```

```
//Local Variables
unsigned int k = 0;
unsigned int j = 0;
//int release_threshold = 100;
start_clear();

// Main Loop
while (1) {

    unsigned long presses = g_ulButtons;
    if (ADC_CaptureStatus() == kADC_CaptureDone) {
        ButtonDebounce((PORTAbits.RA4) | (PORTEbits.RE0) << 1);
        presses = ~presses & g_ulButtons;
        if (presses & 1) {
            key++;
            key %= 12;
        }
        if (presses & 2) {
            major = !major;
        }

        USARTdatacheck();
        for(j = 0; j < ADC_CHANNELS; j++)
        {
            for (k = 0; k < ADC_SAMPLE_SIZE; k++)
            {
                USART_ADCvalues(k,j);
                unsigned char read_fsrs = Read_b_eep(eeprom_fsrs[j]);
                if (read_fsrs == 0x01) {
                   ADC_Read_Threshold(k,j,g_threshold_1);
                }
                if (read_fsrs == 0x02) {
                    ADC_Read_Threshold(k,j,g_threshold_2);
                }
                if (read_fsrs == 0x03) {
                    ADC_Read_Threshold(k,j,g_threshold_3);
                }
            }//for k
        }//for j

        debounced_input = debouncer(raw_input_code, debounced_input);
        updateInput(debounced_input);
        if(Read_b_eep(eeprom_chord) == 0x00){
        displayKey();
        char* chord[3];
        getChord(input_code, chord);
        // A Diminished Chord
        if ((input_code == 5 && key == 10 && major) || (input_code == 2 && key == 7 && (!major))) {
            signalPin(1, 2);
            signalPin(2, 3);
            signalPin(3, 0);
            signalPin(4, 3);
        }
        else if(input_code != 0){
            chordToPin(chord);
            }

        }
            else if(Read_b_eep(eeprom_chord) == 0x01){
                display_clear();
                getNote(input_code);
            }
            ADC_StartCapture();
        }//if
    }//while
}//main
```

Buttons.c (Provided by Professor Gene Bogdanov):

```
/*
 * buttons.c
 *
 *  Created on: Aug 12, 2012
 *      Author: Gene Bogdanov
 */
#include "buttons.h"

// public global
volatile unsigned long g_ulButtons; // debounced button state, one per bit in the lowest bits

// update the debounced button state g_ulButtons
void ButtonDebounce(unsigned long ulButtons)
{
 int i;
    int mask = 0;
 static int piState[BUTTON_COUNT]; // button state: 0 = released
          // BUTTON_PRESSED_STATE = pressed
          // in between = previous state
 for (i = 0; i < BUTTON_COUNT; i++) {
  mask = 1 << i;
  if (ulButtons & mask) {
   piState[i] = piState[i] + (BUTTON_PRESSED_STATE/BUTTON_SAMPLES_PRESSED);
   if (piState[i] >= BUTTON_PRESSED_STATE) {
    piState[i] = BUTTON_PRESSED_STATE;
    g_ulButtons |= mask; // update button status
   }
  }
  else {
   piState[i] = piState[i] - (BUTTON_PRESSED_STATE/BUTTON_SAMPLES_RELEASED);
   if (piState[i] <= 0) {
    piState[i] = 0;
    g_ulButtons &= ~mask;
   }
  }
 }
}
```

Chords.c:

```
/*
 * File:   chords.c
 * Author: DMD Hand
 * Chip: PIC18F46K20
 *
 * NOTE: Code using XC8 compiler with legacy peripheral libraries
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <xc.h>
#include <string.h>
#include "chords.h"
#include "variables.h"
#define LATCHING 3750 // 30s with measured frequency of 125Hz (non-periodic)
// Variables

const unsigned char* notes[12] = {"c", "c#","d","d#","e","f","f#","g","g#","a","a#","b"};

const unsigned char* stringG[5] = {"g", "g#", "a", "a#", "b"};
const unsigned char* stringC[5] = {"c", "c#", "d", "d#", "e"};
const unsigned char* stringE[5] = {"e", "f", "f#", "g", "g#"};
const unsigned char* stringA[5] = {"a", "a#", "b", "c", "c#"};

const unsigned char** strings[4] = {stringG, stringC, stringE, stringA};

int non[4] = {0, 0, 0, 0};
int first[4] = {1, 0, 0, 0};
int second[4] = {0, 1, 0, 0};
int third[4] = {0, 0, 1, 0};
int fourth[4] = {0, 0, 0, 1};

int* S_table[5] = {non, first, second, third, fourth};

int pressed = 0;
int current_fret = 1;
int last_input = 0;
int debouncer_count = 50;
int latching_count = 0;
// Function figures out which chord the user is trying to play by converting the
// input code into an array of notes (the chord)
void getChord(int input_code, char** chord)
{
    switch(input_code)
    {
        case 0b00000000:
            start_clear();
            break;
        case 0b00000001:
            if(major){
                formChord(key, 0, chord);
            }
            else{
                formChord(key, 1, chord);
            }
            break;
        case 0b00000010:
            if(major){
```

```
            formChord(key+2, 1, chord);
        }
        else{
            formChord(key+2, 2, chord);
        }
        break;
    case 0b00000100:
        if(major){
            formChord(key+4, 1, chord);
        }
        else{
            formChord(key+3, 0, chord);
        }
        break;
    case 0b00001000:
        if(major){
            formChord(key+5, 0, chord);
        }
        else{
            formChord(key+5, 1, chord);
        }
        break;
    case 0b00010000:
        if(major){
            formChord(key+7, 0, chord);
        }
        else{
            formChord(key+7, 1, chord);
        }
        break;
    case 0b00000011:
        if(major){
            formChord(key+9, 1, chord);
        }
        else{
            formChord(key+8, 0, chord);
        }
        break;
    case 0b00000110:
        if(major){
            formChord(key+11, 2, chord);
        }
        else{
            formChord(key+10, 0, chord);
        }
        break;
    case 0b00001110:
        if(major){
            formChord(key+10, 0, chord);
        }
        else{
            formChord(key+7, 0, chord);
        }
        break;
  }//switch
}//getChord
```

```
void formChord(int root, int type, char** chord){
    chord[0] = (char*)(notes[root%12]);
    if(type == 0){
        chord[1] = (char*)(notes[(root+4)%12]);
        chord[2] = (char*)(notes[(root+7)%12]);
    }
    else if(type == 1){
        chord[1] = (char*)(notes[(root+3)%12]);
        chord[2] = (char*)(notes[(root+7)%12]);
    }
    else if(type == 2){
        chord[1] = (char*)(notes[(root+3)%12]);
        chord[2] = (char*)(notes[(root+6)%12]);
    }
}

// This function takes in a chord and decides which output pins to signal


int chordToPin(char** chord){
    int i, j, n;
    int stringFound[4] = {-1, -1, -1, -1};
    int noteFound[3] = {-1, -1, -1};
    for(j = 0; j< 3; j++)//each note
    {
        for(i = 0; i< 4; i++)//each string
        {
            for(n = 0; n < 5; n++){//each fret
                if(chord[j] == strings[i][n]){
                    if(i != noteFound[0] && i != noteFound[1] && i != noteFound[2]){
                        noteFound[j] = i;
                        stringFound[i] = n;
                    }
                }
            }//for n
        }//for i
    }//for j
    int k,x,y;
    for(k = 0; k < 4; k++){
        if(stringFound[k] == -1){
            for(y = 0; y < 5; y++){
                for(x = 0; x < 3; x++){
                    if(chord[x] == strings[k][y]){
                        stringFound[k] = y;
                    }
                }
            }
        }
    }
    int m;
    for(m = 0; m < 4; m++){
        if(stringFound[m] != -1)
            signalPin(m+1, stringFound[m]);
        else
            signalPin(m+1, 0);
    }
    return 0;
}//chordToChannel

// This function decides which pins to signal
void signalPin(int string, int fret)
{
    switch(string)
    {
        case 1:
            LATDbits.LATD2 = S_table[fret][3];
            LATDbits.LATD3 = S_table[fret][2];
            LATCbits.LATC4 = S_table[fret][1];
            LATCbits.LATC5 = S_table[fret][0];
            break;
        case 2:
            LATDbits.LATD4 = S_table[fret][3];
            LATDbits.LATD5 = S_table[fret][2];
            LATDbits.LATD6 = S_table[fret][1];
            LATDbits.LATD7 = S_table[fret][0];
            break;
        case 3:
            LATBbits.LATB0 = S_table[fret][3];
            LATBbits.LATB1 = S_table[fret][2];
            LATBbits.LATB2 = S_table[fret][1];
            LATBbits.LATB3 = S_table[fret][0];
            break;
        case 4:
            LATBbits.LATB4 = S_table[fret][3];
            LATBbits.LATB5 = S_table[fret][2];
            LATBbits.LATB6 = S_table[fret][1];
            LATBbits.LATB7 = S_table[fret][0];
            break;
    }
}//signalpin

// clear solenoids w/ LEDs at start of program
void start_clear(){
    LATDbits.LATD2 = 0;
    LATDbits.LATD3 = 0;
    LATCbits.LATC4 = 0;
    LATCbits.LATC5 = 0;
    LATDbits.LATD4 = 0;
    LATDbits.LATD5 = 0;
    LATDbits.LATD6 = 0;
    LATDbits.LATD7 = 0;
    LATBbits.LATB0 = 0;
    LATBbits.LATB1 = 0;
    LATBbits.LATB2 = 0;
    LATBbits.LATB3 = 0;
    LATBbits.LATB4 = 0;
    LATBbits.LATB5 = 0;
    LATBbits.LATB6 = 0;
    LATBbits.LATB7 = 0;
}
```

```
// clear display LEDs (when in Solo Mode)
void display_clear(){
   LATEbits.LATE2 = 1;
   LATAbits.LATA7 = 0;
   LATAbits.LATA6 = 0;
   LATCbits.LATC0 = 0;
   LATCbits.LATC1 = 0;
   LATCbits.LATC2 = 0;
   LATCbits.LATC3 = 0;
   LATDbits.LATD1 = 0;
}

void getNote(int input_code){
   int string[4] = {0,0,0,0};
   if(debounced_input == 0x00){
         if(pressed == 1){
            current_fret++;
            current_fret %= 5;
         }
         pressed = 0;
   }
   else if(debounced_input == 0x01){
         pressed = 1;
   }
   else if(debounced_input == 0x02){
      string[0] = current_fret;
   }
   else if(debounced_input == 0x04){
      string[1] = current_fret;
   }
   else if(debounced_input == 0x08){
      string[2] = current_fret;
   }
   else if(debounced_input == 0x10){
      string[3] = current_fret;
   }
   signalPin(1, string[3]);
   signalPin(2, string[2]);
   signalPin(3, string[1]);
   signalPin(4, string[0]);
}
int debouncer(int input_code, int debounced_input){
   if(input_code == last_input){
      debouncer_count--;
   }
   else{
      debouncer_count = 10;
   }
   if(debouncer_count <= 0){
      debouncer_count = 50;
      debounced_input = input_code;
   }
   last_input = input_code;
   return debounced_input;
}
```

```
void updateInput(int debounced_input){
   if(debounced_input != 0 && debounced_input != input_code){
         input_code = debounced_input;
         latching_count = 0;
   }
   else if(debounced_input == 0){
      if(latching_count >= LATCHING){
         start_clear();
         input_code = 0;
      }
      else{
         latching_count++;
      }
   }
}

void displayKey(){
   int keyDisplay[7] = {0,0,0,0,0,0,0};

   if(strstr(notes[key], "#") != NULL){
      LATDbits.LATD0 = 1; // set the sharp LED high
   }
   else{
      LATDbits.LATD0 = 0;
   }

   if(key < 5){
      keyDisplay[key/2] = 1;
   }
   else{
      if(key%2 == 0){
         keyDisplay[key/2] = 1;
      }
      else{
         keyDisplay[key/2+1] = 1;
      }
   }
   LATEbits.LATE2 = keyDisplay[5];
   LATAbits.LATA7 = keyDisplay[6];
   LATAbits.LATA6 = keyDisplay[0];
   LATCbits.LATC0 = keyDisplay[1];
   LATCbits.LATC1 = keyDisplay[2];
   LATCbits.LATC2 = keyDisplay[3];
   LATCbits.LATC3 = keyDisplay[4];

   if(major){
       LATDbits.LATD1 = 1; //set major LED high
   }
   else{
      LATDbits.LATD1 = 0;
   }
}
```

CustomADC.c:

```
/*
 * File:   customADC.c
 * Author: DMD Hand
 * Chip: PIC18F46K20
 *
 * NOTE: Code using XC8 compiler with legacy peripheral libraries
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <xc.h>
#include <string.h>
#include "customADC.h"
#include "variables.h"

// Start capture of 8 samples from 5 ADC channels
void ADC_StartCapture(void)
{
    PIE1bits.ADIE = 0; // stop ADC interrupts
    ADCON0 = ADC_InputSelect[0]; // select first channel to convert
    // synchronize with ADC trigger
    PIR1bits.ADIF = 0; // clear ADC interrupt request
    while(!PIR1bits.ADIF);
    PIR1bits.ADIF = 0; // clear ADC interrupt request
    while(!PIR1bits.ADIF);
    // start new capture
    SampleCount  = 0;
    PIE1bits.ADIE = 1; // start ADC interrupts
}

// Returns ADC_CaptureBusy when ADC capture is in progress
ADC_CaptureState_t ADC_CaptureStatus(void)
{
    return PIE1bits.ADIE ? kADC_CaptureBusy : kADC_CaptureDone;
}

// ADC Threshold determination

void ADC_Read_Threshold(unsigned int x, unsigned int y, unsigned int z) {
    if (ADC_samples[x][y] > z) {
        input_code |= (1 << y);
    } else {
        input_code &= ~(1 << y);
    }
}
```

CustomUSART.c:

```
/*
 * File:   customUSART.c
 * Author: DMD Hand
 * Chip: PIC18F46K20
 *
 * NOTE: Code using XC8 compiler with legacy peripheral libraries
 */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <xc.h>
#include <string.h>
#include <plib/EEP.h>
#include "customUSART.h"
#include "variables.h"

// Sends the user direction messages each time the USART board is powered on

void USARTdatacheck(void) {
    // if MCP board is on and the messages have not been previously sent
    if (PORTEbits.RE1 == 1 && USART_flag == 0) {
        // Display System Options
        putsUSART((char *) StartMessage1);
        putsUSART((char *) StartMessage2);
        putsUSART((char *) option1);
        putsUSART((char *) option1_2);
        putsUSART((char *) option1_3);
        putsUSART((char *) option2);
        putsUSART((char *) option3);
        putsUSART((char *) option3_2);
        USART_flag = 1; // Set flag to 1, indicating messages have already been sent
    }

    // if MCP board is not on, reset flag
    if (PORTEbits.RE1 == 0 && USART_flag == 1) {
        USART_flag = 0;
    }

    // Threshold Selection Mode - Light
    if (thread_flag_1 == 1) {
        switch (afinger) {
            case 1:
                Write_b_eep(eeprom_fsrs[0], 0x01);
                putsUSART((char *) selection1_1);
                break;
            case 2:
                Write_b_eep(eeprom_fsrs[1], 0x01);
                putsUSART((char *) selection1_2);
                break;
            case 3:
                Write_b_eep(eeprom_fsrs[2], 0x01);
                putsUSART((char *) selection1_3);
                break;
            case 4:
                Write_b_eep(eeprom_fsrs[3], 0x01);
                putsUSART((char *) selection1_4);
                break;
            case 5:
                Write_b_eep(eeprom_fsrs[4], 0x01);
                putsUSART((char *) selection1_5);
                break;
            default:
                break;
        }
        thread_flag_1 = 0;
    }

    // Threshold Selection Mode - Medium
    if (thread_flag_2 == 1) {
        switch (bfinger) {
            case 1:
                Write_b_eep(eeprom_fsrs[0], 0x02);
                putsUSART((char *) selection2_1);
                break;
            case 2:
                Write_b_eep(eeprom_fsrs[1], 0x02);
                putsUSART((char *) selection2_2);
                break;
            case 3:
                Write_b_eep(eeprom_fsrs[2], 0x02);
                putsUSART((char *) selection2_3);
                break;
            case 4:
                Write_b_eep(eeprom_fsrs[3], 0x02);
                putsUSART((char *) selection2_4);
                break;
            case 5:
                Write_b_eep(eeprom_fsrs[4], 0x02);
                putsUSART((char *) selection2_5);
                break;
            default:
                break;
        }
        thread_flag_2 = 0;
    }

    // Threshold Selection Mode - Hard
    if (thread_flag_3 == 1) {
        switch (cfinger) {
            case 1:
                Write_b_eep(eeprom_fsrs[0], 0x03);
                putsUSART((char *) selection3_1);
                break;
            case 2:
                Write_b_eep(eeprom_fsrs[1], 0x03);
                putsUSART((char *) selection3_2);
                break;
            case 3:
                Write_b_eep(eeprom_fsrs[2], 0x03);
                putsUSART((char *) selection3_3);
                break;
            case 4:
                Write_b_eep(eeprom_fsrs[3], 0x03);
                putsUSART((char *) selection3_4);
                break;
            case 5:
                Write_b_eep(eeprom_fsrs[4], 0x03);
                putsUSART((char *) selection3_5);
                break;
            default:
            break;
        }
        thread_flag_3 = 0;
    }

    // Detect Invalid USART Entry
    if (error_flag == 1){
        putsUSART((char *) error);
        error_flag = 0;
    }

    // Chord Selection Mode
    if (chord_flag == 1){
        Write_b_eep(eeprom_chord, chord_sel);
        chord_flag = 0;
        if (Read_b_eep(eeprom_chord) == 0x00){
            putsUSART((char *)mode1);
        }
        if (Read_b_eep(eeprom_chord) == 0x01){
            putsUSART((char *)mode2);
        }
    }

    else
        return;
}

// ADC values viewer function - view ADC values once enabled through USART
void USART_ADCvalues(unsigned int x, unsigned int y) {
    if (ADC_sem == 1) {
        putsUSART(ADCcompval);
        putsUSART(itoa(ADCstring, ADC_samples[x][y], 10));
        putsUSART((char *)newline);
    }
    else
        return;
}

// Switch function which switches flag based on inputs
void USARTswitch(unsigned char buffer, unsigned int *finger) {
    switch (buffer) {

        case 0x31: // 1 = thumb
            *finger = 1;
            break;
        case 0x32: // 2 = index
            *finger = 2;
            break;
        case 0x33: // 3 = middle
            *finger = 3;
            break;
        case 0x34: // 4 - ring
            *finger = 4;
            break;
        case 0x35: // 5 - pinky
            *finger = 5;
            break;
        default:
            error_flag = 1;
            break;

    }
}
```

Initialize.c:

```
/*
 * File:   initialize.c
 * Author: DMD Hand
 * Chip: PIC18F46K20
 *
 * NOTE: Code using XC8 compiler with legacy peripheral libraries
 */

//includes
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <xc.h>
#include <string.h>
#include <plib/usart.h>
#include <plib/EEP.h>
#include "initialize.h"
#include "defines.h"
#include "variables.h"

void PIC_Init(void)
{
// Disable Global Interrupt Sources
    INTCON = 0;

// Oscillator Configuration to 16 MHz
    OSCCONbits.IRCF = 0b111; // 16 MHz source
    OSCCONbits.SCS = 0b00;  // primary clock
    OSCTUNEbits.PLLEN = 0b0; // PLL disabled

// Disable All Interrupt Sources
    PIE1 = 0;
    PIE2 = 0;

// Key Selecting Buttons
    TRISAbits.RA4 = 1; // set RA4 pin as input -> pin 6
    TRISEbits.RE0 = 1; // set RE0 pin as input -> pin 8

// MCP Detect
    TRISEbits.RE1 = 1; // set RE1 pin as input -> pin 9

//USART
    TRISCbits.RC6 = 0; // set RC6 pin as output -> Tx pin 25
    TRISCbits.RC7 = 1; // set RC7 pin as input -> Rx pin 26
// Display LEDs
    TRISEbits.RE2 = 0; // set RE1 pin as output -> pin 10 (DA)
    TRISAbits.RA7 = 0; // set RA7 pin as output -> pin 13 (DB)
    TRISAbits.RA6 = 0; // set RA6 pin as output -> pin 14 (DC)
    TRISCbits.RC0 = 0; // set RA6 pin as output -> pin 15 (DD)
    TRISCbits.RC1 = 0; // set RA6 pin as output -> pin 16 (DE)
    TRISCbits.RC2 = 0; // set RA6 pin as output -> pin 17 (DF)
    TRISCbits.RC3 = 0; // set RA6 pin as output -> pin 18 (DG)
    TRISDbits.RD0 = 0; // set RA6 pin as output -> pin 19 (DSF)
    TRISDbits.RD1 = 0; // set RA6 pin as output -> pin 20 (DMm)

//stringG
    TRISDbits.RD2 = 0; // set RD2 pin as output -> pin 21 s1.1 G#
    TRISDbits.RD3 = 0; // set RD3 pin as output -> pin 22 s1.2 A
    TRISCbits.RC4 = 0; // set RC4 pin as output -> pin 23 s1.3 A#
    TRISCbits.RC5 = 0; // set RC5 pin as output -> pin 24 s1.4 B

//stringC
    TRISDbits.RD4 = 0; // set RD4 pin as output -> pin 27 s2.1 C#
    TRISDbits.RD5 = 0; // set RD5 pin as output -> pin 28 s2.2 D
    TRISDbits.RD6 = 0; // set RD6 pin as output -> pin 29 s2.3 D#
    TRISDbits.RD7 = 0; // set RD7 pin as output -> pin 30 s2.4 E

//stringE
    TRISBbits.RB0 = 0; // set RB0 pin as output -> pin 33 s3.1 F
    TRISBbits.RB1 = 0; // set RB1 pin as output -> pin 34 s3.2 F#
    TRISBbits.RB2 = 0; // set RB2 pin as output -> pin 35 s3.3 G
    TRISBbits.RB3 = 0; // set RB3 pin as output -> pin 36 s3.4 G#

//stringA
    TRISBbits.RB4 = 0; // set RB4 pin as output -> pin 37 s4.1 A#
    TRISBbits.RB5 = 0; // set RB5 pin as output -> pin 38 s4.2 B
    TRISBbits.RB6 = 0; // set RB6 pin as output -> pin 39 s4.3 C
    TRISBbits.RB7 = 0; // set RB7 pin as output -> pin 40 s4.4 C#
}//PIC_Init

// Initialize Timer1 to trigger an ADC sample every 1ms
void timer1_setup(void)
{
    PIE1bits.TMR1IE = 0; // disable TIMER1 interrupts
    T1CON = 0; // stop timer1, clock source FOSC/4, prescale 1:1
    T3CONbits.T3CCP2 = 0; // Select TIMER1 as clock source for CCP1 & CCP2
    T3CONbits.T3CCP1 = 0;
    TMR1H = 0; // Start timer1 at zero
    TMR1L = 0;
    PIR1bits.TMR1IF = 0; // clear interrupt request flag
    PIE2bits.CCP2IE = 0; // disable CCP2 interrupts
    CCP2CON = 0; // stop CCP2
    CCPR2H = (ADC_TRIGGER_TIME - 1) >> 8;
    CCPR2L = (ADC_TRIGGER_TIME - 1)&0xFF;
    CCP2CONbits.CCP2M = 0b1011; // Enable the special event trigger
    T1CONbits.TMR1ON = 1;
}//timer_setup

// Initialize ADC to sample on inputs AN0, AN1, AN2, AN3, AN4
void ADC_Init(void)
```

```
{
   PIE1bits.ADIE = 0; // disable ADC interrupt
   ADCON0  = 0; // stop ADC, ADC clock FOSC/32
   ANSEL   = 0x1F; // enable AN0, AN1, AN2, AN3, AN4 inputs (disabled otherwise)
   ANSELH  = 0x00; // disable analog inputs on remaining analog pins
   TRISA  |= 0x2F; // make RA0(pin 2), RA1(3), RA2(4), RA3(5), RA5(7) inputs
   ADCON1  = 0; // VREF internal VSS and VDD */
   ADCON2  = _ADCON2_ADFM_MASK // Right justified
         | (0b101 << _ADCON2_ADCS0_POSN) // ADC clock: FOSC/16
         | (0b001 << _ADCON2_ACQT0_POSN); // Acquisition time: 2 TAD
   ADCON0bits.ADON = 1; // power up ADC
   PIR1bits.ADIF = 0; // clear ADC interrupt request
   INTCONbits.PEIE = 1; // enable peripheral interrupts
}

// Initialize UART at a baud rate of 9600
void UART_Init(void)
{
   // Configure UART
   UART1Config = USART_TX_INT_OFF & USART_RX_INT_OFF & USART_ASYNCH_MODE &
USART_EIGHT_BIT & USART_BRGH_HIGH;
   baud = 103; // baud = X => (FOSC/desired rate)/[16(X+1)] -> desired rate = 9600 at 16 MHz
   OpenUSART(UART1Config, baud);
}

// Sets the threshold to the medium setting (500) if the value in a specific EEPROM address is not one of the
selectable options
// Also, makes sure chord selection mode (Smart Ukulele Mode) is already written into EEPROM
void EEPROM_Init(void) {
   int m;
   for (m = 0; m < ADC_CHANNELS; m++) {
      readthresh = Read_b_eep(eeprom_fsrs[m]);
      if (readthresh != 0x01 && readthresh != 0x02 && readthresh != 0x03)
         Write_b_eep(eeprom_fsrs[m], 0x02);
   }

   readchord = Read_b_eep(eeprom_chord);
   if (readchord != 0x00 && readchord != 0x01)
      Write_b_eep(eeprom_chord, 0x00);
   else
      return;
}
```

Variables.c:

```
/*
 * File:   variables.c
 * Author: DMD Hand
 * Chip: PIC18F46K20
 *
 * NOTE: Code using XC8 compiler with legacy peripheral libraries
 */

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

```
const unsigned char ADC_InputSelect[5] =
{
0x00 | _ADCON0_ADON_MASK, //AN0
0x04 | _ADCON0_ADON_MASK, //AN1
0x08 | _ADCON0_ADON_MASK, //AN2
0x0C | _ADCON0_ADON_MASK, //AN3
0x10 | _ADCON0_ADON_MASK, //AN4
};
unsigned int ADC_samples[ADC_SAMPLE_SIZE][ADC_CHANNELS];
unsigned char SampleCount;
unsigned char ADCcompval[] = "ADC value is: ";
unsigned char ADCstring[4];
unsigned char rxbuffer[RX_BUFFER_SIZE];
const unsigned char StartMessage1[] = "Basic System Options\n";
const unsigned char StartMessage2[] = "Please type the option number and then press enter.\n";
const unsigned char option1[] = "1 - Threshold Selection Mode. Please type in the desired threshold for each force-
sensitive button and then press enter.\n";
const unsigned char option1_2[] = "Option 1a[x] - Light Threshold; Option 1b[x] - Medium Threshold; Option 1c[x]
- Hard Threshold\n";
const unsigned char option1_3[] = "[x] corresponds to a finger, where 1 = thumb, 2 = index, 3 = middle, 4 = ring,
and 5 = pinky.\n";
const unsigned char option2[] = "2 - ADC Values Viewer. This will display the values from the force-sensitive
buttons. Press 2 and then enter after running to stop this mode.\n";
const unsigned char option3[] = "3 - Chord Selection Mode. This allows you to select the style in which you want to
play your device. Again, press the option number and then enter.\n";
const unsigned char option3_2[] = "Chord Selection Mode Option 3a - Smart Ukulele Mode; Chord Selection Mode
Option 3b - Solo Mode.\n";
const unsigned char selection1_1[] = "You selected the Light threshold for the thumb.\n";
const unsigned char selection1_2[] = "You selected the Light threshold for the index finger.\n";
const unsigned char selection1_3[] = "You selected the Light threshold for the middle finger.\n";
const unsigned char selection1_4[] = "You selected the Light threshold for the ring finger.\n";
const unsigned char selection1_5[] = "You selected the Light threshold for the pinky finger.\n";
const unsigned char selection2_1[] = "You selected the Medium threshold for the thumb.\n";
const unsigned char selection2_2[] = "You selected the Medium threshold for the index finger.\n";
const unsigned char selection2_3[] = "You selected the Medium threshold for the middle finger.\n";
const unsigned char selection2_4[] = "You selected the Medium threshold for the ring finger.\n";
const unsigned char selection2_5[] = "You selected the Medium threshold for the pinky finger.\n";
const unsigned char selection3_1[] = "You selected the Hard threshold for the thumb.\n";
const unsigned char selection3_2[] = "You selected the Hard threshold for the index finger.\n";
const unsigned char selection3_3[] = "You selected the Hard threshold for the middle finger.\n";
const unsigned char selection3_4[] = "You selected the Hard threshold for the ring finger.\n";
const unsigned char selection3_5[] = "You selected the Hard threshold for the pinky finger.\n";
const unsigned char mode1[] = "You are in Smart Ukulele Mode.\n";
const unsigned char mode2[] = "You are in Solo Mode.\n";
const unsigned char error[] = "Invalid Entry. Please try again.\n";
const unsigned char newline[] = "\n";
unsigned int q = 0; // FOR USART
unsigned int ADC_sem = 0; // ADC Viewer Semaphore
unsigned int thread_flag_1 = 0; // threshold selection flag (Light)
unsigned int thread_flag_2 = 0; // threshold selection flag (Medium)
unsigned int thread_flag_3 = 0; // threshold selection flag (Hard)
unsigned int g_threshold_1 = 250;
unsigned int g_threshold_2 = 500;
unsigned int g_threshold_3 = 750;
unsigned char chord_sel; // chord selection bit
unsigned int chord_flag; // chord selection flag
unsigned int afinger; // light-threshold finger selection
unsigned int bfinger; // medium-threshold finger selection
unsigned int cfinger; // hard-threshold finger selection
unsigned int error_flag; // flag to determine invalid entry
unsigned char eeprom_fsrs[ADC_CHANNELS] = {0xE1, 0xE2, 0xE3, 0xE4, 0xE5}; // Set to arbitrary address
unsigned char eeprom_chord = 0xF5; // Set to arbitrary address
unsigned char readthresh;
unsigned char readchord;
char UART1Config = 0;
char baud = 0;
unsigned int USART_flag = 0; // Flag to determine if the USART messages have already been sent

int key = 0;
int input_code = 0;
int major = 1;
int debounced_input = 0;
```

Appendix C

Results of the pitch tests conducted. The results show the pitches measured with solenoids depressing a string and strumming that string compared to fingers depressing a string and strumming that string.

	G String					
	Theoretical Pitch (Hz)	Note	Finger Pressed Pitch	Finger Note	Solenoid Pressed Pitch	Solenoid Note
Fret 1	207.65	G#/Ab	211.7	G#	201.3	G
Fret 2	220	A	222.2	A	220.5	A
Fret 3	233.08	A#/Bb	234.3	A#	232.9	A#
Fret 4	246.94	B	247.2	B	244.9	B
	C String					
	Theoretical Pitch (Hz)	Note	Finger Pressed Pitch	Finger Note	Solenoid Pressed Pitch	Solenoid Note
Fret 1	277.18	C#/Db	283.5	C#	276	C#
Fret 2	293.66	D	298.9	D	294.1	D
Fret 3	311.13	D#/Eb	316.8	D#	309.9	D#
Fret 4	329.63	E	335.5	E	325.2	E
	E String					
	Theoretical Pitch (Hz)	Note	Finger Pressed Pitch	Finger Note	Solenoid Pressed Pitch	Solenoid Note
Fret 1	174.61	F	178.4	F	170.4	E
Fret 2	185	F#/Gb	185.8	F#	184.4	F#
Fret 3	196	G	198.4	G	195.7	G
Fret 4	207.65	G#/Ab	209.5	G#	208.7	G#
	A String					
	Theoretical Pitch (Hz)	Note	Finger Pressed Pitch	Finger Note	Solenoid Pressed Pitch	Solenoid Note
Fret 1	233.08	A#/Bb	232.2	A#	233.6	A#
Fret 2	246.94	B	246.6	B	245.9	B
Fret 3	261.63	C	260.9	C	260.8	C
Fret 4	277.18	C#/Db	276.7	C#	276.9	C#

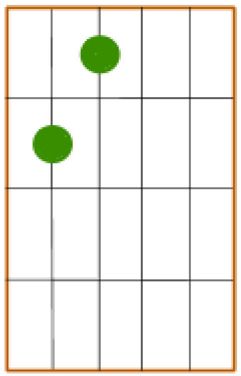
Finger pressed pitch vs solenoid pressed pitch

Appendix D

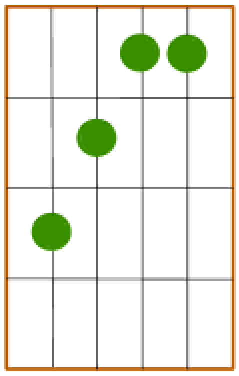
Major chords produced by the device when the ukulele neck is held vertically

(Nut of the ukulele here)

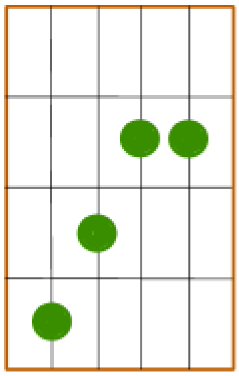
A



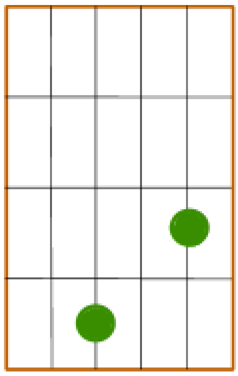
A#



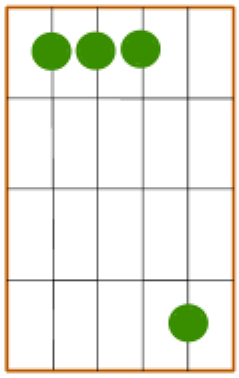
B



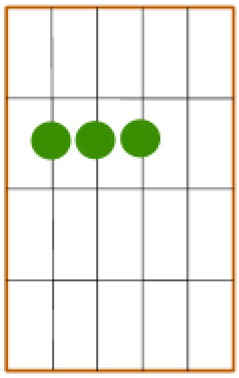
C



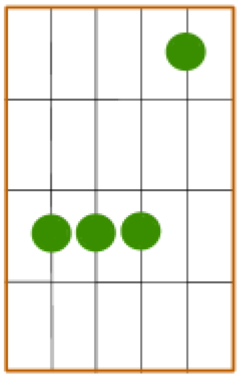
C#



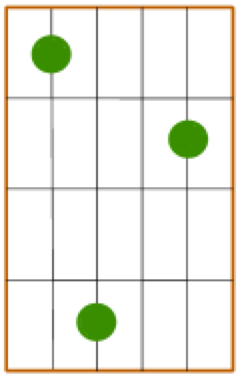
D



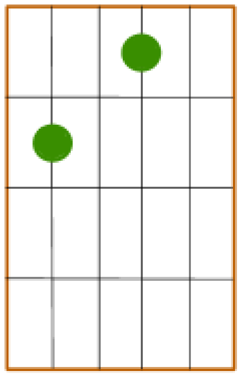
D#



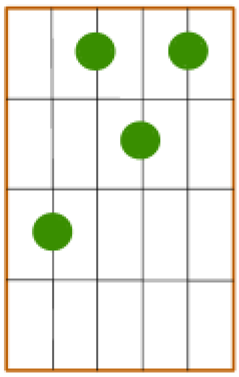
E



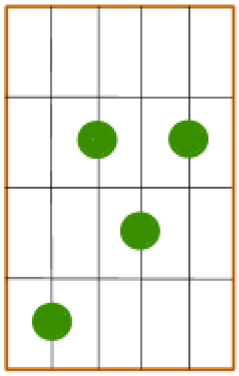
F



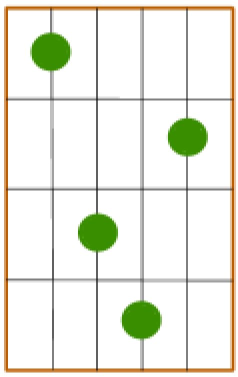
F#



G



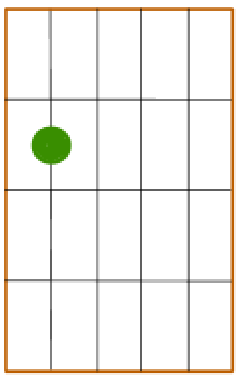
G#



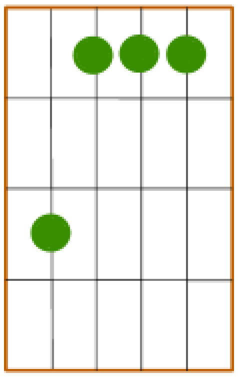
Minor chords produced by the device when the ukulele neck is held vertically

(Nut of the ukulele here)

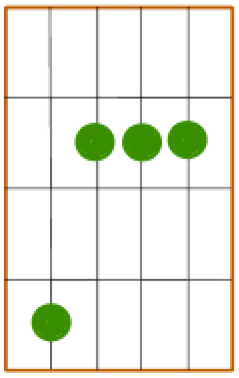
A



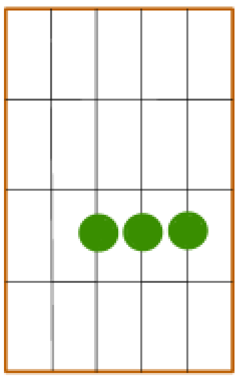
A#



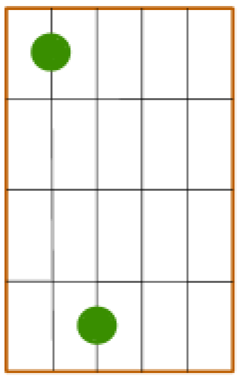
B



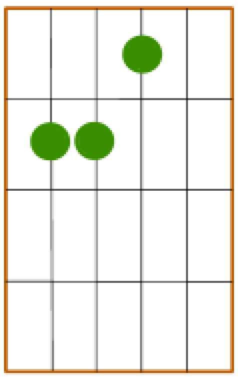
C



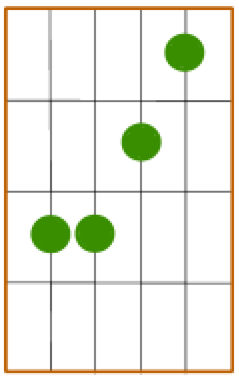
C#



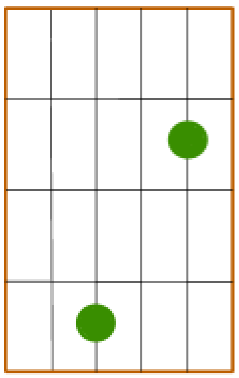
D



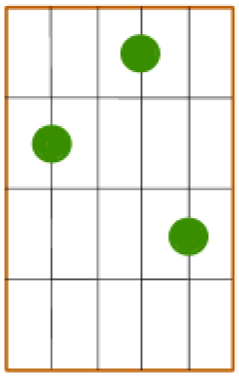
D#



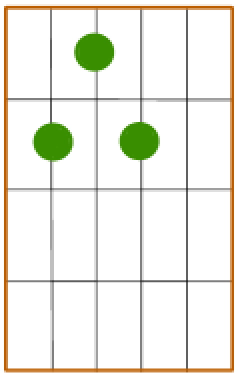
E



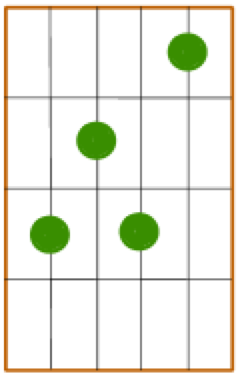
F



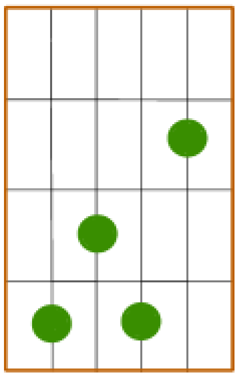
F#



G



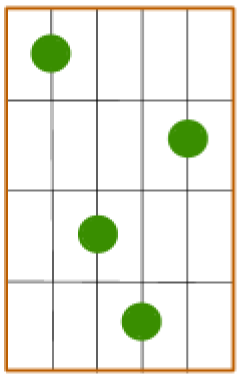
G#



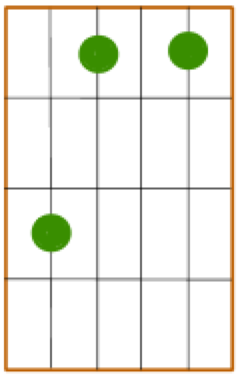
Diminished chords produced by the device when the ukulele neck is held vertically

(Nut of the ukulele here)

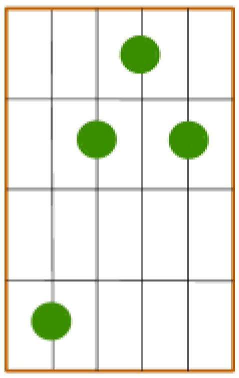
A



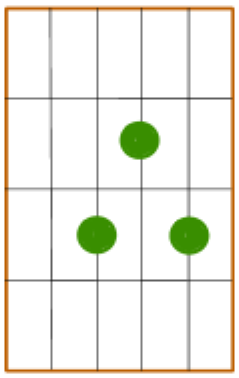
A#



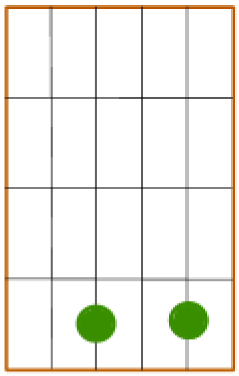
B



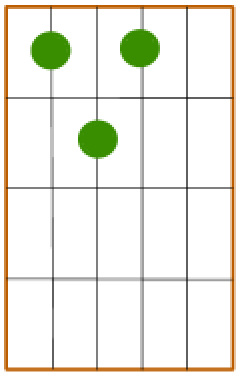
C



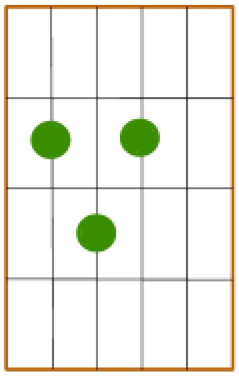
C#



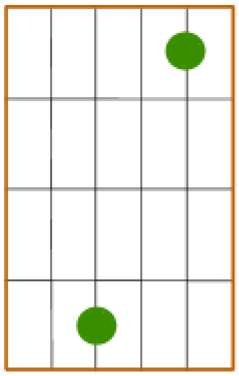
D



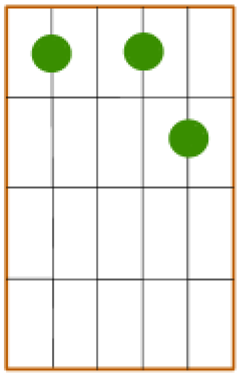
D#



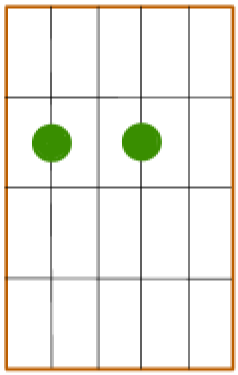
E



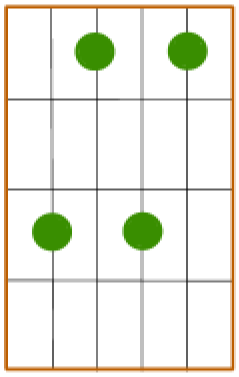
F



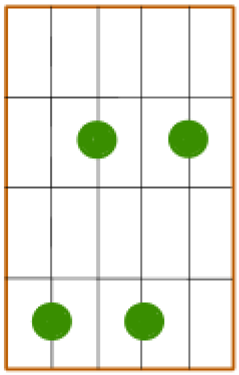
F#



G

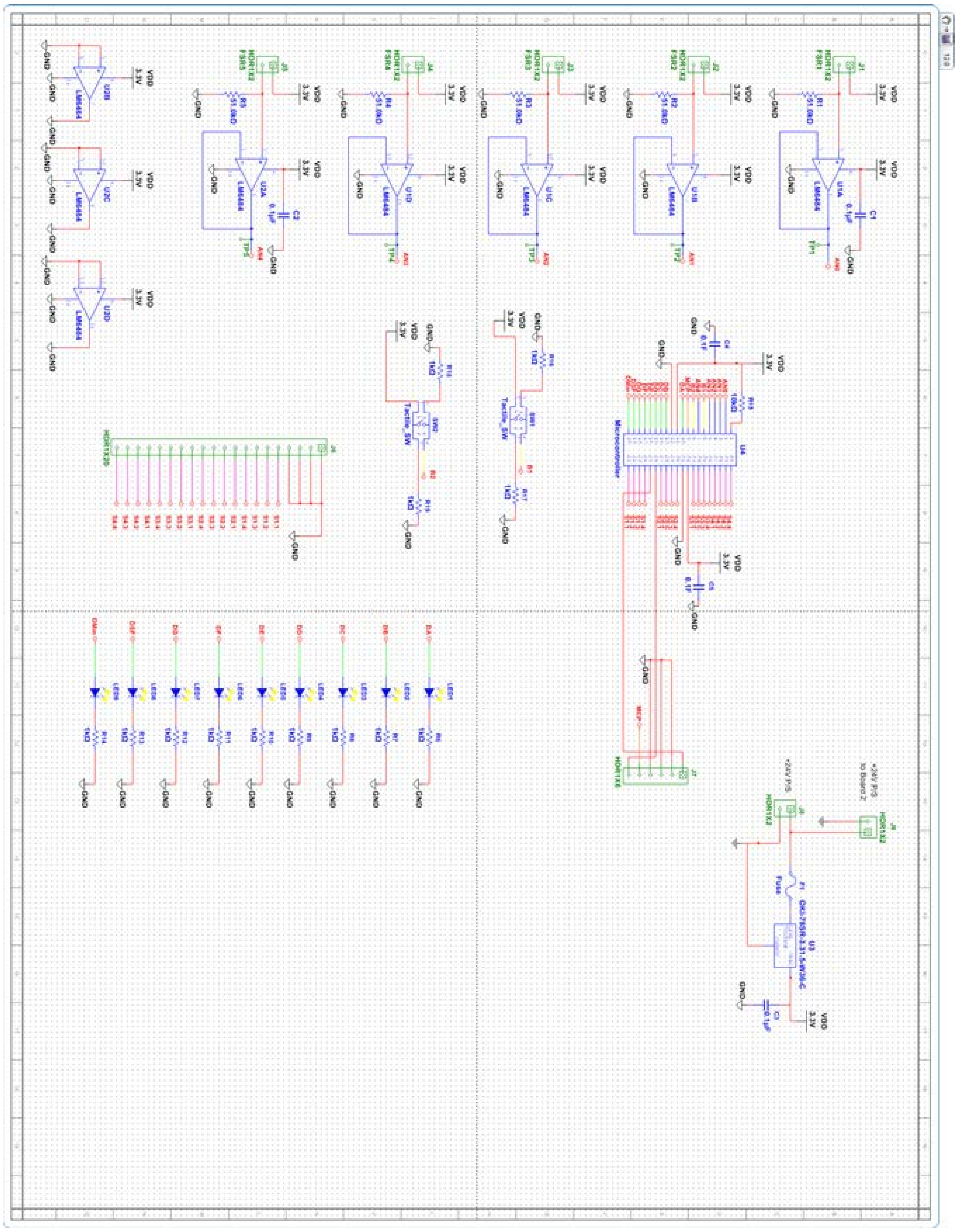


G#

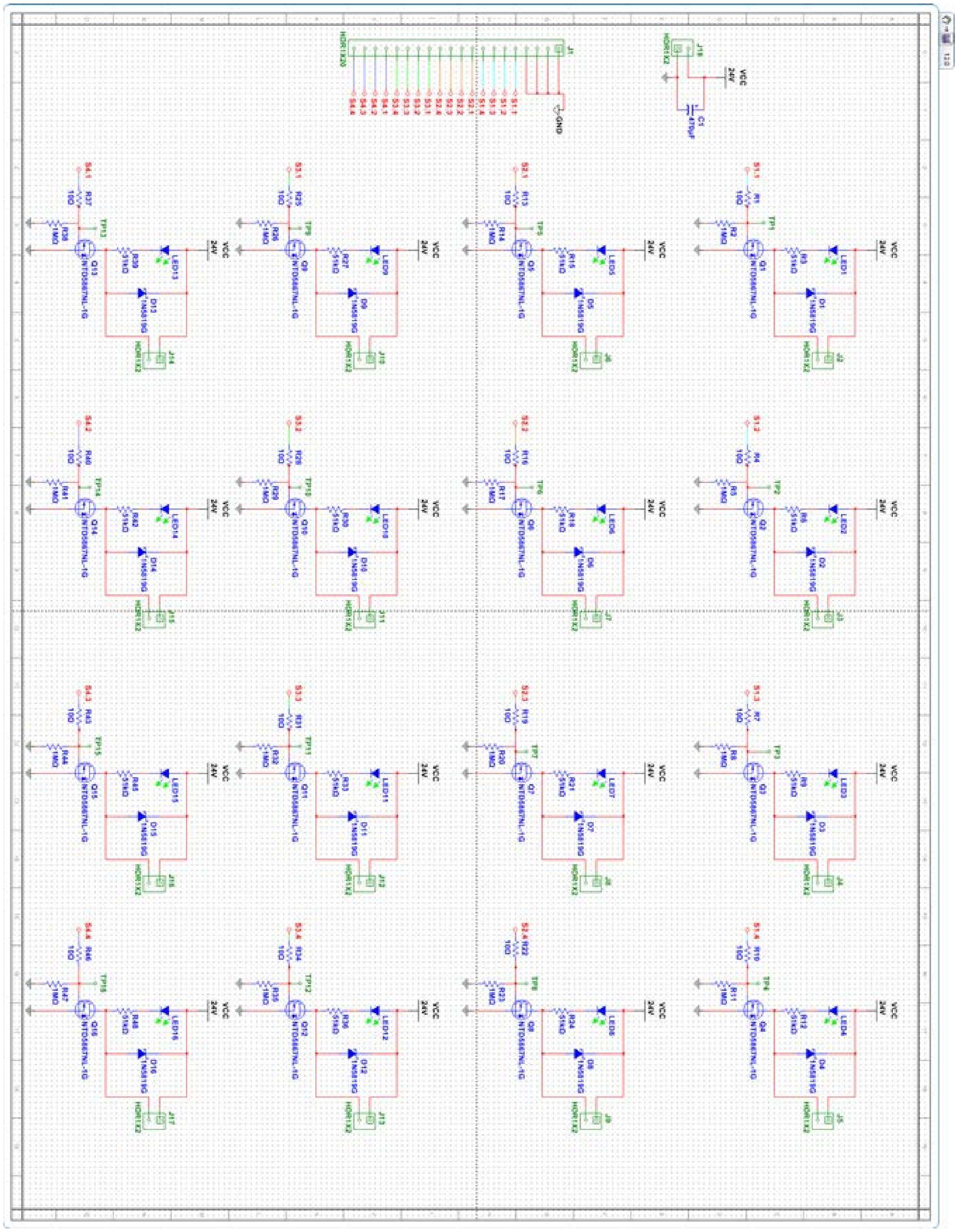


Appendix E

PCB Schematics (Board #1)

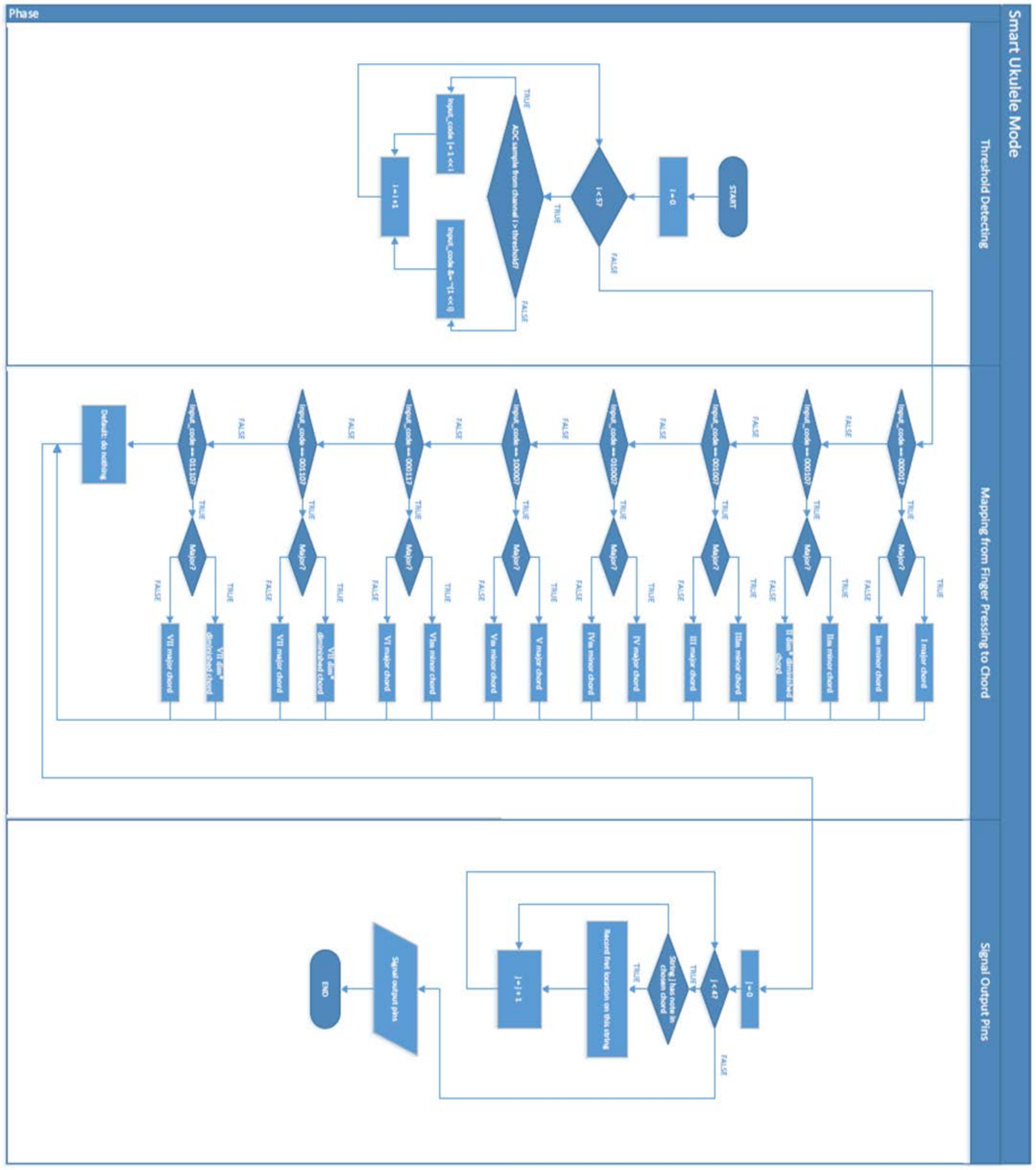


PCB Schematics (Board #2)



Appendix F

A flowchart showing the general layout of the main code. The Roman numerals in the “Mapping from Finger Pressing to Chord” section of the flowchart each correspond to a different note in a key. For example, in the C major key, the C note is I, the D note is II, the E note is III, and so on.



Appendix G

User's Guide

This user's guide is meant to help the user operate the device effectively. This user guide may also be used to help troubleshoot the device in the event that the device is not operating as intended.

Device Power Up

1. Plug the device into a wall outlet using the provided 24V power converter.
2. If the device is powered on correctly, the C and Major/Minor LEDs on the display PCB should be lit.
3. Please check that the FSRs are actuating the solenoids once the device is plugged in properly.

Using the Device

1. The buttons on the PCB are meant to be used to change the key in which chords will be played, as well as if the key is in a major or minor configuration.
   a. Toggling the button labeled "Key Select" on the PCB will cycle through the keys. Since the device starts at C when powered on, pressing this button will change the key into C#.
   b. The button labeled "Major/Minor" will change whether or not the key is major or minor. The LED labeled "Major/Minor" indicates the key is major when it is powered on. When this LED is off, this key is minor.
2. In the "Smart Ukulele Mode," each FSR press corresponds to a different chord.
   a. For example, when the FSR located in the thumb position of the device is pressed while the device is in the C major key, the solenoids will press down on notes that correspond to the C major chord.
   b. When the solenoids depress a string on the ukulele, the LEDs corresponding to those solenoids will light up on the LED display, located on the second PCB. A table is attached at the end of this guide to indicate which FSRs correspond to which chords in any given key.
3. In the "Solo Mode," each FSR (except for the FSR located in the thumb position) corresponds to one note in one fret. The thumb FSR is used to change the fret in which the solenoids will be actuated. The “A” key LED should be lit in this mode.

Initializing the USB Interface

Windows OS

1. If using Windows, please make sure that PuTTY is properly installed on your computer. You can download PuTTY at the following URL:
    a. http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html
    b. Please make sure that the correct operating system is chosen when downloading PuTTY. Only the putty.exe file is required. Once downloaded, it is recommended that the application file be placed onto the desktop of the chosen computer. Placing the application file on the desktop will allow for easy use of the program in the future.
2. Once PuTTY is properly download, make sure that the Category is selected as "Session". Then, select the "Serial" Connection Type under the "Specify the destination you want to connect to" option. Next, you will have to locate the COM port that the USB Serial Bus is using.
    a. **NOTE**: It is first recommended that under the "Terminal" category, both the "local echo" and "local line editing" under the "Line discipline options" should be changed to the "Force on" options. This will allow the user to see what is being written to the device.
3. Before locating the Serial Bus on your computer, you must make sure that the main ukulele device is unplugged and not powered! Then, you can plug the breakout board device into your computer (the USB port), which will provide enough power to the breakout board that it will start to download its drivers on your computer. Once the drivers have finished downloading, you can now search for the COM port.
    a. In Windows, the COM port can be found by searching "Devices and Printers" in the Start menu. Once there, search for a device called "MCP2221 USB-I2C/UART Combo". Right click on it, and select Properties. Go to the hardware tab, and the COM port should be listed under Device Functions.
4. Once the COM port has been successfully found, type it into the PuTTY terminal. Leave the baud rate at the default speed of 9600. Once this is done, open the terminal.
5. Now that PuTTY is set up and the device is plugged into the computer, the main device can be powered on. When this is done, additional messages will be displayed, telling the user how to operate in the terminal.

MAC

1. If using a Mac OS, PuTTY is not required. Instead, the embedded Mac terminal can be used. However, the main device should still be powered off until the device is connected to a computer.

2. When both the power is off and the USB is plugged in, please open up the Mac terminal. Instructions on how to use the terminal with this device can be found at this URL:
    a. http://ww1.microchip.com/downloads/en/DeviceDoc/MCP2200_MCP2221_CDC_Mac_Readme.txt
        i. It may be easier to use the "Serial Tools" described in this text file, but the team has not experimented with these tools at this time.
    b. Please make sure that the Baud rate is set to 9600 before proceeding to operate the device.

Using the USB Interface

NOTE: Please be aware that in this initial device, when the USB is plugged in, the yellow key selection LEDs will be slightly lit even when there is now power plugged into the device. Even though this happens, there is not enough power to actuate the solenoids. Please, do not touch the FSRs when the USB interface is connected. It is recommended to only use the FSRs if testing them while running the ADC values viewer option.

In the event that the display messages do not appear to the user with the USB plugged and the device powered, a means to operate the device without the display will be outlined below.

REMINDER: Please make sure that the device is unpowered before plugging the USB into the computer. Once the USB device is plugged in, open the PuTTY terminal and set the COM port for Windows or find the device in the Mac terminal and set the baud rate to 9600. Once the USB is plugged in and the terminal is opened, power can be applied to the main device.

To use the USB Interface, please read the following directions, which should be displayed on USB device start-up:

- Basic System Options
- Please type the option number and then press enter.
- 1 - Threshold Selection Mode. Please type in the desired threshold for each force-sensitive button and then press enter.
    - Option 1a[x] - Light Threshold; Option 1b[x] - Medium Threshold; Option 1c[x] - Hard Threshold
    - [x] corresponds to a finger, where 1 = thumb, 2 = index, 3 = middle, 4 = ring, and 5 = pinky.

- 2 - ADC Values Viewer. This will display the values from the force-sensitive buttons. Press 2 and then enter after running to stop this mode.
- 3 - Chord Selection Mode. This allows you to select the style in which you want to play your device. Again, press the option number and then enter.
    - Chord Selection Mode Option 3a - Smart Ukulele Mode; Chord Selection Mode Option 3b - Solo Mode.

As an example, suppose the user would like to select the Medium Threshold for the middle finger. The user would type in "1b3" in the terminal and then presses enter. The terminal should then output this message: "You selected the Medium threshold for the middle finger." In a similar manner, to select the "Solo Mode" in option 3, the user would need to type in "3b" in the terminal, and then press Enter. The terminal will then output the message: "You are in Solo Mode."

In the event that the initial display messages shown above are not outputted properly, it is recommended that the user type in "2" and then enter into the terminal to make sure that the serial communication is still working. If a stream of numbers (the ADC data) is outputted in the terminal, the serial communication is working properly. Type in "2" and enter again to end the streaming of numbers.

When finished with the USB interface, please power down the main ukulele device. When powering down the device please make sure that the LEDs are completely shut off (when the device is unplugged, there may be some residual charge in the power converter that is plugged into the outlet). When that task is completed, you may now remove the USB cord from the computer and close the terminal. When powering the main device back on, it should operate normally.