April 2014

Sans Trumpet

Anthony Mastromattei
Worcester Polytechnic Institute

Thomas Van Nguyen
Worcester Polytechnic Institute

Project Number: JAO 1401

SANS TRUMPET

WPI

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

by

Anthony Mastromattei

Thomas V. Nguyen

Date: May 1, 2014

Approved:

Professor John A. Orr, Major Advisor

Professor Scott D. Barton, Co Advisor

Abstract

The goal of this Major Qualifying Project was to create an interactive MIDI controller that is intuitive to use and thus, enables people with various levels of expertise to make music. Based upon a glove concept and utilizing the playing technique of a trumpet (as the gestures of a hand were determined to be most conveniently translated), a MIDI device, the *SansTrumpet*, was created that can be utilized in various Digital Audio Workstations (DAWs), such as *GarageBand*, *ProTools*, and *Logic*. The components of the *SansTrumpet* include a *Teensy* microcontroller, flex sensors, and a proximity sensor. The final product features the full pitch range of a standard trumpet, offers an alternative and intriguing visual aspect in performance, and allows for MIDI capability.

Acknowledgements

The team would like to thank Professor John Orr and Professor Scott Barton for advising the *SansTrumpet* project. Their comments and valuable insight were beneficial and crucial for the development of this project. The team would also like to thank Robert Boisse for his advice on many technical aspects as well as instructions on the usage of various lab equipment. We would also like to thank Tonino Mastromattei for guidance on usage of mechanical equipment. Finally, we would like to thank Esterina Mastromattei for sewing on the gloves. Without their guidance, the project would not have been accomplished successfully.

Table of Contents

List of Figures

List of Tables

1. Introduction

The advent of the computer and electronic instruments has provided a foundation to expand the capabilities of making music. Performances have greater flexibility today as a result of digital technology where instruments are modeled for a greater palette of sounds and a reduction in cost. Computers have also impacted recording technology where anyone with a computer can potentially achieve respectable recordings, and so affordable it's free on an Apple computer (*Garageband*). Technology has also enabled musicians to interact with sound in new ways. Using body gestures to further engage a musician is a growing trend of popular electronics today. The ability to play an instrument, without actually feeling or holding it, is a concept that is both fascinating and visually stimulating.

This project seeks to implement a gesture-based concept to an electronic musical instrument. The main attraction of such a device is the ability to generate sound with a nontraditional instrument through hand movement. The team has decided to design a MIDI (Music Instrument Digital Interface) controller musical instrument built upon a glove-based concept. The feel of an instrument was considered a critical aspect in the initial concept. The ability for the musician to physically be able to sense that resistance is involved during play (by the glove) allows for intended response by the user. Possible playing techniques with the glove were formulated by examining pre-existing instruments that would conveniently translate to a glove-based instrument. The instrument whose playing technique was chosen was a trumpet. The concept of the device is the left hand emulates the musician blowing into the instrument, while the right hand emulates the musician pressing the valves. The market this device is intended for is trumpet players, both beginners and experts, who desire a MIDI controller. With a MIDI controller, a trumpet player is able to control various MIDI parameters in their respective Digital Audio Workstation (DAW), while not having to learn how to play a keyboard, which is the typical form of synthesizers and MIDI controllers. This device assumes its consumers are proficient in trumpet playing and experienced enough to select an instrument plugin within a DAW. The attraction to this device, today, is that it follows a popular trend in society with a fascination of interactive devices.

2. Background

Research was conducted to determine devices that implemented a gesture-based concept to visually convey the generation of music. The characteristics of these devices were essential in the formulation of the *SansTrumpet* design to appropriately balance the expression of the visual aspects and intuitiveness of the playing technique. Further research was also conducted for the current market products associated with unique electronic instruments, as well as to understand the advantages and disadvantages of these instruments to better grasp what users seek in these types of musical devices. MIDI is the standard form of communication between electronic devices and a computer. The team investigated the MIDI operation to better comprehend the proper MIDI implementation for the *SansTrumpet*. Such research is essential in determining requirements of the instrument as well as features to be included.

2.1 Visual Components of Musical Experience

The visual aspects of live performances play an essential role in entertaining the audience. A study conducted by Chia-Jung Tsay investigates the influence of sound and visual aspects on audiences' judgment of live performances. The study asked participants to predict the winner of a live performance of a music competition [1]. The study found that prior to the experiment, 83.3% of the participants made up of both experts and novices, stated that sound is more important in the prediction of the winner [1]. The authors of the study showed the participants sound only excerpts of the performances, as well as video only excerpts [1]. Naturally, one would think that it is the sound that audiences would base their judgments on. However, the results of Tsay's study indicated otherwise; the results of his study showed that people actually base musical performances more on visuals rather than sound [1]. With sound only, the novice participants predicted the winner 25.5% and with video only, 52.5% of the participants correctly predicted the winner. The same was conducted for expert participants where the results displayed that with sound only, 25.7% predicted the winner correctly; with video only, 46.6% predicted the winner [1]. The data from this study indicates that both novices and experts value the visual criteria of musical performances more than sound when rating the performance. Therefore, the design of the musical instrument should implement some form of visual aspect.

2.2 The History of Interactive Controllers

2.2.1 *Theremin*

One of the early instruments that pioneered and inspired hands-free electronic interactive music was the *Theremin*. Through such interactivity, the visual aspect is conveyed to the audience through its playing technique. The *Theremin* was invented by Leon Theremin, a Russian physics professor, in 1919. It was one of the first assembled electronic instruments and was unique because the instrument could be played without any physical contact [2]. It also was the first electronic instrument to utilize wireless, non-contact sensing. The concept of the instrument is that the performer moves their hands in the proximity of two metal antennas: a vertical antenna associated with pitch (typically controlled with the performer's right hand) and a horizontal antenna associated with volume (typically controlled with the performer's left hand) [3].

Figure 1: Leon Theremin Playing *Theremin* [4]

Figure 1 displays Leon Theremin demonstrating his instrument. Each antenna operates using two radio frequency oscillators found in the instrument's circuitry. One oscillator has a fixed frequency, while the other has a variable frequency, which is fluctuated by the player's hand's distance from the antenna. The performer's hand functions as a grounded plate (because the user's body is connected to ground) for a variable capacitor in an inductance capacitor circuit. The difference in frequency that results between the two oscillators forms an audio signal, which is sent to a speaker [2].

To play the *Theremin*, a particular skill in hand "dexterity and movement" is necessary, as well as having a good "ear" for music [5]. An expert performer is able to link sections in space to particular notes, essentially being able to see a "piano keyboard that is invisible to everyone else" [6]. To sound a note, the player must place the "pitch" hand in the proper proximity of the antenna until the desired note is reached, and must place the "volume" hand in the appropriate distance of the antenna until the desired level is attained. The left hand can be pulled away at the suitable time to sound different variations of note intervals (full, half, quarter, etc.). The *Theremin* can also produce vibrato and tremolo. To form a vibrato effect, the performer must keep the "volume" hand steady, while shaking the "pitch" hand. To produce a tremolo effect, the player must keep the "pitch" hand steady, while shaking the "volume" hand [5].

While the *Theremin* provides the performer a highly responsive instrument, one can imagine how challenging it is to play depending on where the location of notes is in space.

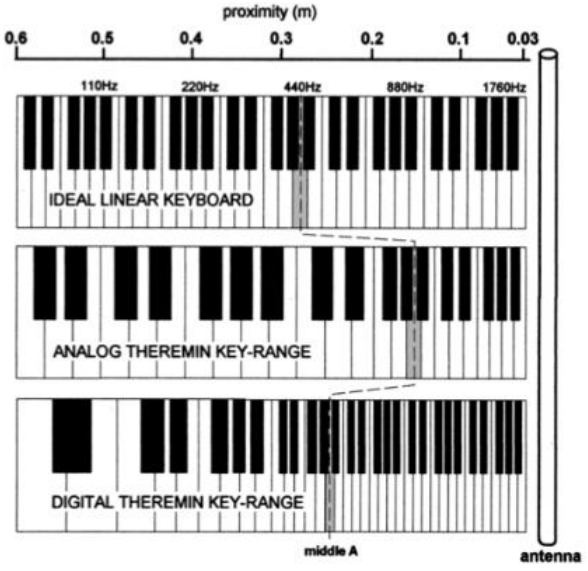

Figure 2: Key Ranges of Analog and Digital Theremin compared to Linear Keyboard [7]

Figure 2 compares a typical linear keyboard to the key range of an analog and digital *Theremin*, relating to the proximity of notes in relation to an antenna. As can be seen, the key range rapidly expands when the "pitch" hand is brought closer to the antenna. Therefore, the instrument seems reasonably difficult to perform when playing near the antenna to locate the correct note. The *Theremin* is still in production today by the company, *Moog*.

2.2.2 *The Hands*

Figure 3: *The Hands* [8]

The Hands, developed by Michel Waisvisz, is also based on the concept of generating music through hand movements. According to Waisvisz, *The Hands* is considered the first MIDI controller [9]. *The Hands* utilizes small keyboards along with sensors that generate signals, which are then sent to a computer in MIDI format [10]. *The Hands* incorporates sensors such as pressure sensors, mercury switches, and ultrasound transducers that are mapped to several music parameters [11]. Therefore, the MIDI signal can then be processed by the computer software to perform various functions such as playing a note.

2.2.3 *Lady's Glove*

Figure 4: The first version of the *Lady's Glove* [12]

The *Lady's Glove,* which was first built by Laetitia Sonami and Paul DeMarinis in 1991, also incorporates the concept of generating music through hand gestures [13]. The first *Lady's Glove* consisted of only Hall Effect transducers and a magnet, as depicted in Figure 4 [13]. The idea of the *Lady's Glove* is to exploit the concept of the Hall Effect with the use of the transducers and a magnet to vary the voltage signal that is interpreted into a MIDI message [13]. The magnet is positioned in the palm area of one glove, while the other glove hosts the Hall Effect transducers at the tip of the fingers as seen in Figure 4. Therefore, depending on the proximity between the hand with the magnet and fingers of the other hand, the signal will fluctuate. With the use of a computer, the glove can be used along with a DAW to generate sound. Thus, the glove allows the user to create music with simple finger movements.

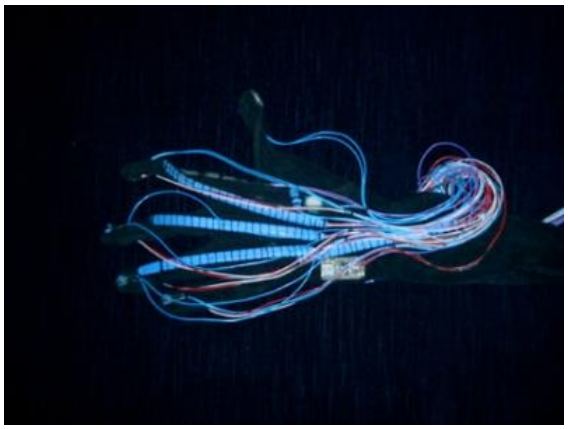

Figure 5: The fourth version of the *Lady's Glove* [14]

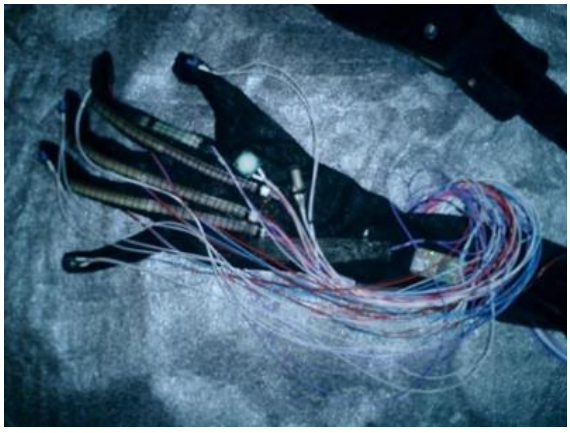

Figure 6: The fifth version of the *Lady's Glove* [15]

The more recent versions of the *Lady's Glove* (fourth and fifth) were developed by Sonami and Bert Bongers in 1994 and 2004 [13]. The fourth and fifth versions of the *Lady's Gloves* are shown in Figures 5 and 6. These versions include additional sensors, such as a pressure sensor and flex sensors for the fingers [16]. The Hall Effect transducers and magnet are still utilized, but the magnet was moved onto the same glove as the transducers [16]. Therefore, the glove can be played with just one hand rather than two. Other mechanisms, such as a mercury switch, were implemented to provide more control over performances [16]. The sensors output signals to the *SensorLab,* which is able to convert the signals into MIDI format that can then be processed by software (*MAX-MSP*) on the computer [16]. Since the *Lady's Gloves* utilize a multitude of sensors, it requires experience to be able to use it effectively. Along with experience on how the sensors are utilized by the hand movements, one must also be familiar with the specific software required to associate the sensors to a function.

2.2.4 *Vocal Augmentation and Manipulation Prosthesis (VAMP)*

Figure 7: VAMP [17]

Another glove and gesture-based music controller is the *Vocal Augmentation and Manipulation Prosthesis (VAMP),* which was developed by Elena Jessop of the MIT Media Lab [18]. According to Jessop, the concept of *VAMP* is to be able to use it to complement the user's vocals during live performances, such as opera [18]. The *VAMP* consists of various sensors, such as flex sensors, pressure sensors, and an accelerometer [18]. Together, these sensors allow the controller to perform various functions during live performances in a fluid manner without

impeding the flow of the performance. With the use of software, the signals from the sensors, as well as the user's voice are processed and interpreted [18]. The VAMP allows the user to sample and hold the user's voice as long as the gesture is held, which is to form a pinching gesture with the thumb and index finger [18]. Therefore, the user, for instance, would be able to record and sustain a note while the user sings another note. The user is also able to pulse the sustained note through movement of the arm that is sensed by the accelerometer and control the amplitude of the recorded note based on the angle of the elbow [18]. The *VAMP* does not appear to be excessively complex and could be picked up quickly since the implemented gestures appear natural and intuitive [18]. For example, the gesture of pinching together the thumb and index finger does indeed suggest a concept of sustaining a note.

2.2.5 *Mi.Mu Gloves*

Similarly to the *Lady's Glove* and *VAMP*, Imogen Heap, along with her team, developed the *Mi.Mu Gloves*, which, too, also incorporates the gesture-based concept with the use of a glove [19]. Her concept is that she wanted a method to perform music in a fluid and expressive manner [20]. She expresses this concept in her demonstration at *Wired Talk 2012* with a wireless piano keyboard, where she explains that the audience would not be able to see what she is doing and would find the playing of a simple keyboard as "unexciting" [20]. Therefore, the *Mi.MU Gloves* provide both the ability to produce music in a unique way, as well as allowing the musician to entertain the audience visually.

Figure 8: Picture of *Mi.Mu Gloves* [21]

Figure 9: Imogen Heap using the *Mi.Mu* Gloves [22]

Similarly to Waisvisz's *the Hands*, the *Mi.Mu Gloves* use both hands to fully utilize the music controller. Figure 8 shows the *Mi.Mu Gloves*, while Figure 9 depicts Imogen Heap using the *Mi.Mu Gloves* and wireless equipment that is used along with the gloves. The *Mi.Mu Gloves* utilizes the *Data Glove 14 Ultra* developed by *Fifth Dimension Technologies (5DT)* as a foundation [23]. The *Data Glove* consists of sensors that produce signals based on the bend of the finger, where there are two sensors for each finger [24]. The *Mi.Mu Gloves* also uses inertial sensors, such as accelerometers and gyroscopes, which are provided by the X-IMU board [25]. Additionally, sensors were implemented onto a "hi-tech suit" to be used along with the glove [23]. Finally, the *Mi.Mu Gloves* also utilizes the *Microsoft Kinect*, which is simply a motion detector, which in this application, senses where the user is in a particular area [26]. Based on location, there are "different effects and layers" added on to the produced music [26]. Imogen Heap's gloves also provide feedback in the form of light, indicating which mode the gloves are currently in [20]. For example, Heap's demonstration at *Wired Talk* indicated that when the light emitting diode (LED) emits red, it indicates that the gloves are in "recording mode", which allows Heap to record her voice and be played back and manipulated with the gloves [20]. Through gestures, the gloves are able to perform a variety of functions where playing the drums, recording, and making bell sounds are a few examples as seen in her demonstration [20]. Heap also mentions that the user is freely able to map the gestures through the use of software [20]. For software, Heap states that there are three custom software programs that were developed to interpret data from the gloves, the *Kinect*, and data in terms of MIDI [23]. The *Mi.Mu Gloves* also appear to be quite complicated for the casual player due to the large amount of functionality. However, once one is familiarized with the

gloves, one is able to perform with the gloves in a very effective, fluid, as well as entertaining manner.

2.2.6 *Leap Motion Controller*

Figure 10: *Leap Motion Controller* [27]

The *Leap Motion Controller* is a gesture-based motion controller, which utilizes an "AppStore" concept, allowing developers to publish their applications that can be sold to consumers [28]. With the *Leap Motion* Controller, users are able to play music through hand movements. The *Leap Motion Controller* captures hand movements through the use of infrared optics and cameras [29]. Although this device is not solely a musical device, such capabilities are enabled by multiple applications available from the "AppStore". One of which is the *Fingertapps Piano*, developed by *Fingertapps*. This application allows the user to play a virtual piano through hand gestures that mirror a real piano [30]. Another is the *Chordion Conductor*, which allows one to perform various functions, such as using gestures to control a synthesizer or arpeggiator [30]. It also allows for gestures to control volume, pitch-bend, and other parameters [30]. There are also other applications that complement musical instruments or MIDI controllers. For example, the *Geco MIDI* application allows users to apply effects through gestures [31]. This application is able to interface with a MIDI compatible device and send data to a DAW through a virtual MIDI port [31].

Although the *Leap Motion Controller* is intriguing and powerful, it does have some restrictions. For example, it is limited to its sensing area, where the user must be within this area

to be able to operate it. There are also concerns of one's arm becoming strained due to the requirement of hovering one's hand above the controller. Users have also voiced their concerns on the software compatibility with where the experience varied from application to application [29]. For example, one may find the controller to be quite accurate in sensing hand gestures in one application, however, in another application, the movement of fingers would have difficulty being detected.

2.3 MIDI

The aforementioned devices, excluding the *Theremin* and the *Leap Motion*, are considered MIDI controllers due to their MIDI capability. MIDI (Music Instrument Digital Interface) is a standard digital protocol that allows electronic devices (musical instruments, computers, etc.) or software to communicate with each other using music information to generate a particular sound. Its original purpose when it was created was to allow musicians to be able to control multiple electronic instruments from one device. When a key is pressed on a MIDI keyboard, it generates MIDI data to share parameters, such as which note was pressed, how hard it was pressed, and how long the note was held. These messages can be sent to other instruments (computer, drum machine, or sampler) to communicate which note was played and how it was played, using MIDI commands. With the use of a DAW (sequencer), users have the ability to record and edit MIDI data, to create electronic music. Multi-track recording software's are also called sequencers because the MIDI events are recorded in a time-based sequence, which are then played back in the same order. The ability to edit and manipulate digital MIDI data is made simplistic in sequencers. The user is allowed to copy and paste audio for more than one track at a time in a straight-forward chronological display from left to right. Also sequencers have virtual faders, knobs, and effects for each instrument, or track, to control parameters such as volume and pan, to produce a good mix between instruments. [32] Hence, sequencers allow users to become producers of music, as well, by providing them a virtual mixer for their home recordings.

There are two types of devices that send MIDI data: MIDI Instruments (synthesizers) and MIDI controllers. A synthesizer is composed of two components: the internal sound module and the controller, such as a keyboard. Therefore, a synthesizer has the functionality of a MIDI controller, however, its only difference is that it contains an internal sound module that generates

sound. MIDI controllers generate and transmit MIDI data, however, they need an external device to produce sound. Through a MIDI connection with a synthesizer or computer, sound will be generated. An example of a synthesizer would be a basic electronic piano that comes with numerous preset instruments sounds or effects, that when a key is hit, you hear a tone. Some MIDI controllers come with particular faders and knobs to manipulate the sound that is being made. These devices are not equipped with preloaded sounds or effects because their sound is generated through third-party hardware or software. The most popular model for a synthesizer or MIDI controller is the keyboard, however guitars, wind instruments, and drums have also been developed [32].

MIDI messages are sent and interpreted for each respective MIDI compatible device by utilizing a particular message format, in binary data. There are two types of bytes that distinguish a MIDI message: the status byte and the data byte. Status bytes always begin with 1, while data bytes begin with 0. This leaves the remaining 7 bits per byte to denote the message (128 possible values). Status bytes are represented in the form: 1sssnnnn. Here, the 3 bits for "s" denote the type of message being sent (channel voice message), while the 4 bits for "n" distinguish the channel number to which the message is applied [33]. The channel voice message can be a variety of options. Examples of such messages are if the note on the device is pressed (Note On), if the note is not pressed (Note Off), or when a note is applied with more pressure, after being pressed, to control a variety of effects or parameters, a concept known as aftertouch (Poly Key Pressure). Through one MIDI link, a MIDI device can receive MIDI messages through a range of 16 MIDI channel numbers. The particular voice of a device will only respond to the channel that it is tuned to and ignores all other channel messages. This is similar to a how a television or radio receives only the station it is tuned to, while rejecting the others [34]. Data bytes are represented in the form: 0xxxxxxx. Data messages are an extension of the status messages to indicate which additional parameter was modified, in association to the respective status message. For example, the Note-On status message has two data bytes associated to the parameters of the note number and note velocity. As mentioned earlier, like status bytes, data bytes denote values from a range of 0 – 127 values in binary (128 possible values). Thus, the note number data byte covers the full range of note values of the chromatic western music scale (C0 – 0, G10 – 127) [35]. Figure 11

displays this range, represented in MIDI, using a keyboard (instead of beginning on C0, it starts on C-2):

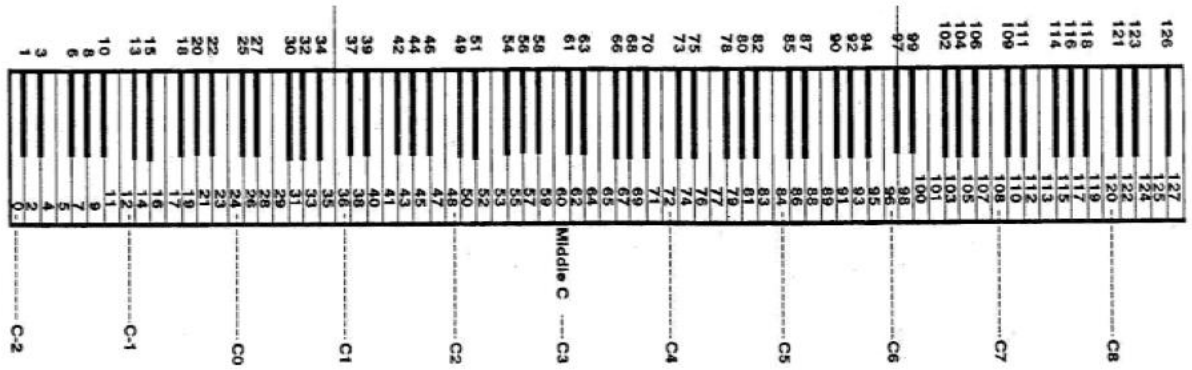

Figure 11: MIDI Representation of Chromatic Western Music Scale [36]

Therefore, to represent that the note for middle C (C5) is pressed, on MIDI channel #3, with full velocity, in MIDI code, perform the following method:

- Status Byte (always begins with 1)
    - Type of Message (s)
        - Note On = 000
    - Channel Number (n)
        - 3 = 0011
- Data Byte (always begins with 0)
    - Parameter Association with Note On status – Note Number and Velocity
        - Note Number
            - C5 = 60 (0111100) [35]
        - Velocity
            - Max = 127 (1111111)

The table below displays the final status and data bytes for the procedure above:

Table 1: Example of MIDI Message

Status Byte (Note On)	Data Byte (Note Number)	Data Byte (Velocity)
10000011	00111100	01111111

2.4 Competition

There are two MIDI trumpet controllers that have been developed on the market: the *Morrison Digital Trumpet (MDT)* and the *Yamaha EZ-TP Trumpet*. A typical trumpet requires the player to blow into the mouthpiece with their lips closed to create a buzzing sound. This pitch can be altered by the embouchure, which is the lip tightness and aperture, of the performer. Rather than using embouchure, the *MDT* utilizes the concept of breathing into the mouthpiece. A pressure sensor measures the movement of your breath and converts it into an electric signal. The use of one's breath while playing allows for more expression compared to other MIDI controllers. The *MDT* follows a typical trumpet's approach of three valves to form desired pitches. However, the *MDT* has a range of up to ten octaves, compared to a typical trumpet, which has only three. Also, unlike a regular trumpet, the *MDT* has a vibrato lever [37]. The *MDT* sells for $2995.

Figure 12: The *MDT* [38]

The *Yamaha EZ-TP Trumpet* shares many qualities to the *MDT*. It does not have the same advanced capabilities, such as the ten octaves range or vibrato lever, but follows the same basic functionality of valves and pitch variation. Perhaps the most significant difference of all is the method of input from the user. Unlike the *MDT*, which required the performer to breath into the mouthpiece, the *Yamaha EZ-TP Trumpet* requires the user to whistle or hum into the mouthpiece. This device also has a built-in speaker and is a music-learning device for trumpet players [39]. Utilizing the "singing play" mode, the user has the ability to hum any melody into the mouthpiece, and the device is able to demonstrate how to play the given melody by lighting the appropriate valves. The *Yamaha EZ-TP Trumpet* is currently being discontinued by *Yamaha*, but can be found on eBay for prices as low as $390.

Figure 13: *Yamaha EZ-TP Trumpet* [40]

3. Product Requirements

Based on market research, the advantages and disadvantages were determined for several products that are associated with interactive hand gesture-based MIDI controllers. From this analysis, the requirements of the customer are determined:

- Easy to use
- Plays like a trumpet
- Usefulness
- Affordable
- Portable

Easy to use: The resulting product should be intuitive to play for the consumers, which are both novice and experienced trumpet players. Therefore, the MIDI controller must incorporate the technique of how trumpets are played in order to provide this sense of familiarity.

Plays like a trumpet: The device will attempt to both emulate a real trumpet and incorporate MIDI parameters to distinguish it as a MIDI controller. The initial design will focus strictly on replicating the basics of a trumpet on a MIDI controller, depending on the complexity of the project. This includes the valves fingering, embouchure register, and wind pressure blown into the instrument. Also, this design will include the same pitch range as a trumpet. However, if time permits, additional features will be added to enhance the device from both a realistic trumpet and MIDI controller perspective. Additional features to enhance the real trumpet aspects include: the ability to create different effects, such as vibrato and tremolo, as well as being able to use different techniques to play, such as "half-valving" creating a glissando effect. Additional features to enhance the MIDI controller aspects include: incorporating MIDI parameters, such as pitch bend, and expanding the amount of octaves for a trumpet.

Usefulness: The purpose of this device is to allow trumpet players to use a MIDI controller, without having to learn how to play a typical keyboard MIDI controller. With this capability, trumpet players will be able to utilize their favorite DAW to record or play music, using virtual instruments.

Affordable: Current MIDI trumpet controllers are the *MDT Trumpet* and the *Yamaha EZ-TP Trumpet*. As explained in the market research, the *MDT Trumpet* costs $2995, which was determined to be too expensive for consumers. With such a high price, novice players are excluded.

Although the *Yamaha EZ-TP* is sold for as low as $390 on eBay, *Yamaha* is discontinuing this product. Therefore, the resulting MIDI controller should aim to be around the cost of the *Yamaha EZ-TP*.

Portable: Convenience would allow users to be able to transport their MIDI controllers with ease.

4. Methodology and Initial Design Concept

The team performed initial research to determine the scope of the project. After research, the team commenced the design phase of hardware and software. Implementation was performed through utilizing a hardware prototype and a software development environment. Testing was conducted after every subtask to ensure that each module was functioning as intended. The team also considered the possibility of incorporating additional features. As a result the design was formulated to support this possibility. The purpose of the device is to create a MIDI controller for trumpet players that utilizes a glove-based design that is akin to playing a trumpet.

From initial research, the resulting MIDI controller should utilize sensors attached to the gloves that can interpret human gestures. A microcontroller is required to interpret the voltage signal from the sensors and transmit it to a DAW through the MIDI protocol. The DAW produces the sound based on the respective MIDI input. The microcontroller enclosure will likely be mounted on a belt attached to the user's backside waist. A USB cable connects the device to the computer and should not impede performance, since the device will not interfere with the playing hands. The block diagram depicted in Figure 14 shows the initial concept of the MIDI controller.

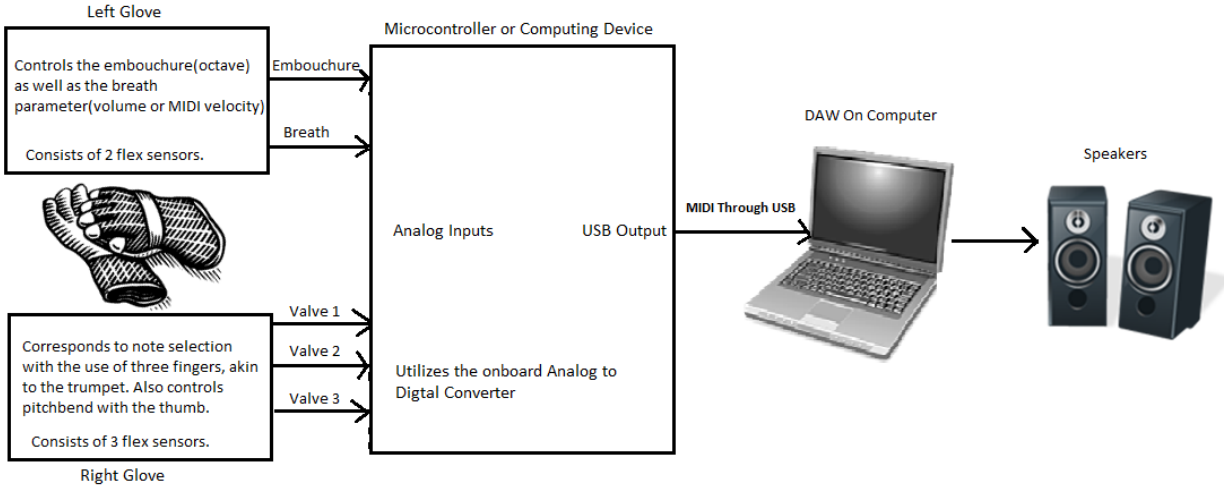

Figure 14: Block Diagram for *SansTrumpet*

The capabilities of this MIDI controller compared to a typical trumpet are shown below:

Real Trumpet

- Sound generated internally
- Combination of embouchure (from lip tightness) and valve fingering (3 valves) determines the note played
- Wind pressure from breath controls how loud or soft the note is
- 4 Tuning Slides to tune the trumpet
- Techniques to create unique sounds by using the mouth or lips
    - Tonguing
    - Growling
    - Lip Trill
- "Half Valving"
    - Technique used by trumpet players to slide between notes when the particular valve (s) is not pressed down completely, creating a glissando effect
- Tremolo effect
    - Because a certain note can have several different valve combinations on a trumpet, alternating between these combinations, while playing, results in tremolo
- Vibrato Effect
    - Executed by either gently rocking your fingers on a valve, jaw motion, or breath (pulsating your stomach or diaphram muscles)
        - The speed that each of the following tasks is accomplished, directly effects the speed of the vibrato

***SansTrumpet* MIDI Controller**

- Sound generated externally (speaker)
- Same pitch range as trumpet (represented in MIDI as F#3 to C6)
- Right Hand
    - Valve Fingering
- Left Hand
    - Embouchure Register

- Dependent on how index finger sensor is bent
- Each emboushure level will contain a certain register of MIDI notes, which never have an instance of the same fingering – 6 levels

- Thumb Trigger
  - Determines when the note is played and its respective volume
  - Accomplished by moving the thumb down until the desired level, at which time it is brought up to play the note

The basic technique of playing a trumpet had to be researched in order to determine functionality of the device. The formation of a note involves two factors: the valves being pressed (valve fingering) and the lip tightness of the trumpet player (embouchure). These two parameters establish the particular note that is being played. To comprehend the proper embouchure registers for each fingering of the standard pitch range of a trumpet, Figure 15a was followed:

	C	C#	D	D#	E	F	F#	G	G#	A	A#	B		Fingerings	
3							165	175	185	196	208	220		Open	13
4	233	247	262	277	294	311	330	349	370	392	415	440		1	23
5	466	494	523	554	587	622	659	698	740	784	831	880		2	123
6	988													12	

Figure 15a: Valve Fingering/Embouchure Levels for Trumpet [41]

Figure 15a was created by a team of students at Cornell University for a project, which pertained to a MIDI controller that utilized an actual trumpet. As can be seen, the appropriate valve fingering and embochure level combinations are shown, where each are distinguished respectively as color and frequency. The embouchure level, here, was determined by a piezoelectric transducer, which measured breath sensing in frequency (Hz) [42]. These frequency values are not applicable to this device, though, because breath sensing is not required for the device. However, instead of making the correct frequency with your breath to play a note, the device will incorporate certain pitch registers that do not have instances of the same valve fingering to demonstrate embouchure level. The concept is to vary each embouchure register by the bend of one's index finger to assign certain voltage ranges (by the flex sensor) to the desired embouchure register. MIDI note numbers span from 0 to 127, where 0 represents pitch, C0, and 127 represents pitch, G10. Thus, the

following MIDI pitch registers, associated with voltage ranges by the flex sensors (shown below), were developed:

- Embouchure Register 1: Voltage Range = 1.18 V to 1.39 V, Pitch Register = F#3 to C4 (MIDI Note = 42 to 48)
- Embouchure Register 2: Voltage Range = 1.39 V to 1.60 V, Pitch Register = C#4 to G4 (MIDI Note = 49 to 55)
- Embouchure Register 3: Voltage Range = 1.60 V to 1.81 V, Pitch Register = G#4 to C5 (MIDI Note = 56 to 60)
- Embouchure Register 4: Voltage Range = 1.81 V to 2.02 V, Pitch Register = C#5 to E5 (MIDI Note = 61 to 64)
- Embouchure Register 5: Voltage Range = 2.02 V to 2.23 V, Pitch Register = F5 to A5 (MIDI Note = 65 to 69)
- Embouchure Register 6: Voltage Range = 2.23 V to 2.43+ V, Pitch Register = A#5 to C6 (MIDI Note = 70 to 72)

Based upon the chart shown in Figure 15a and our Embouchure ranges, the following chart in Figure 15b, was constructed to better depict the relationship between fingering combination and the embouchure register.

Embouchure Register	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
1	C4						F#3	G3	G#3	A3	A#3	B3
2		C#4	D4	D#4	E4	F4	F#4	G4				
3	C5								G#4	A4	A#4	B4
4		C#5	D5	D#5	E5							
5						F5	F#5	G5	G#5	A5		
6	C6										A#5	B5
VALVE FINGERING												
Open												
1												
2												
12												
13												
23												
123												

Figure 15b: *SansTrumpet* Embouchure Table

The thumb trigger finger will effectively determine when a note is played, as well as its corresponding volume. The playing technique involves swooping the thumb down to a certain position and then bringing it back up. This technique was determined because the device has no "bottom-out" like a keyboard for each note that determines how a key is played. With a keyboard, the player can gauge the way a note is played by the manner in which the key is pressed. Because the device is being played in air and nothing is being pressed, this method allows for the player to determine the way the pitch is to be played by the bend of their thumb. Therefore, the manner in which the note is to be played will be pre-determined by a concept known as the amplitude envelope, which will be discussed later. In terms of the corresponding voltage signal of the thumb sensor, certain voltage ranges were assigned particular volumes, in a similar fashion to how the embouchure registers were determined. MIDI velocity ranges from 0 to 127, where 0 represents that softest a pitch can be played (or not being played at all), while 127 represents the loudest a pitch can be played. Thus, the values for volume (MIDI velocity), related to the voltage ranges by the flex sensors, for each pitch were assembled:

- Trigger Level 1: Voltage Range = 1.18 V to 1.31 V, Volume = 12
- Trigger Level 2: Voltage Range = 1.31 V to 1.44 V, Volume = 24
- Trigger Level 3: Voltage Range = 1.44 V to 1.57 V, Volume = 36
- Trigger Level 4: Voltage Range = 1.57 V to 1.70 V, Volume = 48
- Trigger Level 5: Voltage Range = 1.70 V to 1.83 V, Volume = 60
- Trigger Level 6: Voltage Range = 1.83 V to 1.96 V, Volume = 72
- Trigger Level 7: Voltage Range = 1.96 V to 2.09 V, Volume = 84
- Trigger Level 8: Voltage Range = 2.09 V to 2.22 V, Volume = 96
- Trigger Level 9: Voltage Range = 2.22 V to 2.35 V, Volume = 108
- Trigger Level 10: Voltage Range = 2.35 V to 2.43+ V, Volume = 120

Currently, this a theoretical representation of the trigger levels to demonstrate the manner in how each will be implemented. However, once the voltage values for the analog input circuit are measured, there will be multiple trigger levels to allow for a wide range of volumes to be displayed by the thumb position.

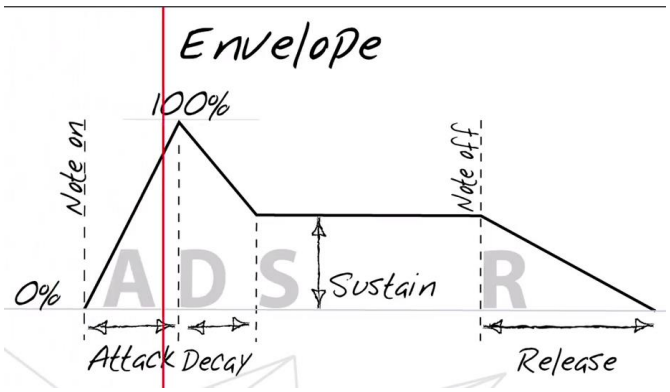

Figure 16: Amplitude Envelope [44]

The concept of amplitude envelope must be considered for the left hand thumb operation. The sound of the pitch should not just simply commence and stop abruptly when a note is triggered on and off. The volume of the pitch should vary depending on how hard and how long the note is pressed. This fluctuation in volume is represented by the concept of the amplitude envelope. The amplitude envelope entails how the sound's amplitude, in this case, volume, develops over time [43]. Because the functionality of the sensor on this thumb is to control the volume of how loud/soft the note is played, the voltage signal must follow a waveform that is similar to the actual waveform of playing a trumpet [43]. The waveform of the amplitude envelope is displayed in Figure 16.

The amplitude envelope begins with the attack of the note, followed by a decay. Here, the voltage signal reaches a peak value depending on the extent to which the note was hit (attack) and is decreased to a particular value, depending on how slowly the note loses its volume after it is played (decay). After, the note is sustained for a particular time period (sustain), and ends with the cutoff and decay when the note is off (release) [43]. In terms of replicating this concept on the device, the software must read the trigger level that the thumb is located in and incorporate it in a formula. As discussed earlier, each trigger level has a particular volume corresponding to it. Therefore, depending on the trigger level, the software will produce numerous Note-On messages to form the waveform of the amplitude envelope. Upon release, a Note-Off message will be triggered. In this fashion, the attack of the amplitude envelope will change depending on the trigger level, however, the overall formation of the amplitude envelope will remain consistent. Each trigger level will have its own unique time parameters (attack, decay, sustain, release) that will be modified to represent the appropriate sound.

The initial design concept utilizes flex sensors and a microcontroller/computing device, which make up the MIDI controller. The data sent from the controller is interpreted by the DAW on a computer, in which it then produces sounds/notes corresponding to the provided data through the speaker. The idea is that the user would play the MIDI controller similarly to that of the trumpet. The flex sensors would vary the voltage signal depending on the strain of the sensor. Code would need to be developed and loaded onto the microcontroller. The developed code would utilize the Analog to Digital Converter (ADC) on the board to interpret the voltage signals as binary data. The code would also associate notes to the digital representation of the voltage signals. The microcontroller can then output the data by using the MIDI protocol. Due to the implementation of MIDI data through a USB connection, the Baud rate or transmission speed of data must be considered. A possible issue pertaining to the transmission speed is that the MIDI protocol standard uses a Baud Rate of 31,250 bits per second. If the data is sent at a faster rate than the MIDI standard, potential data may be lost due to the transmission limit. Such data refers to the generated MIDI messages based on the user's finger movements. If too many messages are sent in a given time, only a select few will be interpretted due to the transmission limit. Therefore, the development of the software for the *SansTrumpet* must include a system to achieve the MIDI Baud rate. A possible method could be implemented through the use of software delays.

To model the conversion of analog voltage signal from the glove to MIDI data, using a microcontroller, a initial concept flow chart of software functionality and an example diagram is shown below:

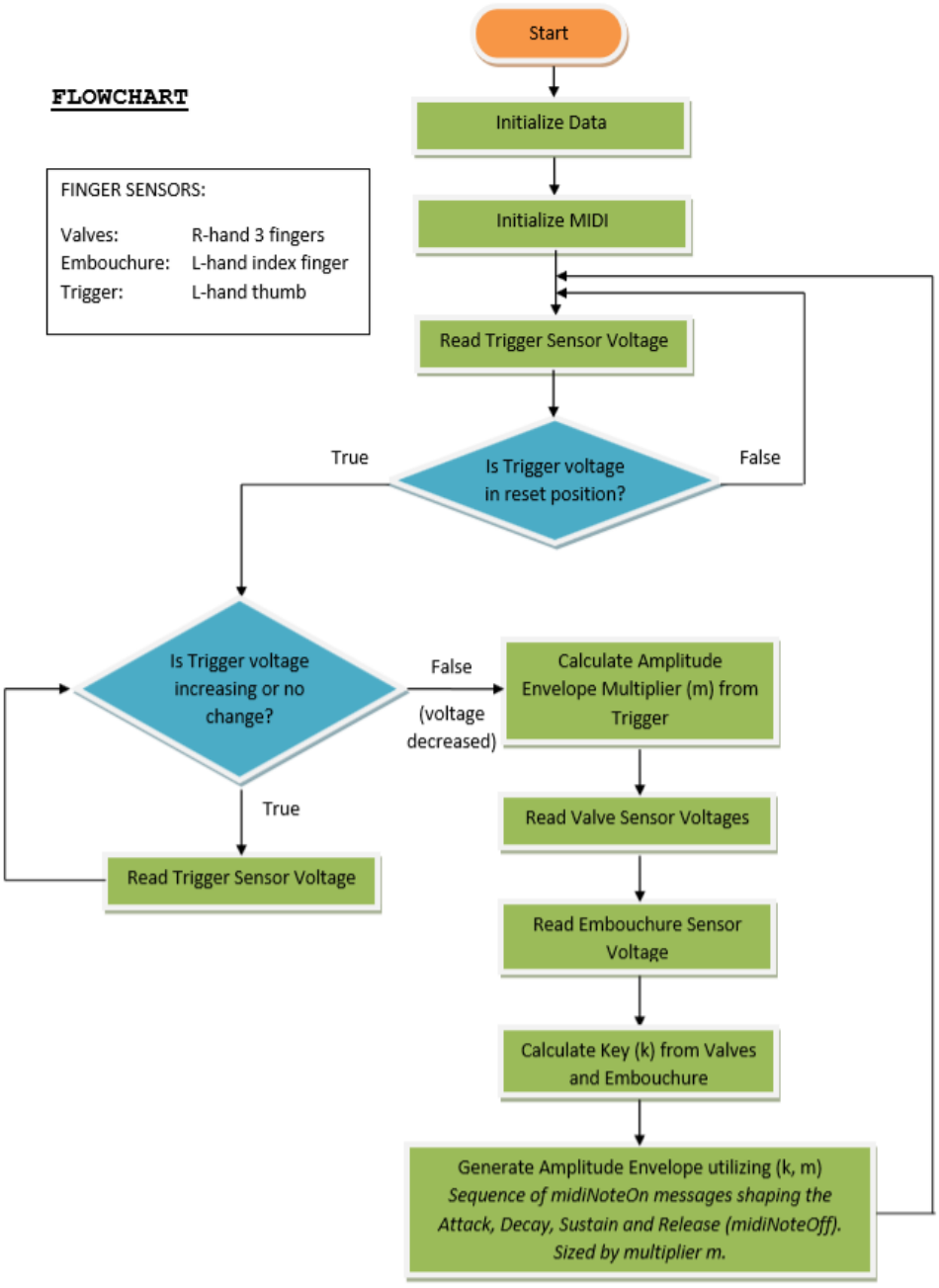

Figure 17: Initial Concept Flow Chart of Software Functionality

The chronology of software functionality is demonstrated in Figure 17. Upon starting the device, all parameter values are initialized, which include the embouchure registers, valves pressed, and trigger volume. The first parameter the program reads is the trigger sensor (left thumb) and checks if the sensor is in reset position. It will loop until the trigger sensor is in this state. When it has, the software will read if the trigger sensor voltage either keeps increasing or if no change in

voltage occurs. This block will continue to loop until this voltage decreases. When the voltage does decrease, the program calculates the amplitude envelope multiplier (trigger level) from the voltage signal of where the decrease occurs. Next, the software reads the valve sensor and embouchure register sensors, in order to calculate the desired pitch. The note number (determined by the valve and embouchure sensors) and velocity (determined by the trigger sensor) allows a MIDI Note-On message to be sent. Consequently, the software creates the amplitude envelope for the volume by sending various Note-On messages to form the waveform of the amplitude envelope, which is sized by the multiplier, as explained above. Finally, the software returns to the loop of waiting for the trigger sensor to be in reset position, and thus, the process is repeated. Therefore, if the user wants to play a new note, they must move the trigger thumb to reset position each time they want to do so.

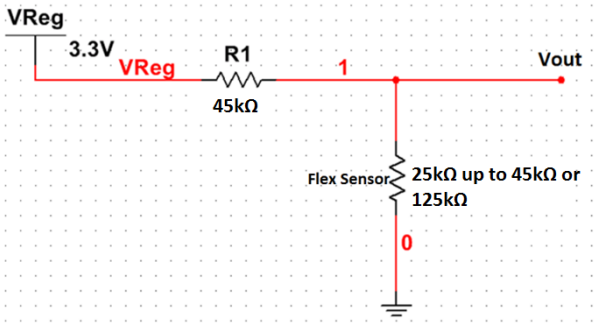

Figure 18a: One voltage divider branch for a flex sensor

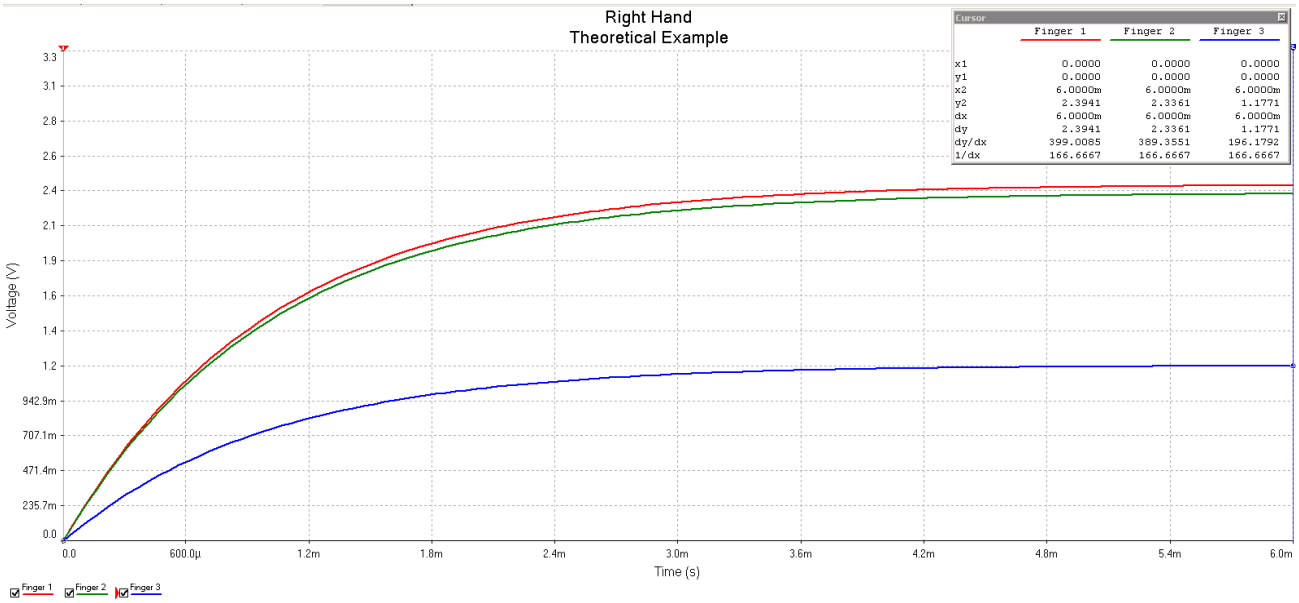

Figure 18b: Theoretical plot of generated voltage signals for right hand

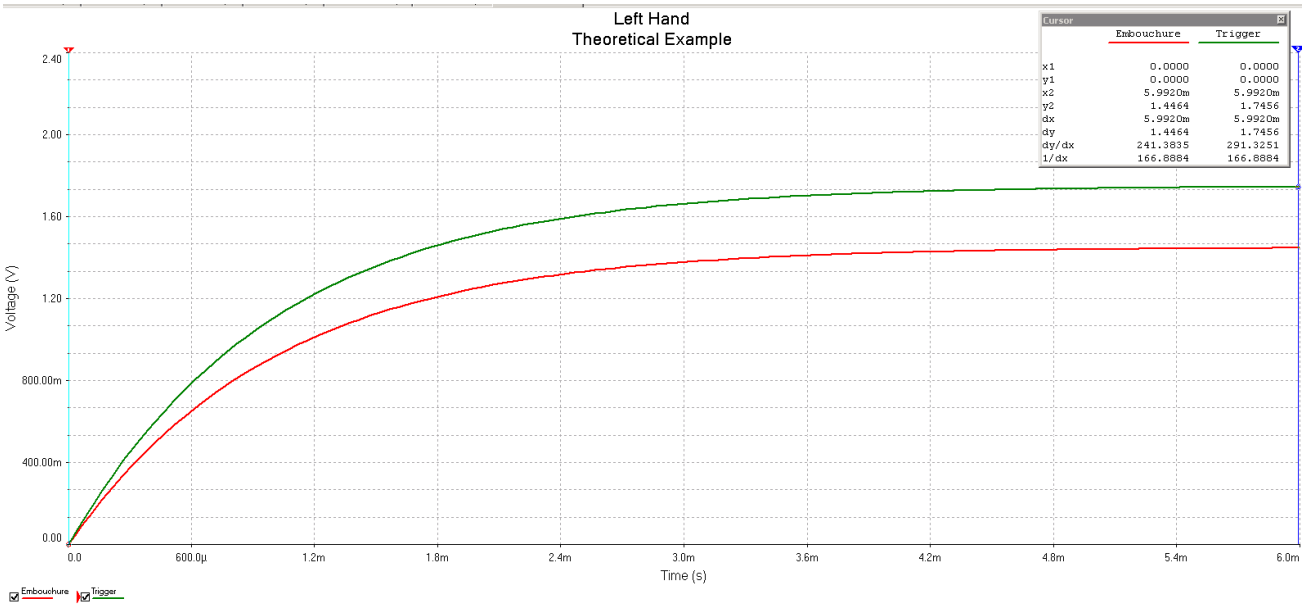

Figure 18c: Theoretical plot of generated voltage signals for left hand

In order to convey the initial concept of the *SansTrumpet*, a theoretical example of how the bend of fingers is interpreted as MIDI data. Figures 18b and 18c displays a theoretical instance where analog voltage signals were produced from the bending of the left and right hand fingers. These plots were constructed with the use of *MultiSim* where an exponential source and voltage dividers were used. An exponential curve was used in the theoretical example due to the fact that it can approximate the representation a basic bend of the finger. This is based on the idea that the user would bend their finger until they reach the desired bend angle and would maintain their finger at that position. As the user bends their finger, the flex sensor resistance steadily increases. Due to its implementation with the voltage divider concept, the voltage reading at the microcontroller input will also steadily increase. Once the user reaches the desired position, the finger is held at that position and therefore the voltage is maintained at a near constant value. This characteristic is observed in both Figures 18b and 18c.

Figure 18a shows the basic configuration of a flex sensor and a resistor to form a voltage divider, which is the basis of the analog input circuitry. Figure 18b shows a theoretical example of the right hand where the red, green, and blue lines represent the voltage generated by fingers 1, 2, and 3, respectively. The assumed ranges of resistor values for the flex sensors, in accordance with the amount of strain applied, was 25kΩ (0 degrees) to 45kΩ(45 degrees) to 125kΩ (90 degrees –

Max). A voltage divider was integrated and a 45kΩ resistor was initially utilized with the resistance values of the flex sensors. However this resistor value may change depending on actual resistance measurements of the flex sensors. It was determined from the voltage divider concept that the voltage signal created by each resistor value was in the range of 1.18V to 2.43V. For this initial concept example, it is assumed that any voltage signal below or equal to 1.18V will be interpreted as "unbent" while if readings are 2.43V or above, they will be interpreted as "bent". For the example, two of the three fingers on the right hand are bending where the 1st and 2nd flex sensors produces two voltage signals near max voltage of approximately 2.43V as seen in Figure 18b. The third finger of the right hand is not bent and therefore produces a voltage of approximately 1.18V. As mentioned, a voltage below or equal to 1.18V is considered as "unbent", which represents an unpressed valve. Therefore, the right hand produces a total of three voltage signals, where only two are considered as pressed valves and one as an unpressed valve .

With these two valves pressed, consider that the embouchure finger of the left hand is bent slightly creating a voltage signal in the vincinity between 1.18V to 1.65V, near 1.45V as shown in Figure 18c**.** The red line represents the embouchure finger, while the green line represents the trigger finger, which is the thumb. A voltage of 1.45V is within the range of 1.39V to 1.60V. This range was desginated to the second set of frequencies pertaining to the notes from C#4 to G4, as described earlier. A voltage of 1.45V and considering the right hand finger (valve) combinations, this would result in a MIDI pitch D4. Also consider that the thumb of the left hand is bent to such a degree that the resulting voltage is 1.75V. 1.75V is within the range of 1.70V to 1.83V, which would result in a MIDI velocity value of 60. A voltage as analog data will be converted into binary data. In 13-bit resolution, the voltage range of the right hand is represented from 0.47V to 2.14V as 1167 to 5312. From the signals given, the appropriate MIDI message can be sent:

- Status Byte
    - Type of Message (s)
        - Note On = 000
    - Channel Number (n)
        - 1 = 0001
- Data Byte
    - Parameter Association with Note On status – Note Number and Velocity

- Note Number
    - D4 = 50 (00110010) [5]
- Velocity
    - Max = 60 (01000000)

Thus, it can be determined that the MIDI message is: 10000001 00111001 00111100

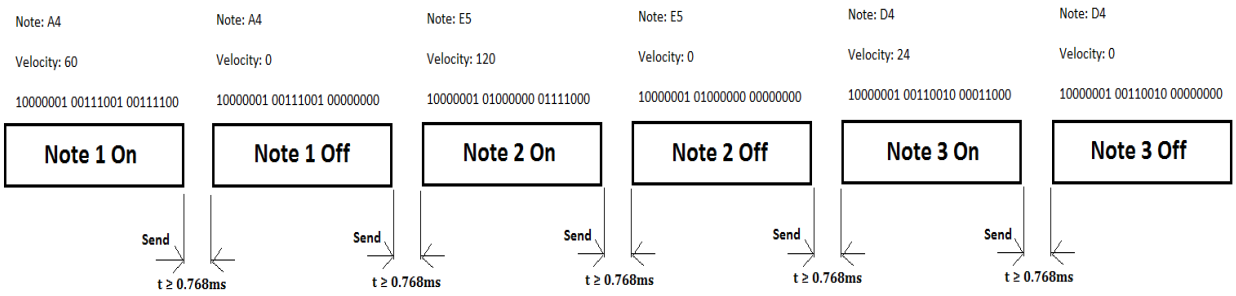

Figure 19: Example of a Chain of MIDI Messages in Succession

Figure 19 depicts a chain of MIDI messages for an instance when a user plays 3 notes in succession. Each block shown in the figure refers to one MIDI message (either a NoteOn or a NoteOff). In this case, the user begins by playing note A4 at medium velocity (volume), which is 60, as seen in the Figure 19. For note A4, fingers 1 and 2 of the right hand are bent to a reasonable degree, while finger 3 is unbent. Additionally, the embouchure finger on the left hand index finger is bent to approximately 50 degrees. This would result in the 3rd embouchure register based on the designated range, as described previously. The trigger finger, which is the left hand thumb, is bent to 45 degrees resulting in the 5th trigger level. Once the trigger thumb returns to the reset position, the note will be transmitted and played. The user is able to freely select the desired pitch combination (valve and embouchure register) until the trigger thumb is in the process of returning to the reset position. As described earlier, a MIDI message consists of a status byte and two data bytes. The status byte is the first byte of the MIDI message; in this case the status byte indicates NoteOn at channel number 1. The two status bytes refer to the note number and velocity values, which are represented in binary. A NoteOff MIDI message is then sent once the note has played for its duration. The user then decides to play the note E5, using a high velocity of value 120. Therefore, the user transitions the fingerings to the appropriate positions for the desired note. For the note E5, fingers 1, 2, and 3, of the right hand are all unbent representing the open position,

which implies that no valves are pressed down. Furthermore, the embouchure finger is bent to approximately 70 degrees. The trigger finger is bent to approximately 85 to 90 degrees for a velocity of 120. Similarly, a NoteOff MIDI message is sent to turn the note off. Lastly, the user bends fingers 1 and 3 of the right hand, while the embouchure finger is bent to approximately 20 degrees; this results in the note of D4. The trigger thumb is bent slightly resulting in a velocity of 24. This process is repeated for each note that is played. Each MIDI message is 24 bits in length. Based on the MIDI baud rate of 31,250 bits per second, the theoretical time for each MIDI message to be sent was calculated to be 0.768ms. Therefore, the minimum time between each MIDI message must be at least 0.768ms. However, time between each MIDI message includes the transmission time and the note duration time. Additionally, it includes the time that the user takes to trigger a note. Thus, the resulting time between each message would be greater than 0.768ms depending on the note duration, as well as the user.

5. Circuit Design and Development

Based upon the initial design concepts, the circuitry of the *SansTrumpet* was developed. The team selected and implemented components that were determined to be the most appropriate in exemplifying the design model. The purpose of this section is to document the selection process of the components as well as the development process of the *SansTrumpet* circuit.

5.1 Initial Circuit Component Selection

5.1.1 Flex Sensors

For sensors, bend/flex sensors were determined to be appropriate for this device. By using flex sensors on a pair of gloves, the user is able to play the MIDI trumpet controller with a very similar technique to that of playing an actual trumpet by using simple finger movements. The concept of the instrument is that with the left hand the user is able to control lip placement, or embouchure, which allows for the selection of pitch register. The left hand would also allow the user to control volume (velocity parameter in MIDI). The right hand allows the user to control the valves being pressed, akin to the trumpet. The left hand glove can be modeled using two bend/flex sensors, when bent, selects the appropriate embouchure register. The right hand glove is modeled using three flex sensors to represent each valve on a trumpet. The chosen flex sensor is developed by *Spectra Symbol*, depicted in Figure 20.

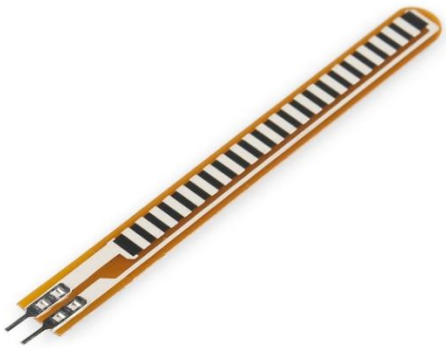

Figure 20: *Spectra Symbol* Flex sensor [45]

The team chose this flex sensor by checking the catalog from known distributors, such as *Jameco, DigiKey,* and *SparkFun*. The *Spectra Symbol* flex sensor was chosen due to the fact that it was more widely used compared to others. The *Spectra Symbol* flex sensor is offered in two variants, the 4.5" and 2.2" versions. The team had initially chosen the 4.5" version due to the fact that the flex sensor functions optimally when the bent is radial. This means that the flex sensor should be bent in a curving method, rather than in a sharp manner. The 2.2" flex sensor is sufficient in spanding over the length of the joint area, however, the team believes that it may be possible that it is not long enough to allow for the curvature. The team will purchase 1 of each sensor to test and determine whether the 2.2" or 4.5" versions will be appropriate for this design. Flex sensors are viewed as variable resistors that vary in resistance depending on the angle of its bend. For the chosen flex sensor specifically, the nominal resistance is 25kΩ and can vary up to 45kΩ to 125kΩ [46].

5.1.2 Microcontrollers/Computing Devices

For the "microcontroller/computing device" block, there are a variety of options that are available on the market. In order to narrow down the options, a select few were chosen to be compared. The team selected microcontrollers/computing devices that appear to be widely used and popular. Therefore, the team selected the *Arduino UNO*, *Arduino Leonardo, Rasperry Pi, Teensy++ 2.0,* and the *Teensy 3.0*.

In order to compare the microcontrollers, parameters were decided for the requirements of the microcontroller/computing device, as well as their corresponding weight value on a scale of 1 to 3 shown in Table 2:

Table 2: Assigned Weight Values

Category	Weight Value
Size	1
Programming Language	3
Environment	3
Analog Input Pins	3
MIDI Capability	3
Flash Memory Capacity	3
Cost	2

The Size of the microcontroller was given a weight value of 1. This was based on the aspect that the size of the microcontroller must be small enough such that the user's gestures, as well as movements are not impeded by the microcontroller wiring. The idea is to have the microcontroller/computing device attached onto either the glove, or perhaps on a belt that the user can wear. However, another applciation could be that the microcontroller/computing device be placed near the computer where long cables are directed from the back of the user to the microcontroller/computing device. Therefore, a value of 1 was assigned since a small sized microcontroller/computing device is desired, but is not a necessity.

The Programming Language was assigned a weight value of 3 due to the importance that the team must be at least familiar with the language in order to properly implement the design. However, if the team is not familiar with a given language, the team could learn the language in the process.

The development environment for the device is also an important parameter and is therefore assigned a value of 3. The environment is helpful in aiding the development of code for the project, especially if the team is not familiar with the particular coding language. The envrionment is also required to more easily interface with the hardware component.

The number of Analog input pins were also considered an important parameter due to the implementation of a number of sensors. The minimum number of analog pins required are 5 for the basic functionality of velocity, embouchure register, and notes (pitch). However, for additional functionality, more analog pins are required.

The MIDI capability is also an requirement due to it being the main feature of the *SansTrumpet* and therefore was assigned a value of 3. The microcontroller/computing device must be able to in some way interface with the MIDI compatible device. If such capability is not native to the microcontroller/computing device, it could be built. However, it is desired that the MIDI data be transferred through USB, which would eliminate the need for a MIDI to USB adapter that is rather redundant due to the fact that the microcontrollers are connected via USB.

The Cost is also an important parameter due to the $125 per student budget for this project. The cost is important since one of the goals is to create a cheap alternative MIDI trumpet controller that both novice and experts could use. If the cost is rather high, the device may not be appealing to the wide range of consumers.

The flash memory capacity was assigned a weight value of 3 due to its importance. The microcontroller or computing device must consist of sufficient flash memory capacity in order to properly store the compiled code. Without enough memory, the compiled code will not function properly. It is currently uncertain exactly how much memory capacity is required for this application. However, the flash memory capacity can be estimated through compiling a similar program. The compiled code size can then be utilized as a reference for how much memory would be required. A MIDI controller program written by "spoocter" on the *Arduino Forums* was compiled, which resulted in a size of 5972 bytes or approximately 6 kilobytes (kB) [47]. A simple MIDI example program for the *Teensy 3.0* compiled to be approximately 11kB [48**]**. These MIDI controller programs, although different than the *SansTrumpet* concept, also interface with a DAW to adjust the various parameters associated with MIDI. The MIDI controller example code for the *Teensy 3.0* exhibits basic functionaility, where as it plays only a few notes. The *SansTrumpet* concept is more complex and therefore the absolute minimum code size is determined to be 10kB, however, the team expects the resulting size to be significantly larger.

The processor speed was not listed in the table due to the uncertainty of exactly how fast of a microcontroller is required. The processor would not be the only factor affecting the speed, the code layout/format will also affect the overall speed. Therefore, it is difficult to pinpoint exactly how fast of a processor would be needed, however, the faster the processor speed is, the less constraints there will be for the code layout.

For assigning raw values to the microcontrolles/computing devices, a scale of 0 to 3 was used:

Size: Size of the microcontroller or computing device

- 3 for very small (Can fit on the wrist area with the glove)
- 2 for reasonable size (fit in one hand)

- 1 for medium-large (can be attached on to a belt)
- 0 for not reasonable (too large to even fit on belt)

Programming Language: Programing language used for the microcontroller or computing device

- 3 for programming language that the team is familiar with, such as C
- 0 for programming language that the team is not familiar with

Development Envrionment:

- 3 for environment that is provided/free
- 0 for no envrionment provided

Analog Input Pins:

- 3 for abundant analog input pins (more than 6)
- 2 for 6 analog input pins
- 1 for 3 analog input pins
- 0 for no analog pins

MIDI Capability:

- 3 for native MIDI capability through USB
- 2 for native MIDI capability (MIDI output which still requires MIDI to USB adapter)
- 1 for external MIDI capability
- 0 for requiring to self build a MIDI (or MIDI to USB) output

Flash Memory Capacity:

- 3 capacity more than 25kB
- 2 capacity 20kB
- 1 capacity of 15kB
- 0 capacity below 10kB

Cost:

- 3 for under $50
- 2 for between $50 to 70
- 1 for between 70 to 100
- 0 for above $100

5.1.2.1 *Arduino* UNO

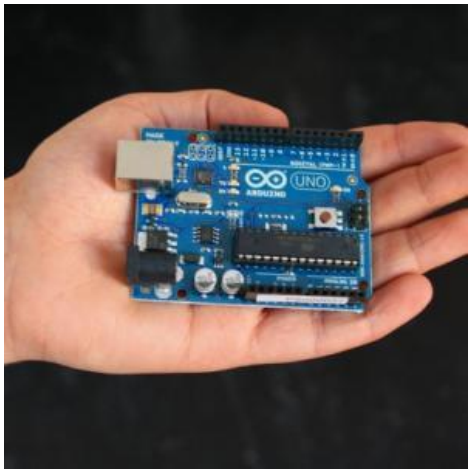

Figure 21: *Arduino UNO* in comparison to a human hand [49]

The *Arduino UNO* is depicted in Figure 21, which shows a size comparison between the microcontroller and a human hand. Table 3 shows the parameters and the assigned raw values for the *UNO* on a scale of 0 to 3. Notice that the *UNO* is rather small and is about the size of a palm. However, this size is rather large to be fit on the glove and therefore would require an implementaion method of using a wearable belt, or by simply placing it near the computer; therefore a value of 2 is assigned. The programming language for the *UNO* is C, which the team is familiar with, and therefore a value of 3 is assigned [50]. The development environment used is the *Arduino IDE*, which is free and open-sourced [51]. The number of analog pins for the *UNO* is 6 and therefore a raw value of 2 was assigned [52]. This is due to the fact that 6 analog pins is just one more than the absolute minimum required of 5 for basic functionality of this application. As mentioned earlier, for more features, more analog inputs are required. For MIDI capability, a MIDI shield is required, which can be attached onto the *UNO*. "Shields" are simply add-on hardware components to the Arduino board. The cost is assigned a value of 1.5 due to the fact that not only

the board must be bought, but an add-on component for the MIDI capability. The flash memory capacity parameter was assigned a value of 3 due to the fact that the UNO has 32kB total flash memory with 31.5kB available to user, which is much larger than the 10kB estimated minimum requirement [52]. Lastly, the UNO costs $30 while the MIDI shield costs $20, which results in a total cost of $50. Table 3 summarizes the assigned raw values for the *UNO*.

Table 3: Assigned Raw Values for *Arduino UNO*

Category	Raw Value
Size	2
Programming Language	3
Development Environment	3
Analog Input Pins	2
MIDI Capability	2
Flash Memory Capacity	3
Cost	1.5

5.1.2.2 *Arduino Leonardo*

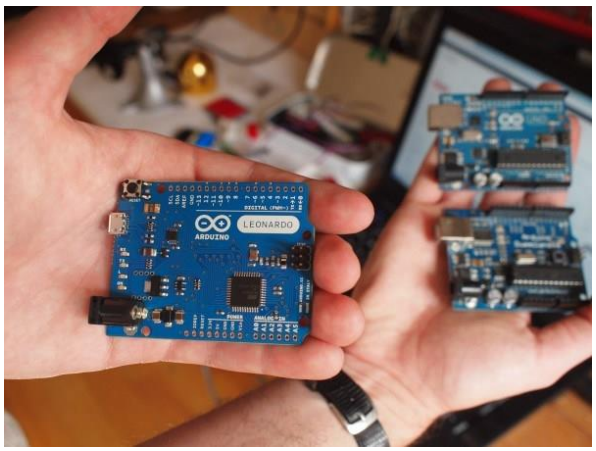

Figure 22: *Arduino Leonardo* in comparison to a human hand [53]

Figure 22 shows the *Arduino Leonardo* in comparison to a human hand. Similarly to the *UNO*, the *Leonardo* is also relatively small but is not small enough to be placed on the wrist with the glove. Therefore, like the *UNO*, a raw value of 2 is assigned to the size parameter. As mentioned earlier, the Arduino microcontrollers are programmed with C and use the same development environment; therefore, a raw value of 3 is assigned to the programming language

and environment parameters. The Leonardo consists of 6 analog pins along with 6 of the digital pins that can be utilized as an analog pin, totaling to 12 analog pins [54]. For MIDI capability, the MIDI shield is not entirely compatible with the *Leonardo* due to the difference in processors between the *UNO* and *Leonardo* as stated by Evan Spitler, a *SparkFun* employee [55]. Such incompatibility relates to the architecture of the processors, and so the serial pins layout are different. Thus, modifications would be required in order to correctly connect the MIDI Shield to the *Leonardo*. Due to the requirement of such modifications, a value of 1 is assigned for this parameter. Same as the *UNO*, the *Leonardo* has 32kB of flash memory capacity [54]. Lastly, the cost is assigned a value of 1.5, since it too would cost approximately $50, like the *UNO*, due to other materials such as the MIDI shield. Table 4 shows the resulting assigned raw values for the *Leonardo*.

Table 4: Assigned Raw Values for *Arduino Leonardo*

Category	Raw Value
Size	2
Programming Language	3
Development Environment	3
Analog Input Pins	3
MIDI Capability	1
Flash Memory Capacity	3
Cost	1.5

5.1.2.3 *Raspberry Pi*

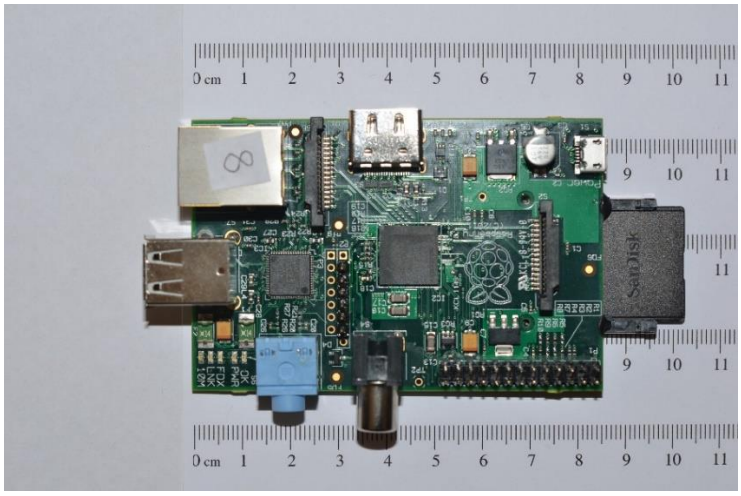

Figure 23: *Rasperry Pi* in length [56]

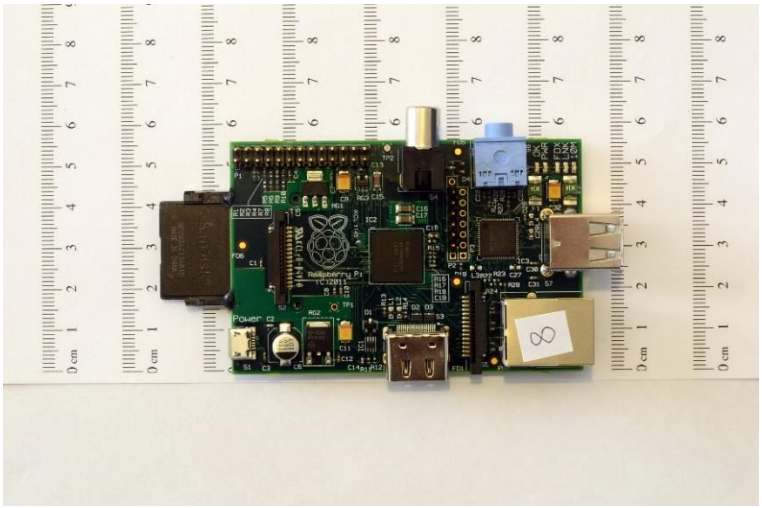

Figure 24: *Rasperry Pi* in width [57]

The *Raspberry Pi* is actually an inexpensive small sized computer rather than a microcontroller [58]. The Raspberry Pi runs the Linux operating system which is loaded onto a SD card [58]. The *Rasperry Pi*'s official supported coding language is Python, however, other programming languages are possible, but compatibility is not guaranteed [58]. Since the *Raspberry Pi* is an actual computer, this differentiates it from microcontrollers, such as the *Arduino*. This also indicates that the *Raspberry Pi* is much more powerful, however, it would require more experience with computer coding, specifically with Python on a Linux platform.

Figures 23 and 24 shows the *Raspberry Pi,* along with measurements of length and width. Notice that the length is only approximately 10cm and the width is approximately 6cm. This size is also relatively small, however, it is slightly larger than the Arduino microcontrollers; therefore a raw value of 1.5 is assigned. The officially supported programming language, as mentioned, is Python in which the team are not proficient in. However, the *Raspberry Pi* is not limited to Python and can be programmed in other languages that can compile for ARMv6 [58]. Thus, a value of 2 is assigned. The environment is assigned a value of 3 due to the wide variety of environments available for the Linux platform. The *Raspberry Pi* does not have analog pins and therefore was assigned a value of 0 [58]. In order to utilize the *Raspberry Pi*, the analog input signals must be converted to digital input signals with an external Analog to Digital Converter (ADC) [59]. For MIDI capability, the *Raspberry Pi* does not have a native MIDI output and therefore would require the team to build a MIDI output circuit. The memory capacity is based on the storage device, such

as a SD card; thus, the capacity varies from user to user [58]. With a varying memory capacity, the memory capacity should not be an issue since the user is able to select the most appropriate SD card for their application and so, a value of 3 was assigned. The cost is assigned a raw value of 3 since it only costs $35, which is rather cheap. Table 5 summarizes the assigned values for the *Raspberry Pi*.

Table 5: Assigned Raw Values for *Raspberry Pi*

Category	Raw Value
Size	1.5
Programming Language	2
Development Environment	3
Analog Input Pins	0
MIDI Capability	0
Flash Memory Capacity	3
Cost	3

5.1.2.4 *Teensy++ 2.0*

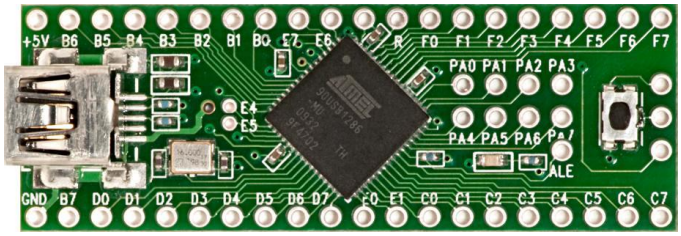

Figure 25: *Teensy++ 2.0* [60]

The *Teensy++ 2.0* is only 2.0 inches by 0.7 inches in size; hence, it is assigned a value of 3 [61]. The *Teensy* is programmed in C, and similarly to the Arduino, a value of 3 is assigned due to the fact that the team is familiar with the C programming language. For the development environment, a value of 3 is also assigned since there are a variety of development tools available. In fact, the add-on, *Teensyduino* allows the user to utilize the *Arduino IDE*, which is free, to develop and load compiled code onto the *Teensy* [62]. The *Teensy++ 2.0* has 8 analog input pins, which is more than 6, and therefore assigned a value of 3 [60]. It also has USB device capability, allowing it to appear as a USB device, including MIDI, to the computer [63]. Thus, a value of 3 is assigned since it has native MIDI capability through USB. The Flash memory capacity was

indicated to be 130048 bytes, which is approximately 130kB [62]. This is much more than the estimated 10 kilobyte minimum requirement, hence a value of 3 was assigned. The cost is rather cheap at about $25, thus a value of 3 was given for the cost parameter. Table 6 summarizes the assigned values for the *Teensy++ 2.0*.

Table 6: Assigned Raw Values for *Teensy++ 2.0*

Category	Raw Value
Size	3
Programming Language	3
Development Environment	3
Analog Input Pins	3
MIDI Capability	3
Flash Memory Capacity	3
Cost	3

5.1.2.5 *Teensy 3.0*

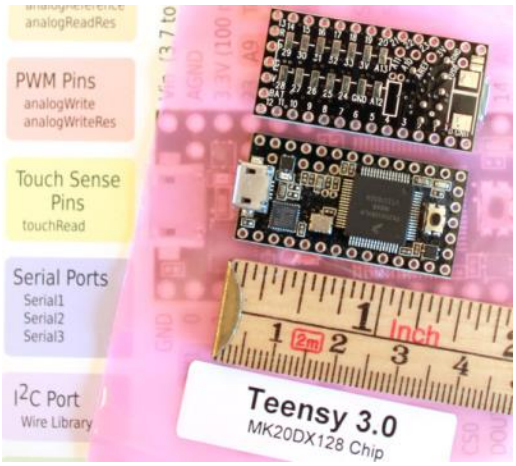

Figure 26: *Teensy 3.0* [64]

The *Teensy 3.0*, like the *Teensy++ 2.0*, is assigned a value of 3 due to its small size. In fact, as shown in Figure 26, its length is only approximately 1.5 inches. The programming language, as stated earlier, is C for *Teensy* microcontrollers, and they utilize a variety of development environments, including the *Arduino IDE*. Therefore, a value of 3 is assigned to both the programming language and environment parameters. The *Teensy 3.0*, however, has 14 analog input pins, compared to 8 for the *Teensy++ 2.0* [62]. Like the *Teensy++ 2.0*, the *Teensy 3.0* also

consists of MIDI capability through USB, therefore a value of 3 is assigned. The *Teensy 3.0* has slightly more flash memory than the *Teensy 2.0* with 131072 bytes, which is approximately 131kB [62]. Lastly, the cost for this microcontroller is only $20, leading to a value of 3 for the cost parameter. Table 7 shows the resulting assigned raw values for the *Teensy 3.0*.

Table 7: Assigned Raw Values for *Teensy 3.0*

Category	Raw Value
Size	3
Programming Language	3
Development Environment	3
Analog Input Pins	3
MIDI Capability	3
Flash Memory Capacity	3
Cost	3

5.1.2.6 Value Analysis

Table 8: Value Analysis for Microcontroller/Computing Device Selection

Value Analysis

			Raspberry Pi		*Arduino Leonardo*		*Arduino UNO*		*Teensy++ 2.0*		*Teensy 3.0*	
		Weight	Value point	Total	Value point	Total	Value point	Total	Value point	Total	Value Point	Total
1	Size	1	1.5	1.5	2	2	2	2	3	3	3	3
2	Programming Language	3	2	6	3	9	3	9	3	9	3	9
3	Development Environment	3	3	9	3	9	3	9	3	9	3	9
4	Analog Input Pins	3	0	0	3	9	2	6	3	9	3	9
5	MIDI Capability	3	0	0	1	3	2	6	3	9	3	9
6	Flash Memory Capacity	3	3	9	3	9	3	9	3	9	3	9
7	Cost	2	3	6	1.5	3	1.5	3	3	6	3	6
Totals:				**31.5**		**44**		**44**		**54**		**54**

Table 8 depicts the value analysis table in order to choose the microcontroller/computing device for this project. As seen, the top two microcontrollers are the *Teensy++ 2.0* and *Teensy 3.0* both tied with 54 points. Notice that the flash memory capacity values for all 5 of the selected microcontrollers/computing devices were assigned a maximum of 3. This indicates that these 5 microcontrollers/computing devices are all more than sufficient for this application. The two total point values are equal, however, the processor speed parameter was not considered in the value analysis. As indicated earlier, the Team was uncertain how fast the processor must be. However, the *Teensy 3.0* has both more memory and processor speed than the *Teensy++ 2.0* [62]. As

explained, a microcontroller with more speed would result in less layout/formatting constraints and therefore, it is a safer and more futureproof choice to select the *Teensy 3.0*, in case processor speed becomes a limiting factor. The *Teensy 3.0* is a class compliant MIDI device such that it complies with the USB MIDI standards [65]. Since it is class compliant, the *Teensy 3.0* would utilize the generic USB drivers of the operating systems. Due to the fact that the *Teensy 3.0* complies with the USB MIDI standards, it is assumed that the baud rate of 31,250 bits per second is used. Thus, this verifies that the team must implement a method to match the baud rate to prevent potential loss of data.

The power source for the entire device must also be considered. Both the flex sensors attached on to the gloves, as well as the *Teensy 3.0* microcontroller, must be powered. The *Teensy 3.0* can be powered via a USB cable or an external power supply. However, the *Teensy 3.0* must be connected to the computer via a USB cable in order for it to send MIDI data to the DAW. Therefore, the microcontroller should be powered via the USB cable since the USB cable is already required for data transfer. The flex sensors attached on to the gloves would also require a source in which the options are to use a battery, a power supply connected to 120V AC line voltage, or the USB. The usage of a battery would allow for portability. However, due to the fact that the microcontroller must still be connected to both the glove and the computer, the usage of the battery is not reasonable. The usage of an outlet is possible, however, it is not feasible due to the application. A power supply unit would have to be built, which includes AC to DC conversion. Therefore, the portability factor would diminish since the user would need to carry the power supply unit along with the glove. The USB option would be the most reasonable choice due to the fact that the *Teensy 3.0* microntroller is required to be connected to the computer via USB. Therefore, the USB that is used to source the microcontroller can also be used to source the flex sensor circuitry. Although not as portable as the battery, the USB can be directed from the user's back to the computer to minimize disturbances in usage due to wiring.

The USB 2.0 standard is known to provide up to 100 mA at 5V by default [66]. However, up to 500mA can be provided as long as the device sends a request message to the computer [66]. The *Teensy 3.0* is designed to only handle voltages up to 3.3V, rather than 5V, and so the voltage at the analog input pins must not exceed 3.3V [60]. To remedy this issue, a voltage regulator can

be utilized which will drop the 5V voltage down to 3.3; this will protect the *Teensy 3.0* from overvoltage. The voltage regulator was selected on a criteria that it must be a through-hole mounting type, rather than surface mount. This is due to the fact that surface mounts will lead to difficulty in constructing the circuit due to the miniscule size of surface mount pins. Two main factors considered in the selection of the voltage regulator are the voltage drop across the regulator and the current draw of the regulator. The voltage regulator must be a (low drop out) LDO voltage regulator. The LDO voltage pertains to the voltage required for the regulator to function and therefore is the voltage drop across the regulator. Since the USB voltage is 5V and the desired voltage is 3.3V, the voltage drop across the regulator can be a maximum of 1.7V in this application. The regulator musy also draw only a small amount of current to leave enough for the *Teensy 3.0*, which is called the quiescent current. From a search through *Digikey*'s catalog, three 3.3V voltage regulators fit the mentioned criteria, which are summarized in Table 9.

Table 9: Selected Voltage Regulators and their corresponding voltage drop and typical quiescent current

Regulator	Voltage Drop	Typical Quiescent Current
MCP1700-3302E/TO-ND	178mV at 250mA [67]	1.6µA [67]
MCP1702-3302E/TO-ND	625mV at 250mA [68]	2.0µA [68]
AP1086T33L-UDI-ND	1.4V max [69]	5mA [69]

Although the three listed votlage regulators fit the stated criterias, the MCP1700-3302E/TO-ND and MCP1702-3302E/TO-ND appear to be more appropriate for this application due to the significantly less voltage drop and quiescent current. The MCP1700-3302E/TO-ND was chosen due to the fact that it costs less than the MCP1702-3302E/TO-ND regulator and can still meet the requirements of the project.

The flex sensor circuitry simply consists of voltage dividers such that each flex sensor is paired with a resistor in series to form a voltage divider. Each divider is connected in parallel such that each voltage at the divider is considered an input to the analog input pins of the micrcontroller. The data sheet for the flex sensor recommends the usage of a buffer to reduce sampling errors [46]. The datasheet for the AtMega328 processor, which is utilized by the Arduino, indicates that the output impedance (source impedance) must be equal to or less than 10kΩ, otherwise significant

delay in the analog to digital conversion would occur [70]. As indicated earlier, the team selected the *Teensy 3.0* and not the *Arduino* microcontrollers, however, the data sheet for the *Teensy* does not indicate a requirement for the output impedance at the analog input pins. Therefore, it is assumed that like the *Arduino*, the output impedance should also be less than or equal to 10kΩ. The resistor value used for each voltage divider along with the flex sensors is assumed to be 45kΩ. This assumption is made on the concept that for a 45 degree bend of the flex sensor, the resulting resistor of the sensor would be 45kΩ. However, flex sensors have a tolerance of 30%, therefore it is expected that the resistance at a certain angle will vary from sensor to sensor [46]. Thus, the actual resistor value that will be used will be determined through the measurement and testing of the flex sensors. The Thevenin resistance of each sensor output, or each voltage divider branch was calculated in order to determine the necessity of buffers. With assumption that the resistors used for each voltage divider will be 45kΩ, the Thevenin equivalent resistance of each voltage divider was determined to be approximately 16.07kΩ which is grater than 10kΩ. This value is calculated as a minimum resistance such that the flex sensors are at a nominal value of 25kΩ with no bend. Thus, the output impedance must significantly increase once the flex sensors are bent, which may increase up to 125kΩ. With the flex sensors bent up to 90 degrees, the Thevenin resistance would become approximately 33.09kΩ, which is also significantly greater than the 10kΩ limit. Therefore, these Thevenin values indicate the need for buffers to be implemented into the circuit to reduce error with sampling. With the buffers, the output impedance from the analog input pins would appear to be low since the op-amp is designed to have a high input impedance (ideally infinite) and a low output impedance(ideally 0).

Similarly to the voltage regulator requirements, the operation amplifier(op-amp) must also be through-hole mounting type and only draw low current as to minimize the impact on the current supplied to the *Teensy 3.0*. Other criterias include the voltage offset and current bias. The voltage offset and current bias refer to the additional DC voltage and current draw at the + and – pins due to intrinsic error of the op-amp. Ideally, the input impedance is infinitely high, however, realistically, this is not so. These biases would affect the the voltage reading at the pin, and therefore introduce an error into the system. Thus, it an op-amp exhibiting low biases would be desired. Lastly, since the device is intended to be on the user's body, it is imperative that the size

of the op-amps must be small in size. With these criterias, three op-amps are considered for this application which are shown in Table 10.

Table 10: Selected Op-Amps and their corresponding maximum current draw, voltage offset, and current bias

Op-Amp	Max Current Draw	Max Voltage Offset	Max Current Bias at 85°C
MCP602-I/P-ND	325µA [71]	2mV [71]	60pA [71]
MCP604-I/P-ND	325µA [71]	2mV [71]	60pA [71]
MCP606-I/P-ND	25µA [72]	250µV [72]	80pA [72]

As shown in Table 10, the MCP602-I/P-ND, MCP604-I/P-ND, and the MCP602-I/P-ND all meet the electrical criteria. As shown in the above data, the MCP606-I/P-ND has significantly lower max current draw and max voltage offset. In terms of physical criteria, the three op-amps are through holes. All three op-amps are also quite small, however the MCP604 IC is an array of 4 op-amps as opposed to the MCP602 and MCP606 with 2 and 1 respectively. Due to the fact that the MCP604 is an array of op-amps, it is the more reasonable choice for the purposes of the project's application.

5.2 Initial Circuit Design

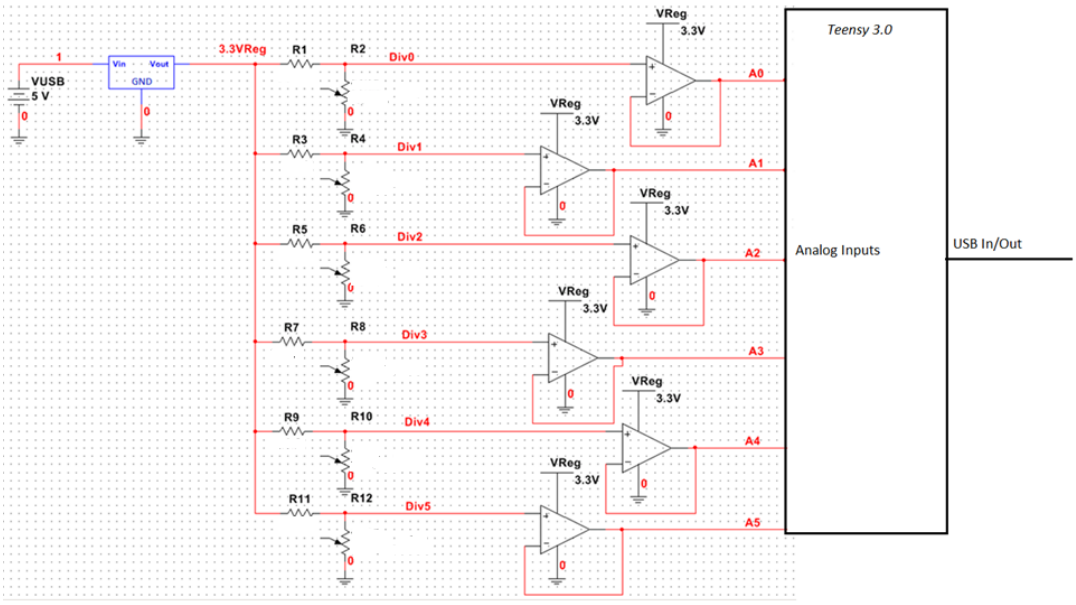

Figure 27: Initial Circuit Design Schematic

Figure 27 depicts the initial circuit schematic modeled with *MultiSim* with generic components. As seen, the circuit consists of a voltage regulator, resistors, operational amplifiers, and a microcontroller. Through measurements, the values for certain components such as the resistors, were determined. As shown, the circuit is comprised of six voltage dividers in which the output of each divider serves as an input to the *Teensy 3.0* microcontroller. Each voltage divider consists of a resistor with a fixed value and a flex sensor. The flex sensor performs as a variable resistor in which the resistance varies, depending on the angle of bend. The idea behind each voltage divider is that as the flex sensor is bent, the resistance of the flex sensor increases. Due to this increase in resistance, the voltage across the flex sensor also increases. The initial design consists a total of six flex sensors where five are used for the main functionality. The main functionality consists of valve combination, embouchure register, and the trigger. Three flex sesnsors are used for the valve combination, one for the embouchure register, and one for the trigger. The sixth flex sensor is intended for additonal funcitonality such as pitchbend.

The op-amps as seen in the schematic, are configured as buffers to address potential errors due to high impedance. The datasheet for the AtMega328 processor of the *Arduino* microcontroller, indicates a requirement of a source impedance than 10kΩ, otherwise errors in the analog to digital conversion would occur [70]. This is due to the fact that the analog to digital conversion circuit of the Arduino involves the use of a sample and hold capcitor [70]. Due to the use of a sample and hold capacitor, the amount of time for the capcitor to charge to the appropriate voltage will beaffected by the source impedance. If the impedance is larger than 10kΩ, then the time constant, RC, will be large. Therefore with a large impedance, significant time would be required in order to charge and discharge the capcitor during the sampling of the analog input pins. This would cause errors in the analog to digital conversion to occur since the during the instance that the pin is sampled, the capcitor is charged to a voltage that may significantly differ from the voltage at the corresponding pin. As indicated earlier, the team selected the *Teensy 3.0* and not the Arduino microcontrollers, however, the data sheet for the *Teensy* does not indicate a requirement for the output impedance at the analog input pins. However, it is assumed that like the *Arduino*, the output impedance should also be less than or equal to 10kΩ. This was further investigated, and will be discussed later in this report. The analog voltage is converted to a digital representation through the use of the Analog to Digital Converter (ADC) on the *Teensy*. A voltage regulator was also

necessary due to the fact that the *Teensy* analog input can only handle a maximum voltage of 3.3V. As shown, the voltage source used is the USB voltage which provides 5V. Therefore, the regulator is utilized to drop the USB voltage of 5V to 3.3V in order to protect the *Teensy* analog input pins from overvoltage.

5.3 Intermediate Circuit Design

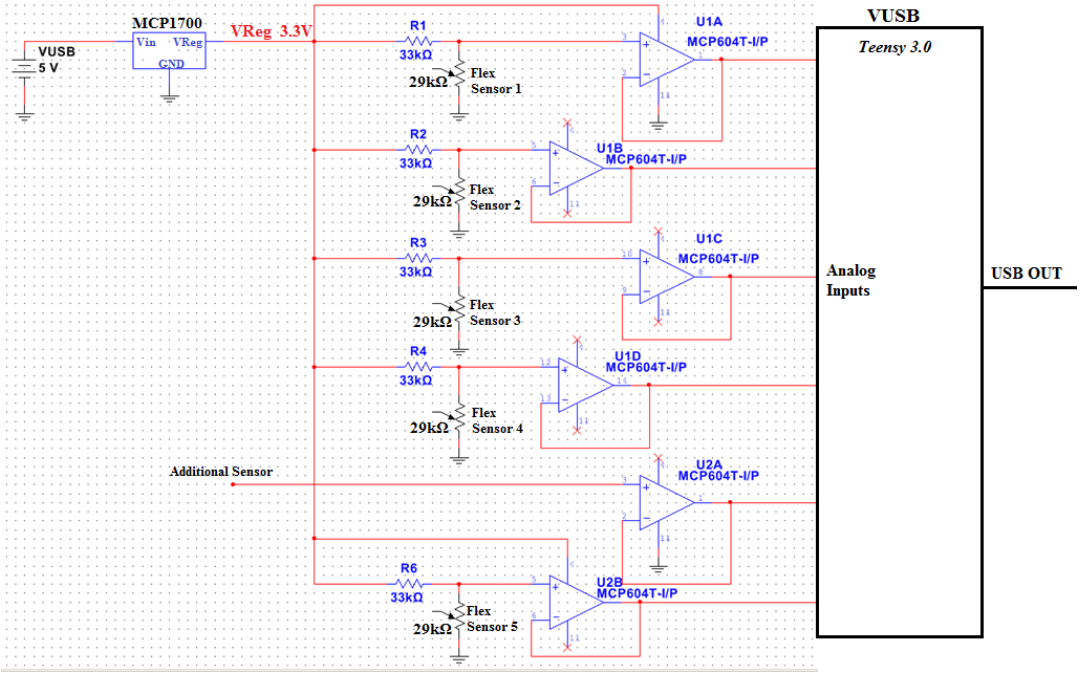

Figure 28: Intermediate Circuit Schematic

Figure 28 depicts the modified circuit schematic for the *SansTrumpet*. Notice that the general form still follows the initial schematic. However also notice that the resistors in the schematic now have specific values. These values were determined on the basis of the selected flex sensors. As indicated earlier, the team decided that the 2.2" flex sensor would be the most appropriate for the project's application. In particular, the 2.2" flex sensor would provide the ability to pinpoint the area of bend, allowing easier accommodation to people's hands. As indicated earlier, the chosen resistor values must correspond to the resistance of the flex sensors. Initially, the concept was to use 45kΩ resistors such that when the flex sensors are bent to 45°, the voltage would be halfway. For more accurate values, the DMM was utilized to measure the resistance at

the corresponding angles of 0°, 45°, and 90°. The measured resistance at the 0° bend was used to determine the appropriate resistance value for the voltage divider.

Table 11a: First Flex Sensor Resistance Measurement

Angle	Resistance
0°	33.20kΩ
45°	40.30kΩ
90°	54.10kΩ

Table 11b: Second Flex Sensor Resistance Measurement

Angle	Resistance
0°	34.58kΩ
45°	50.30kΩ
90°	78.20kΩ

Table 11c: Third Flex Sensor Resistance Measurement

Angle	Resistance
0°	28.82kΩ
45°	36.26kΩ
90°	39.99kΩ

Table 11d: Fourth Flex Sensor Resistance Measurement

Angle	Resistance
0°	29.57kΩ
45°	50.60kΩ
90°	68.60kΩ

Table 11e: Fifth Flex Sensor Resistance Measurement

Angle	Resistance
0°	29.65kΩ
45°	52.90kΩ
90°	68.60kΩ

Table 11f: Sixth Flex Sensor Resistance Measurement

Angle	Resistance
0°	30.92kΩ
45°	48.87kΩ
90°	58.30kΩ

Table 11g: Seventh Flex Sensor Resistance Measurement

Angle	Resistance
0°	28.95kΩ
45°	34.67kΩ
90°	42.20kΩ

Table 11h: Eighth Flex Sensor Resistance Measurement

Angle	Resistance
0°	29.27kΩ
45°	41.50kΩ
90°	49.02kΩ

Tables 11a to 11h depict the resistance measurements for the flex sensors when bent at angles of 0°, 45°, and 90°. Notice that the measured values vary significantly from sensor to sensor. As seen in the measured data, the resistance of the flex sensors at a 45° bend appear to vary significantly from sensor to sensor. However, the resistance at 0° appear to be rather consistent in which they range from approximately 29kΩ to 34kΩ. Therefore, the initial concept of choosing resistor values for 45° degree bend of the flex sensors would be unreasonable due to such inconsistency. Instead, the team decided to use 33kΩ resistor to correspond to 0° bends since the flex sensors resistance appear to be more consistent when unbent. Also notice the large discrepancy between the measured resistances at both 45° and 90° bends. This variation of resistance may pose to be an issue for this project; however there are two possible solutions to address this issue. One of which is to appropriately assign different fixed resistor values to each flex sensor in order to obtain the appropriate voltage reading. For example, the eight flex sensor measurements shown in Table 10h indicate that the snesor exhibits a resistance of 29.27kΩ at 0°. Therefore, the resistor associated with his flex sensor will be set to approximately 30.00kΩ instead of 33kΩ. However from the measurements, the 45° and 90° exhibit more significant discrepancy and so the method of changing resistors will have minimal impact when the sensors are bent to these angles. The second method is to implement calibration through software which will be further explained in the software section of this report.

Due to the fact that the voltage dividers are comprised of components with high resistance relative to the 10kΩ limit, the total impedance looking into the divider toward the source from the *Teensy* input pins, is relatively high. The need for buffers was further invesigated by utilizing a

simple program which serially prints text to the serial monitor. The simple program reads the analog pins and displays on the serial monitor the 10-bit digital equivalent. The fixed resistance values were chosen such that when the flex sensors were unbent, the voltage at the analog input should be approximately half. In digital representation, the ADC was set to 10-bit resolution, indicating that the digital range is from 0 to 1024. Therefore for a voltage of half the supply voltage to the divider, a digital value of approximately 512 is expected. From the simple serial print program, the resulting values were unstable such that the values fluctuated between 10 to 700. This result indicates that perhaps due to the high resistance, the analog input appeared to be open, resulting in error in the reading of the analog voltage. Therefore, this result verifies that buffers are required in addressing the impedance issue. With the buffers, the digital values on the monitor indicated to be approximately 500 to 514, matching expectations.

Specifically, two MCP604 Op-Amp IC Arrays are used for this circuit. The main reason for the selection of this IC is due to the size and power consumption. In terms of size, each IC consists of four Op-Amps, and therefore only two are requried for this application. Due to the requirement for the device to be on the person, it is crucial to minimize the size of the device. For power consumption, the Op-Amp consumes 230μA and requires 2.7V [71]. Although the power consumption is higher than the MCP606 Op-Amps, considering size, the MCP604 Op-Amp arrays are more appropriate. The USB typically provides 100mA, however it can also provide up to a maximum of 500mA maximum at 5V; this is certainly enough for the Op-Amp IC, and therefore is appropriate in terms of size and power requirements. The regulator for this application is the MCP1700 3.3V regulator. As indicated, this is required to protect the *Teensy* from overvoltage. The voltage regulator too, also consumes low power where it draws only 1.6μA and only requires 178mV(at 250mA) [67].

Notice that the number of flex sensors have decreased from six to five. This is due to the fact that the flex sensor that was intended for the embouchure was replaced with a different sensor denoted as “Additional Sensor” as shown in the schematic. This is due to the consideration of ergonomics such that if the left hand were to control both the trigger and embouchure, the thumb and the index finger may end up impeding each other when the device is being played quickly. Therefore, the trigger was relocated to the index finger while the embouchure will be moved to

another location. Currently the team is debating on which sensor to utilize and where to implement the sensor that would be the most appropriate for this application. Some implementations that the team is currently considering are the use of accelerometers, magnetometers, proximity sensors, or perhaps to simply relocate the flex sensor to a pedal. The team currently posess an accelerometer however the implementation of it may pose to be an issue. Specifically, rapid hand flicking would interfere with the fluidity of the playing technique. The accelerometer would also be quite limiting since the user would have to manually increment/decrement the embouchure through flicking. However, perhaps a combination of an accelerometer and an additional sensor such as the proximity sensor would resolve this issue and may allow for more advanced techniques. Further research is required to decide the best implementaion of one or more of these sensors for the embouchure and perhaps additional features.

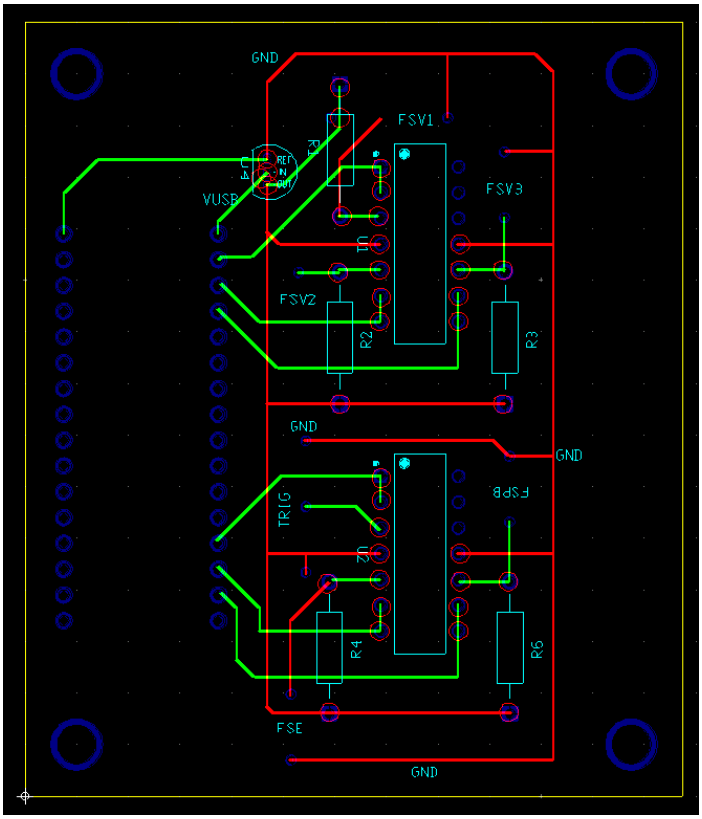

Figure 29: Initial PCB Layout

Table 12: Measured Diameters of Component Leads

Component	Diameter
Resistor	25.2mils
Wires from ECE Kit	25.8mils
Breakaway Header Pins	25.2mils

With the initial schematic depicted in Figure 28, the Printed Circuit Board(PCB) layout was developed with *National Instrument*'s *UltiBoard* as shown in Figure 29. The yellow border seen in the layout refers to the physical board outline. The size of the board is currently set to approximately 2.6 x 2.6 square inches, however this would perhaps be increased to 3 x 3 square inches after some adjustements are applied. The green and red lines refer to the top and bottom traces respectively. The blue circles shown in the layout refer to the vias for the through hole compoenents. The four large vias in the corners are intended for the screws or bolts for mounting the PCB on to the enclosure. Concerns for the correct via diameters were addressed by using a *Vernier* Micrometer from the ECE Shop; the resulting measurements are shown in Table 12.

5.4 Final Circuit Design

As previously indicated, the embouchure was to use an "Additional Sensor" as demonstrated in Figure 28. The team investigated possible sensors for the implementation of the embouchure. The team considered three possible options, the accelerometer, magnetometer, and proximity sensors. As indicated previously, the accelerometer's functionality posed a potential issue for the playing technique, and therefore, is not a reasonable choice. Specifically, the user would have to flick their hand in order to transition from one embouchure level to another. However, such hand flicking would interfere with the fluidity of the playing technique, as well as the performance aspect. A magnetometer, as its name implies, measures the magnetic field; generally, magnetometers are utilized as a digital compass [73]. Therefore, if a magnetometer was implemented, the horizontal hand position would correlate to the embouchure level. Proximity sensors generally measure distance between the sensor itself and an object. Similarly to the magnetometer, the implementation of the proximity sensor would also correlate the embouchure level to the hand position. Unlike magnetometers, the proximity sensor would not be limited to only horizontal position. However, proximity sensors would be limited such that an object must be present in order to measure the distance. Any relatively large object could be used, however the

most reliable object would be the floor. Therefore, in some sense, the proximity sensor is limited to vertical position due to reliability. Both magnetometers and proximity sensors appear to be applicable for the *SansTrumpet* such that they can allow for fluidity in playing technique, providing a performance aspect.

The team decided that the proximity sensor would be the most appropriate sensor for the implementation of the embouchure. Although the magnetometer is applicable, it also introduces potential interference issues. Since the magnetometer measures the magenetic field, it may be susceptible to other electronic components within the vicinity of the device. The other compoenents of the *SansTrumpet* device, such as wires, may also interfere with the magnetometer. The proximity sensor is also susceptible to inteference in which some objects may be in between the sensor and the intended reference point/object. However, such physical interference for the proximty sensor can be addressed by the user much more conveniently than addressing electrical interference regarding the magnetometer.

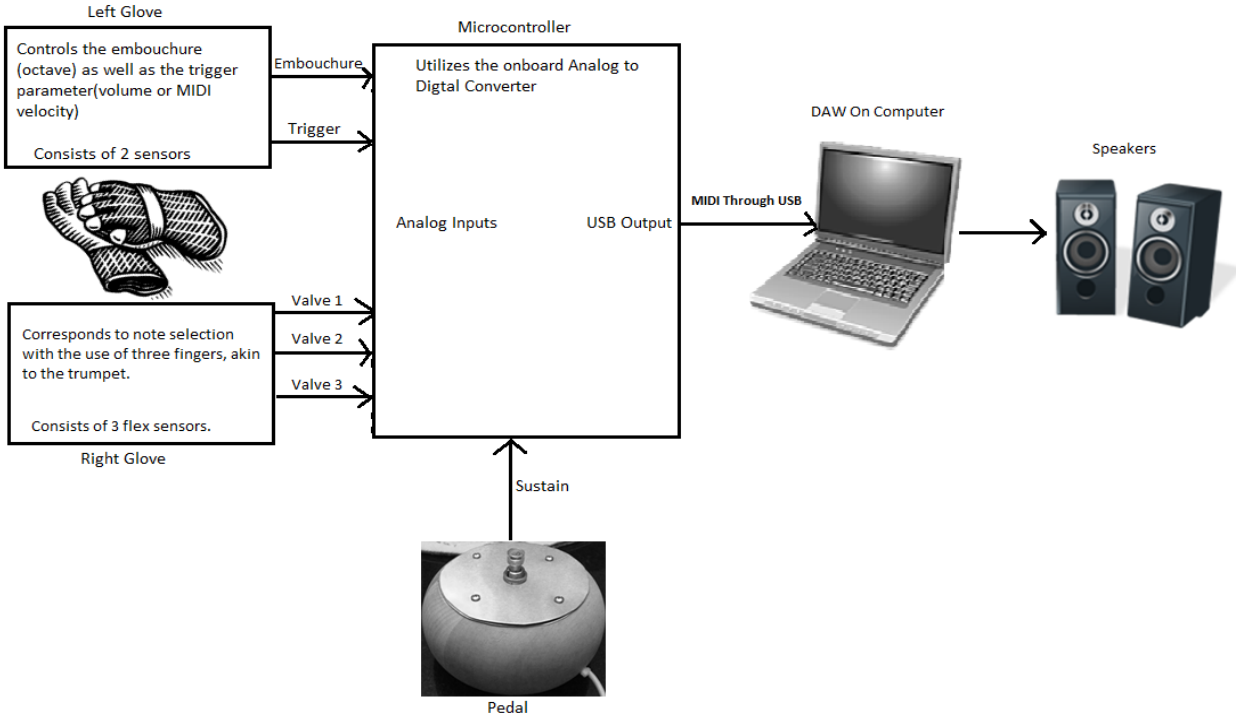

Figure 30: Block Diagram of Final *SansTrumpet* Design

Figure 30 depicts the block diagram of the *SansTrumpet* in consideration of the various alterations from the initial design to the final design. In particular, left hand glove was updated to

indicate the use of “2 sensors” rather than “2 flex sensors”. This is to indicate the replacement of the embouchure register sensor with the proximity sensor as indicated previously. In addition, the block diagram also includes the sustain pedal as one of the inputs. As described, the sustain pedal is utilized to switch between the two modes of the *SansTrumpet*. The output remains the same such that the microcontroller sends the MIDI signal to the DAW. The DAW would interpret the signal and produce the corresponding sound through the speakers.

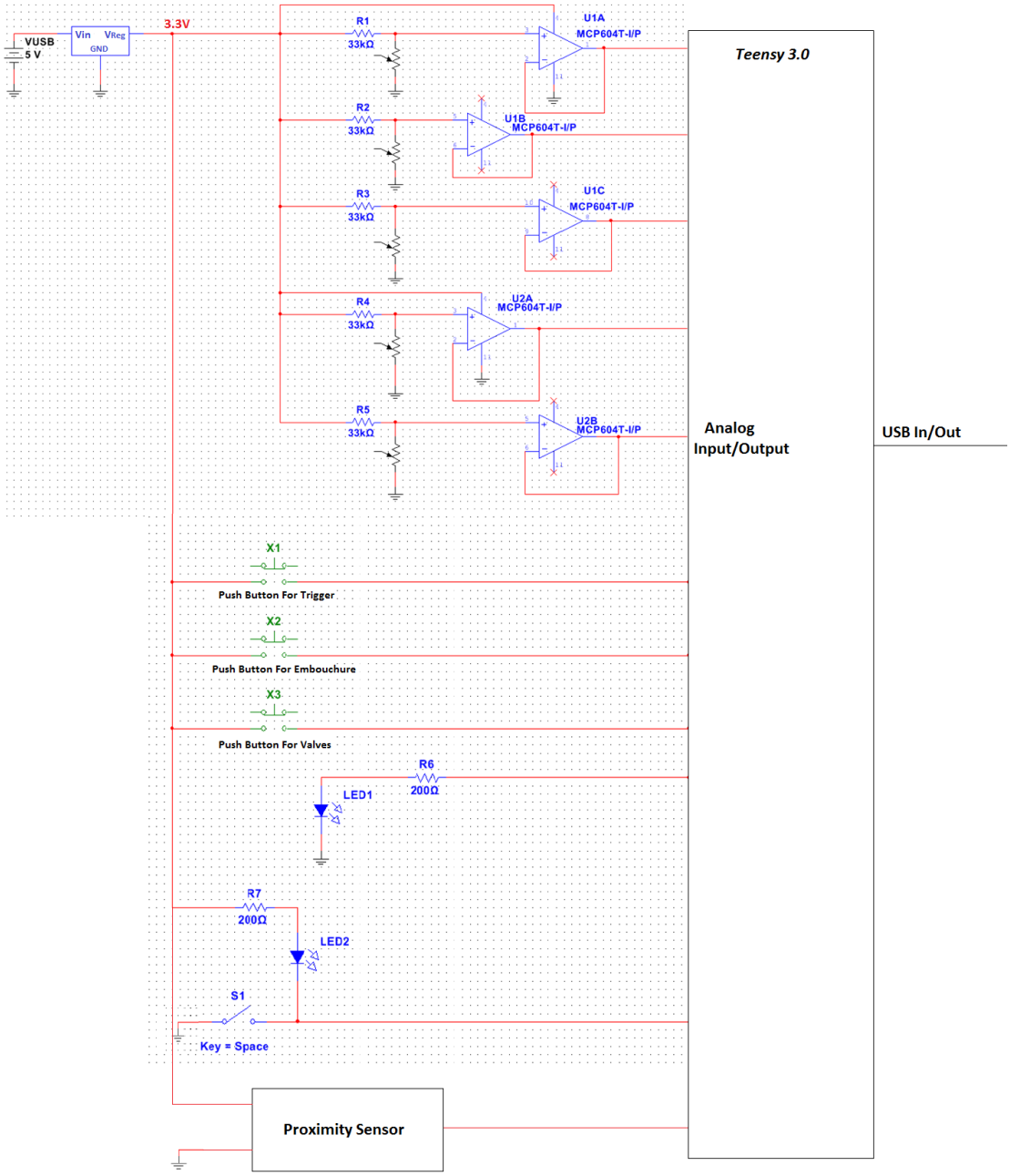

Figure 31: Final Circuit Schematic

Figure 31 shows the final circuit schematic of the *SansTrumpet*. Notice that several changes were applied from the previous circuit schematic. The circuit schematic consists of five Op-Amps which correspond to the five flex sensors that are utilized for the *SansTrumpet*. Specifically, the flex sensors are utilized for the three valve fingers, one for trigger, and another for the possible addition of a pitch bend functionality. As indicated earlier, the embouchure was to use an "Additional Sensor" as indicated in Figure 28. Shown in Figure 31, the proximity sensor is utilized for the embouchure.

Due to the inclusion of a buffer in the proximity sensor unit, an additional Op-Amp was not necessary. Therefore, only five op-amps are necessary, as opposed to the six indicated in the previous schematic. Also, notice the inclusion of several additional components such as the push buttons, LEDs, and a switch. These components are utilized for the calibration functionality. The three push buttons, depicted in green, correspond to the initiation of the calibration; each button relates to the trigger, embouchure, and valves respectively. The green LED labeled as "LED1" in the schematic, is utilized to guide users through the calibration procedure. The LED blinks through the duration of the calibration capture to alert the user. The switch, "S1" and the LED, "LED2", correspond to the illuminated switch, which allows the user to switch between two modes of the *SansTrumpet*, sustain and single-note, which will be further discussed in the software portion of this report.

Figure 32: "MaxBotix Ultrasonic Rangefinder LV-EZ1" Proximity Sensor [74]

A concern that was considered when selecting a proximity sensor is that it must be able to function with at least 3.3V. Other considerations were performance and cost related such that the device must be able to function well enough that the distance measure can be split into six intervals. These intervals correspond directly to the six embouchure levels, as described previously, for the

embouchure concept. The proximity sensor chosen for the *SansTrumpet* device is the "MaxBotix Ultrasonic Rangefinder LV-EZ1" sensor, as shown in Figure 32. The proximity sensor functions by emitting an ultrasound signal and measuring the time required for the signal to rebound back to the sensor [75]. The sensor output voltage would, therefore, correspond to the distance between the object and the sensor.

For the calibration buttons, single pole single throw momentary pushbuttons were utilized. This is to allow the user to hold the buttons for a set amount of time to initialize the calibration function. By requiring the user to hold the push buttons down, this prevents unintentional activation of calibration when the buttons are accidently tapped or pressed. In order to provide a form of feedback to the user regarding calibration, a LED was utilized, as shown in the circuit schematic. Through software, the LED is controlled, where one of the *Teensy* pins is utilized to output either a high (Vcc) or low (Gnd) to turn the LED on or off. Lastly, a single pole single throw illuminated pushbutton switch was implemented to allow the user to transition between the modes of the *SansTrumpet*.

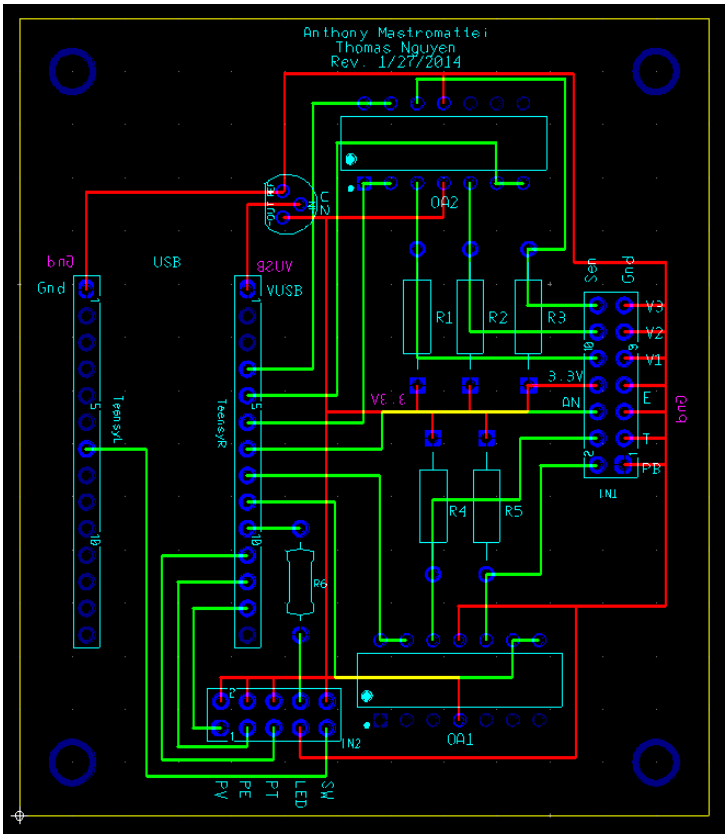

Figure 33: Final PCB Layout

Figure 33 depicts the final PCB layout in which modifications were applied to the initial PCB layout to improve visual comprehensibility. As shown, the component placements were relocated such that the op-amps are located near the top and bottom of the PCB while the resistors are grouped towards the middle. Two headers are used for the sensor and calibration connections. This is to group the corresponding inputs in one area so as to maintain the visual comprehensibility. As shown in the layout, inputs associated with the sensors (flex and proximity) are located on the right side of the PCB. Similarly, components associated with the calibration are located towards the bottom. Notice that the copper traces were modified from the initial PCB layout such that trace angles were limited to 90°, due to visual considerations. Other modications correspond to the via diamater size based on measurements, which were conducted with a micrometer.

The PCB layout was submitted to *Advanced Circuits* through their company website to fabricate the *SansTrumpet* PCB.

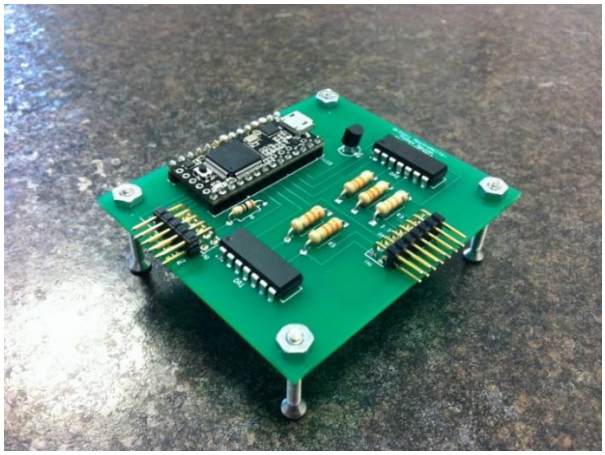

Figure 34a: Final PCB Product with Components Mounted

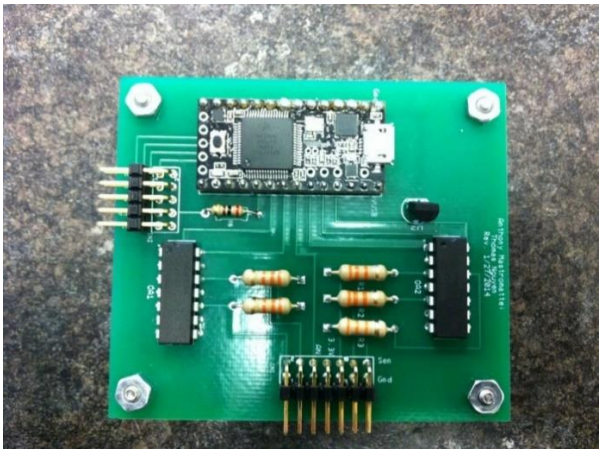

Figure 34b: Final PCB Product with Components Mounted (Top View)

As displayed in Figures 34a and 34b, the PCB was populated with the components that were discussed in the schematic. As shown, four bolts are utilized for the purpose of moutning the PCB to the enclosure. For the sensor inputs, as well as calibration components, right angle male headers were implemented. By using right angle headers, female headers can be be used to easily connect the sensors to the PCB and allow for modular replacement in case of component failure.

5.5 Power Budget

As indicated, the USB 2.0 can provide up to 100mA of current at 5V initially, and up to 500mA, if a request message is sent [66]. With any circuitry, the power budget must be analyzed to estimate the feasibility of the circuit based on power usage. From the USB standard, the available power is found by simply taking the product of the supplied voltage and current resulting in 500mW. If 500mA is requested, the available power can be increased to 2.5W. For the purpose of the power budget analysis, the case of a 100mA USB current will be used. The components of the circuit as seen in the final schematic consists of 5 voltage dividers(33kΩ resistors with flex sensors), 5 op-amps, two LED's(with 200Ω resistors), one proximity sensor, and 1 voltage regulator. The power supplied to these components are shared with the microcontroller, specifically the *Teensy 3.0*. Thus, it is imperative to verify that the power usage of the components will not impede the microcontroller.

The current drawn for each divider can be calculated with the use of *Ohm's Law*. As seen in the final circuit schematic, the resistors chosen are 33kΩ in series with a flex sensor. The calculations are based on two scenarios, the nominal resistance value for the flex sensor pertaining to no bend and the maximum resistance value for a 90 degree bend. For the maximum current through the voltage divider, the flex sensor is assumed to be 25kΩ at no bend, which is the minimum resistance of the flex sensor. The voltage supplied to each divider is the 3.3V from the votlage regulator. Therefore, with *Ohm's Law*, the resulting current is calculated to be 56.90μA per divider. The power, as indicated earlier, can be calculated by simply finding the product of the voltage and the current, which results in a power consumption of 0.188mW for each divider. There are 5 voltage dividers as seen in Figure 31, therefore, the total current drawn by the dividers is 0.285mA and the total power consumed by the voltage dividers is 0.940mW. Similarly, for the scenario with the flex sensors at a 90 degree bend for maximum resistance (125 kΩ), the current

and power consumption results to be 20.89µA and 68.94µW per divider and 0.104mA and 0.345mW in total.

The data sheet for the MCP604-I/P-ND op-amp indicates that the maximum current drawn is 325µA per op-amp [73]. Therefore, the 5 op-amps will draw a total of 1.625mA. Notice that in the schematic, the supplied voltage to the op-amps is also the 3.3V regulated voltage. Thus, the calculated power consumption per op-amp is 1.07mW and total power consumption of the op-amps is 5.36mW. The voltage source for the proximity sensor is also the regulated 3.3V. According to the data sheet, for 3V input, the current draw is approximately 2mA [75]. Therefore the power consumption is approximately 6.6mW. As seen in the schematic, the two LED's utilized are sourced by 3.3V and are in series with 200Ω resistors. Due to the resistors, the current drawn is approximately 16.5mA; therefore each LED branch (LED + resistor) consumes 54.45mW. The data sheet of the MCP1700-3302E/TO-ND voltage regulator indicates that the maximum quiescent current is 4.0µA [67]. The quiescent current, as explained earlier, refers to the current drawn in order for the regulator to function properly. The total current drawn from the regulator would therefore be the sum of the load current and the quiescent current. For this application, the "load current" is the current drawn from the components.

Table 13: Power Calculation for Flex Sensor at 0 Degree Bend(25kΩ)

Component	Voltage	Current Drawn	Power Consumption
Voltage Dividers	3.3V	0.285mA	0.940mW
Op-Amps	3.3V	1.625mA	5.36mW
Proximity Sensor	3.3V	2mA	6.6mW
LEDs(w/ Resistors)	3.3V	33.0mA	108.9mW
Totals	**N/A**	**36.91mA**	**121.8mW**

Table 14: Power Calculation for Flex Sensor at 90 Degree Bend(125kΩ)

Component	Voltage	Current Drawn	Power Consumption
Voltage Dividers	3.3V	0.104mA	0.345mW
Op-Amps	3.3V	1.625mA	5.36mW
Proximity Sensor	3.3V	2mA	6.6mW
LEDs(w/ Resistors)	3.3V	33.0mA	108.9mW
Totals	**N/A**	**36.73mA**	**121.2mW**

Tables 13 and 14 summarize the calculated current draw and power consumption of the components. With the values of current drawn known, the total current through the regulator can be found for the cases of 0 and 90 degree bend of the flex sensors. Therefore, the total current drawn through the regulator is 36.914mA and 36.734mA for the 0 and 90 degree bend of the flex sensors respectively. The voltage across the regulator is 1.7V since it is utilized to drop the 5V USB voltage down to 3.3V. Therefore, the power consumption of the regulator is 62.75mW and 62.45mW for the 0 and 90 degree scenarios respectively. With the components drawing 36.914mA and consuming 184.55mW at most based on these calculations, the remaining current and power available for the *Teensy 3.0* is 63.09mA and 315.45mW. This value is an estimate for worst case scenario such that all components are drawing the maximum current at the same time. Notice that the LEDs are responsible for the majority of current drawn and power consumption. However in a practical scenario, it is expected that only one of the LED would be on rather than both at the same time. If both LEDs were to be on at the same time, the calibration LED would only be functioning for a short duration (as defined in the software) due to how it is utilized for the calibration. Therefore the actual current drawn and power consumption is expected to be significantly less than this estimate.

A microcontroller's power consumption depends on which features of the microcontroller are enabled or disabled. PJRC, the developers of the *Teensy* microcontrollers, summarized the typical current draws at different configurations of the *Teensy 2.0* and *Teensy ++ 2.0*, however, this does not indicate the current draw of the *Teensy 3.0* [76]. Paul Stoffregen, one of PJRC's employees, states that the *Teensy 3.0* draws less current than the *Teensy ++ 2.0* [77]. The data sheet of the *Teensy 3.0* indicates that in the "run mode", with all clocks enabled and executing code from the flash memory, the current draw is 18.4mA [78]. Since the *Teensy 3.0* runs on 3.3V, the power consumed is 60.72mW. Hence, the circuit as a whole (components and *Teensy 3.0*) consumes a total of 245.27mW of the available 500mW. This value indicates that the remaining 63.09mA is enough for the *Teensy 3.0*. The *Teensy ++ 2.0* draws a maximum current of 60.2 mA at a voltage of 5V, with a clock frequency of 16MHz and the USB feature enabled, and running at 100% [76]. If the *Teensy 3.0* drew a similar amount of current as the *Teensy ++ 2.0*, then the 63.09mA available current should still be sufficient. In this scenario, the power consumed by the *Teensy 3.*0 would be 198.66mW and therefore the circuit consumes a total of 383.21mW from the

available 500mW. As a result, the chosen components are viable for the circuit design in terms of power consumption.

5.6 Budget

The cost for the various material required to construct the *SansTrumpet* based on the final design was calculated. Table 15 shown below portrays the major components of this device, as well as their respective costs:

Table 15: Budget Table for *SansTrumpet* Materials

Item	Qty	Description	Unit	Subtotal
1	**1**	Teensy 3.0	19.0000	19.00
2	**1**	Enclosure	12.0000	12.00
3	**1**	Ribbon Cable (10 ft)	5.0000	5.00
4	**1**	PCB Fabrication	33.0000	33.00
5	**5**	Resistor	1.0000	5.00
6	**4**	Flex Sensor	7.9500	31.80
7	**1**	Proximity Sensor	24.9500	24.95
8	**1**	Illuminated Switch	3.5000	3.50
9	**3**	Push Buttons	2.5000	7.50
10	**1**	Green LED	3.0000	3.00
11	**1**	Red LED	2.0000	2.00
12	**1**	USB Cable	6.0000	6.00
13	**1**	USB Adapter	3.9500	3.95
14	**1**	Isotoner Gloves	10.0000	10.00
15	**6**	Stereo Panel Mounted Audio Jack	1.7500	10.50
16	**11**	Male Audio Jack	1.5000	16.50
17	**5**	Female Audio Jack	1.7500	8.75
18	**2**	Op-Amp Arrays	1.2500	2.50
19	**1**	Voltage Regulator	0.4400	0.44
				$205.39
			TotalCost	**205.3900**

The budget for this project is $125 per student and therefore the total budget is $250. As indicated in the value analysis, the team selected the *Teensy 3.0* as the analysis portrayed that it would be the best fit out of the five selected microcontrollers/computing devices for the project. As indicated by the overall *SansTrumpet* design, four 2.2” flex sensors was required. For each flex

sensor, a buffer is required; thus, five op-amps are necessary. As shown in budget table, two Op-Amp arrays were purchased in consideration of size. The Op-Amp arrays each contain four individual amplifiers, for a total of eight. One of the major components as indicated in the budget table is the proximity sensor. Another major component of the *SansTrumpet* was the *Isotoner* gloves, which was utilized to support the flex and proximity sensors. Other various parts include the audio jacks (male and female) and ribbon cables that were used to interconnect the sensors to the control unit. Notice that one ribbon cable was purchased rather than two. This is due to the fact that the team decided that a length of 5 feet was sufficient. Notice that the most expensive components for the project were the sensors and the PCB fabrication. Several components in which the Team considered to be minor parts were omitted from the table. Such components include basic wires and header pins. The budget table shown in Table 15 does not entirely represent the budget of the project as a whole. Prior to the final design, an initial prototype was constructed as indicated in the initial design section. However, a variety of components were transferred from the initial design to the final design, such as the *Teensy* microcontroller.

6. Software

The software was written in the C language using the Arduino development environment and utilizes the *Arduino* and *Teensyduino* library when applicable (analogRead, usbMIDI, Serial, etc.). This code is designed in the usual *Arduino* format, where two functions are expected to be present: setup() and loop(). The Arduino runtime environment calls setup() once, which is typically reserved for initializations, followed by continuous calls to loop(). The loop() function contains the actual logical for the intended embedded system. A call to loop() is an opportunity to read the analog pins and process accordingly. Once the loop() function services the current pin values and exists, loop() is immediately called again, forever until the unit powers off (USB disconnected).

6.1 Initial Code

For the purposes of testing, basic software was developed to verify that a physical movement such as a bend of the flex sensor can interface with the computer digitally. Specifically, code to serially print the digital value of the analog value at one of the input pins was written. Code to verify that the bend of the flex sensor can produce a digital sound on the Digital Audio Workstation (DAW) was then developed. Shown in Appendix A and B are the two simple test programs. The basic functionality of the code shown in Appendix B is that when the flex sensor is bent to a certain degree, a MIDI ON message will be sent via USB to the DAW. For this example, a threshold value of 550 was used. As indicated earlier, the resistor value was selected such that at a 0° bend, the voltage value would be approximately half of the input voltage (3.3V), therefore the expected digital value is approximately 512. Once bent significantly, the voltage across the flex sensor will increase, and therefore the digital value will also increase. Once the threshold value of 550 is exceeded, MIDI ON message is sent via USB to the DAW to sound a note. Otherwise, a MIDI OFF message is sent to turn the note off.

The team initially wrote software for the amplitude as shown in Appendix C. The concept is based on manually adjusting the amplitude envelope based on trigger position. However, the user would only be able to control the velocity with a predetermined sustain (directly related to the velocity). The ability to separately control velocity and sustain is rather essential for a musician. This issue was addressed by utilizing a counter concept as shown in the code in Appendix D. Preset values for sustain were implemented such that for a specific range of counter values, the

corresponding sustain parameter is used. Although the ability to control the velocity was maintained with the user's trigger finger position, this approach introduced latency. The latency occurs due to the user continuously selecting sustain and velocity value for each pitch while still hearing the previously produced pitch. This problem was addressed with the "Trigger Reset" solution utilizing a state machine (refer to 6.4.2), which allows a pitch to sustain indefinitely until the user directs the trigger into the reset position (and stops the pitch).

6.2. Global Data

The software implementation maintains data, which is shared among the entire execution and is visible throughout all functions. The global data found in this software is described below:

GLOBAL VARIABLE	**COMMENT**
`state`	Begin at the Initial state
`currTrigger`	Current Trigger analog read
`prevTrigger`	Previous Trigger analog read
`currPitch`	Current pitch
`isDeviceCalib`	Device calibrated flag
`calibValve1Threshold`	Calibrated Valve 1 on/off threshold
`calibValve2Threshold`	Calibrated Valve 2 on/off threshold
`calibValve3Threshold`	Calibrated Valve 3 on/off threshold
`calibTriggerSweep[MIDI_RANGE]`	Calibrated Trigger sweep
`calibEmbouchureSweep[MIDI_RANGE]`	Calibrated Embouchure sweep
`pitchBuffer[PITCH_BUFFER_SIZE]`	Track pitches sounded in circular buffer
`isPitchOff[MIDI_RANGE]`	Track which pitches are reset
`pitchIndex`	Pitch buffer index
`embouchureReg1-6[EMBOUCHURE_NUM_ENTRIES]`	Registers 1-6
`fingeringStr[8]`	Debug fingering strings

6.3 Functions

The following are user-defined functions to support the state-machine processing. Each function accepts input and returns data when applicable.

6.3.1 setup ()

The purpose of the setup() function is to initialize the *Arduino* framework for implementation. This includes setting the pins to "input", the analog resolution to 10 bits, and setting the trigger state machine to "INITIAL".

6.3.2 loop ()

The loop () function controls the entire algorithm by focusing on the trigger behavior. The trigger controls when to sound the pitch, in particular, it establishes the attack (velocity) and release of its amplitude envelope. The attack is determined by the trigger sweep depth position, while the release is determined by the trigger reset position. In an effort to reduce trigger fluctuation readings, a 100 ms delay is executed per loop () call. This delay allows the sensor to perform greater movements between readings, thus, returning more predictable values. This is especially crucial in determining forward and reverse direction since an immediate reverse direction signals the sounding of a pitch. Since this function is continually being called by the Arduino runtime environment, the trigger algorithm must maintain the "state" global variable. Therefore, it's implemented as a state machine as shown in the flow chart below depicted in Figure 35b.

6.3.3 pitchOn ()

The pitchOn function is called when the trigger state machine determines that a pitch is to be sounded. It references all the necessary inputs to calculate the expected pitch, which include the following:

1. Calculate the MIDI velocity based on trigger sweep depth
2. Read valve sensors and calculate a fingering value
3. Determine embouchure register from embouchure sensor position
4. Extract pitch by indexing embouchure register with fingering value

5. Ignore a dead pitch when encountered (when a particular fingering does not sound a pitch in the corresponding embouchure register)
6. Sound the MIDI pitch with the associated velocity

6.3.4 getEmbouchureReg ()

The purpose of the getEmbouchureReg () function is to return the embouchure register associated with the embouchure sensor position. Thus, six embouchure registers are implemented and the intended register is calculated by dividing the embouchure sweep position by the number of one register intervals.

6.3.5 readFlexSensor ()

This function returns a sampling of a FlexSensor reading, based on a pin input as its arguments.

6.3.6 loadCalibData ()

This function loads all the calibration data from EEPROM flash memory, for each sensor (Valves, Embouchure, and Trigger). It returns a 'true' value when successful, otherwise a 'false' indicates an uncalibrated device. The Teensy EEPROM interface is byte based but the sensor readings require at least two bytes

6.3.7 readEepromInt16 ()

This function reads and returns a 16bit integer from EEPROM at a specified address. Shifts and or's two bytes.

6.3.8 writeEepromInt16 ()

This function writes a 16bit integer to EEPROM at a specified address. Shifts and or's two bytes.

6.3.9 blinkCalibLED ()

This function blinks the Calibration LED for a specified amount of ms.

6.3.10 checkCalibButtons ()

This function checks whether the user intends to calibrate by checking the corresponding calibration buttons for Trigger, Valves or Embouchure. Calls the appropriate calibration function when applicable. Returns 'true' when all data is calibrated, otherwise returns 'false'. Note - only one button can be active at one time.

6.3.11 uncalibLEDpattern ()

This function blinks an LED sequence to indicate the device is not calibrated yet.

6.3.12 calibValves ()

This function calibrates the valves but assures the calibration button has first been pressed for 3 seconds. The calibration takes the current reading from the valves bend position which are assumed to be where the user expects the on/off threshold to be. Once calibration is complete, the calibration LED blinks for 1 second.

6.3.13 calibSweep ()

This function calibrates the Embouchure and Trigger sweep but assures the calibration button has first been pressed for 3 seconds. During calibration the LED blinks continuously showing calibration is in progress. Calibration entails the 3 second sweep recording of 128 readings of either the Trigger or Embouchure sensor, depending on which is passed in as an argument.

6.3.14 lookupTriggerVelocity ()

This function returns the Trigger velocity by searching for the closest approximation of the current Trigger in the Trigger Sweep Calibration array. Once found, the index (0-127) of that entry represents the pitch velocity and returned.

6.3.15 allPitchesOff ()

This function simulates the Control-Change-All-Notes-Off functionality (All-Pitches-Off) since the actual MIDI message doesn't perform to specification standards. Traverse the pitch

buffer array, shutting each pitch off. Also, track the pitches shut off to avoid redundant MIDI messages.

6.4 Flowchart

This section illustrates the flowchart of the software implementation. The flow chart is depicted on the following pages. Note – the red process boxes represent flowchart expansions (on separate pages).

FLOWCHART

IMPLEMENTED SENSORS:

Valves:
Right hand 3 fingers

Embouchure:
Left hand palm proximity sensor

Trigger:
Left hand index finger

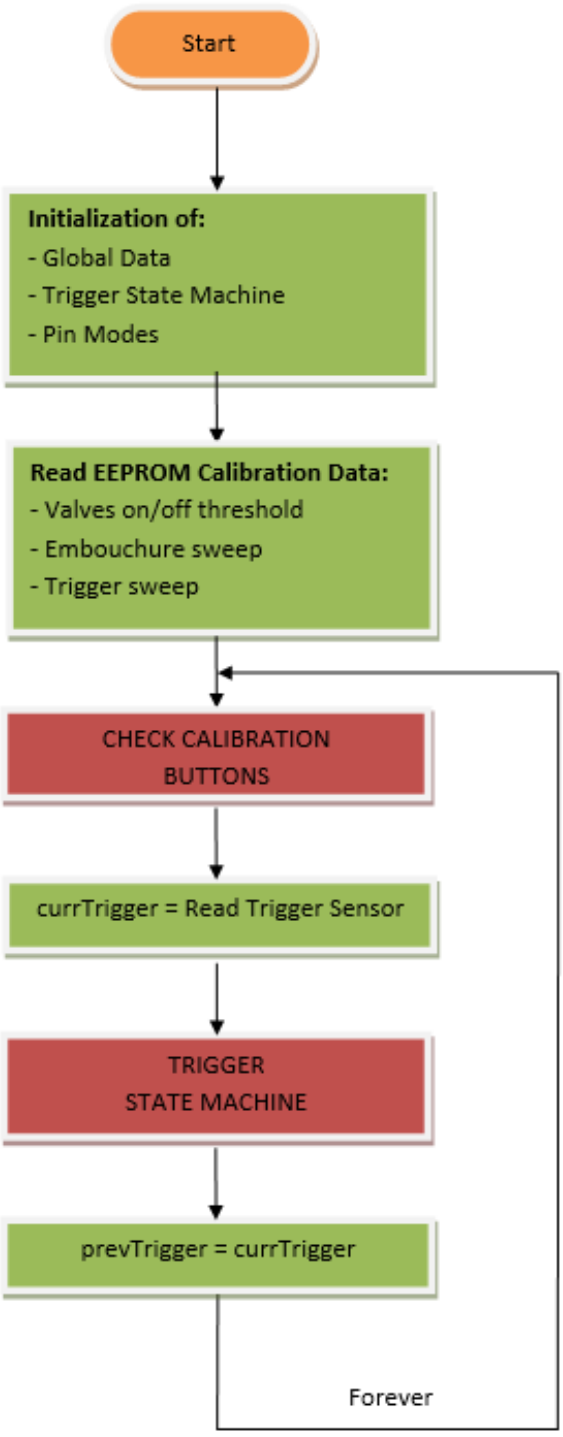

Figure 35a: Flow Chart (Part 1)

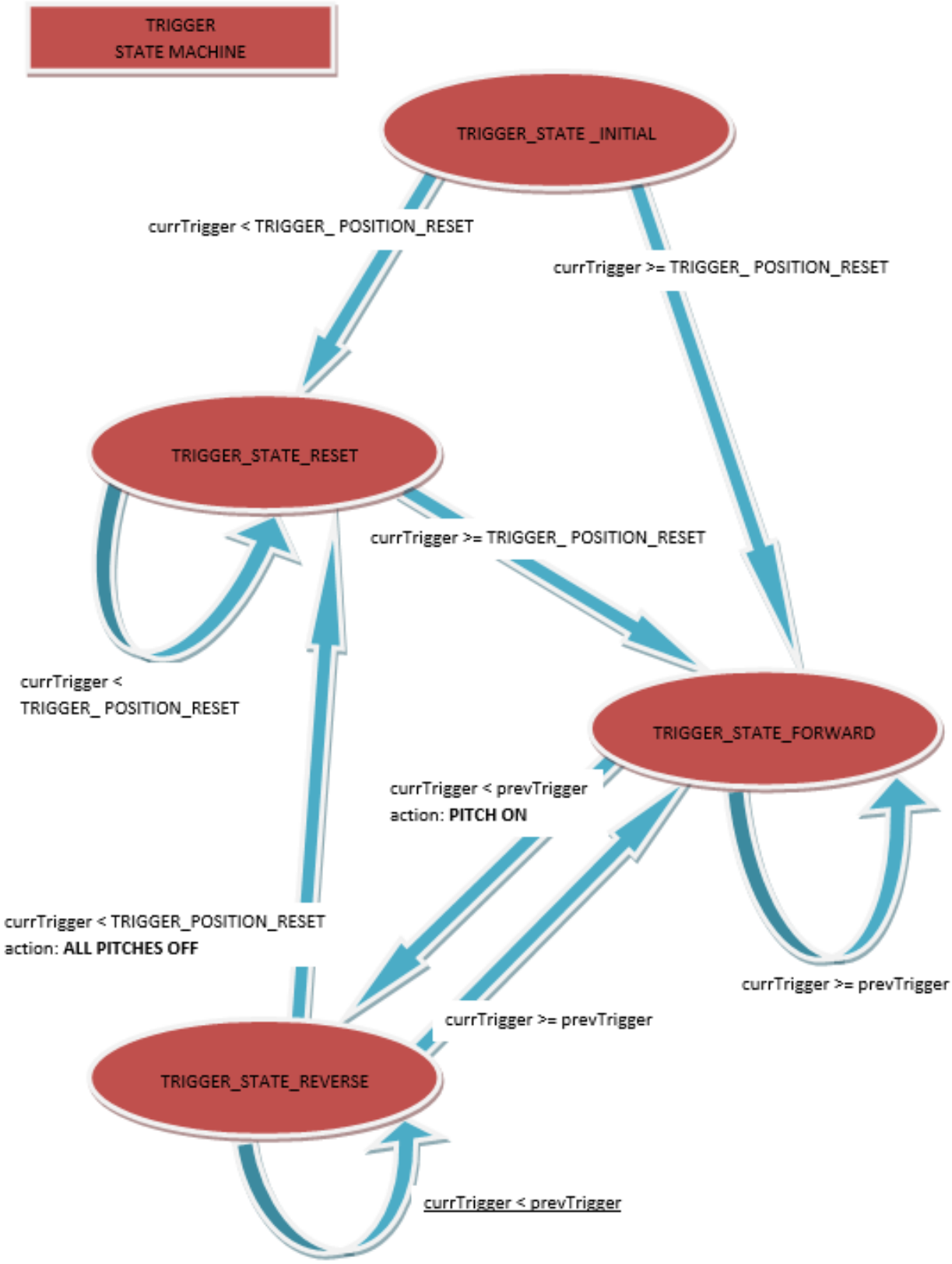

Figure 35b: Flow Chart (Part 2)

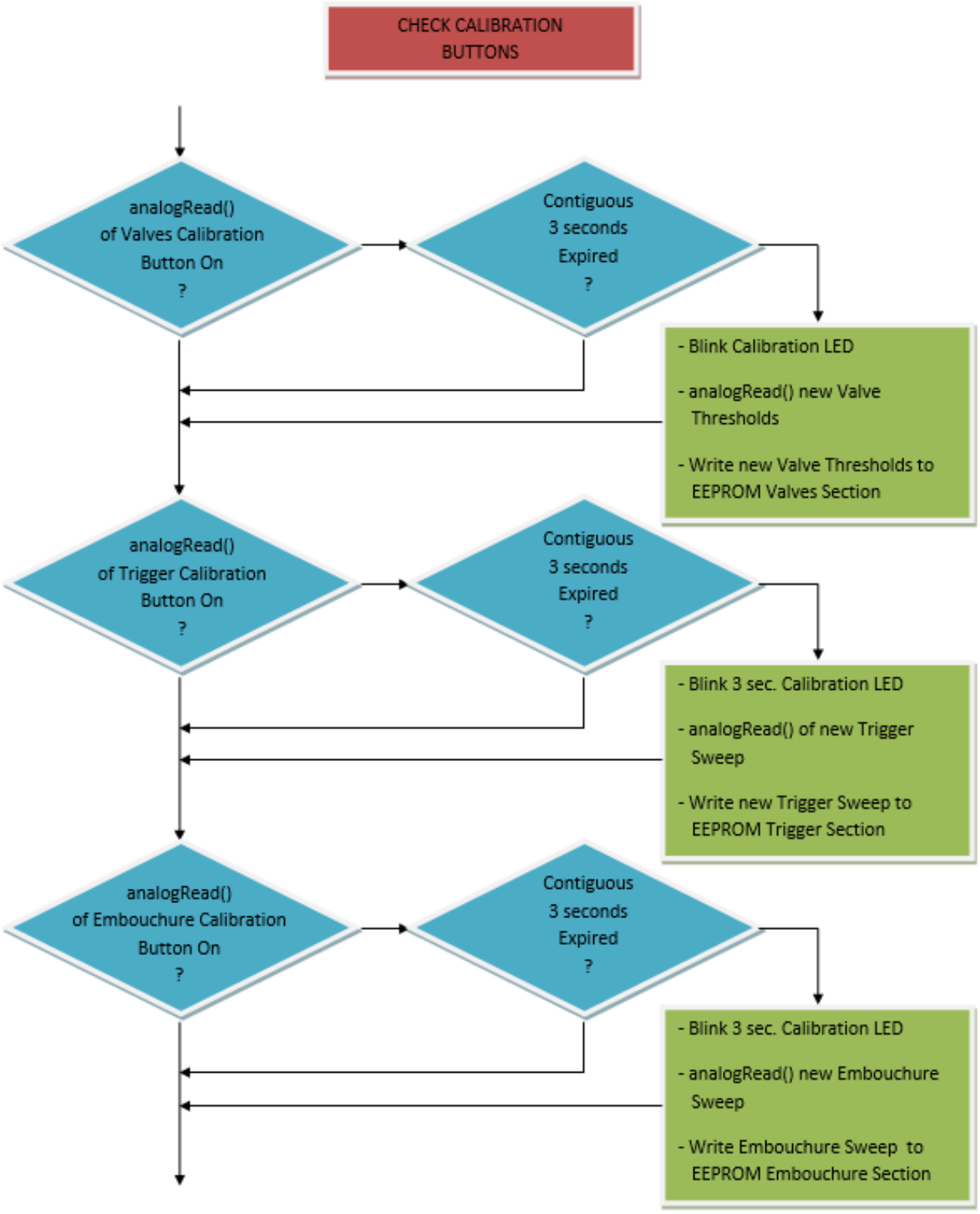

Figure 35c: Flow Chart (Part 3)

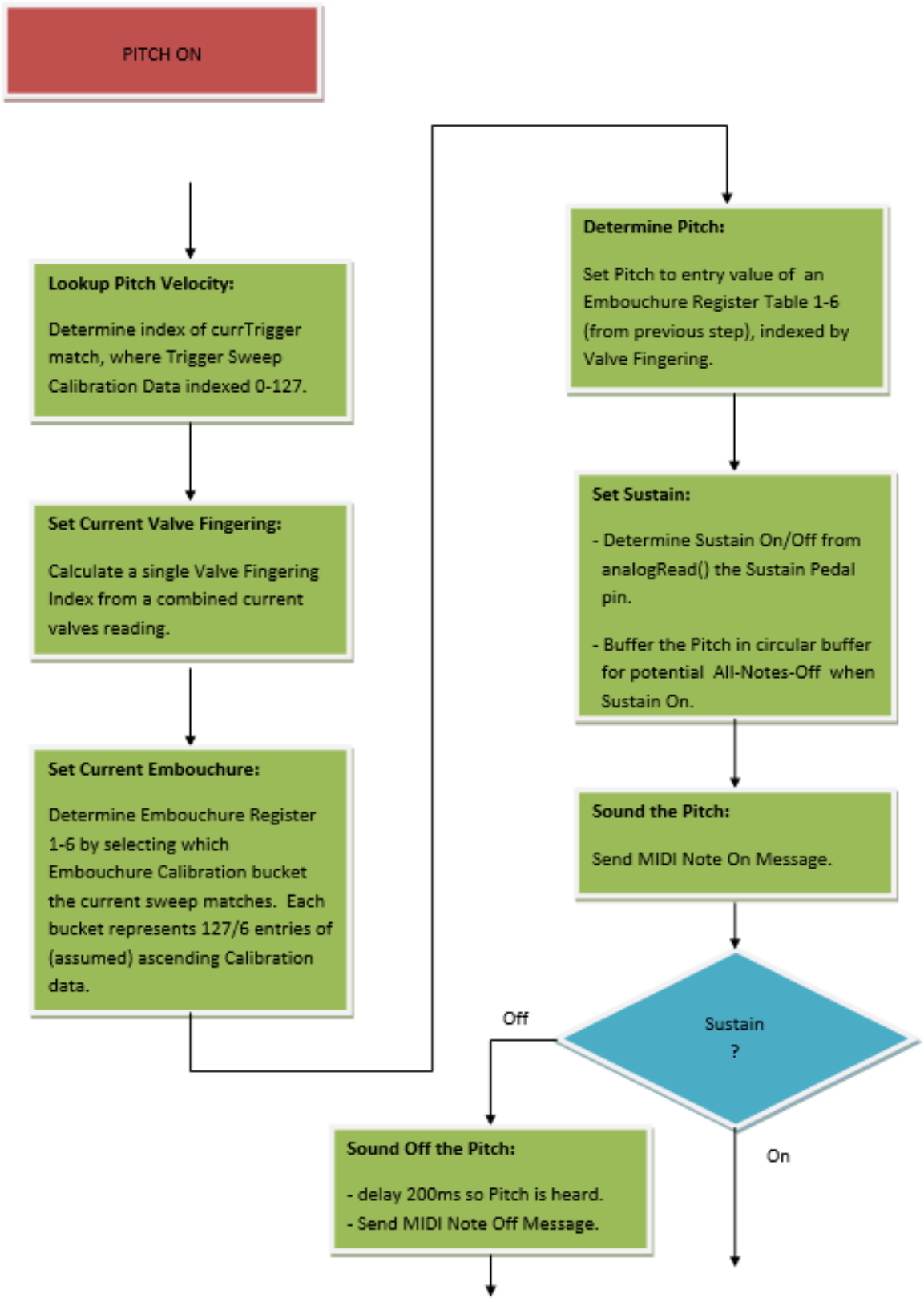

Figure 35d: Flow Chart (Part 4)

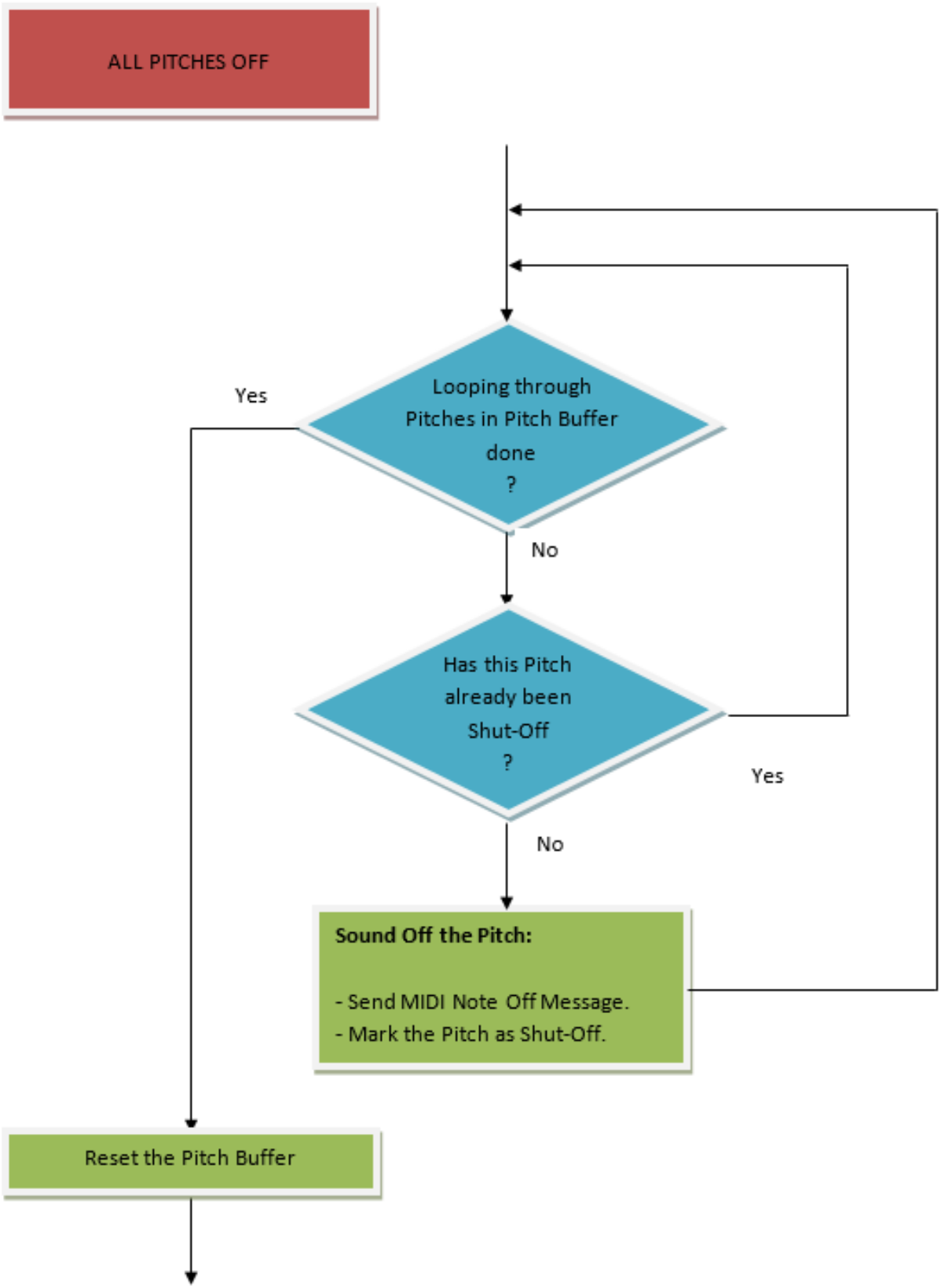

Figure 35e: Flow Chart (Part 5)

6.4.1 Top Level Design

To visualize the algorithm, a flowchart was developed as depicted in Figures 35a to 35e. The top-level design initializes Global Data, the Trigger State Machine, and the pin modes. In addition, it reads the EEPROM Calibration Data followed by a "forever loop" which: checks whether the calibration buttons are being pressed, sets the current trigger and executes the Trigger State Machine. Lastly, sets the previous trigger and repeats.

6.4.2 Trigger State Machine

The Trigger State Machine is depicted in flow chart Figures 35b. The state machine continuously compares the current trigger value and the previous value to determine the direction of the trigger. In addition, it determines the functionality of the device depending on the direction of the trigger. As stated above, the trigger state is initialized to TRIGGER_STATE_INTIAL and enabled when the unit powers on (USB connected). This state examines if the currTrigger is in the reset state (before the trigger threshold) or forward state (after the trigger threshold). If the currTrigger is less than the trigger threshold, then the unit is in the state, TRIGGER_STATE_RESET. This state will continuously loop until the currTrigger is greater than or equal to the trigger threshold, which means that the trigger is now in the forward state (TRIGGER_STATE_FORWARD). Here, the software will continue to loop while the currTrigger is greater than or equal to the prevTrigger. When the currTrigger is finally less than the prevTrigger, this implies that a pitch intends to be "triggered" and the state transitions to the TRIGGER_REVERSE_STATE. The functionality of the trigger pitch action can be referred to in the pitchOn function above. The device remains in this state while the currTrigger is less than the prevTrigger. However, the user then has two options: move the trigger forward (currTrigger >= prevTrigger) to return to the forward state (TRIGGER_STATE_FORWARD) or to go back to the reset state (TRIGGER_STATE_RESET), where the currTrigger is less than the trigger threshold. If the player chooses to go back to the forward state, they will be able to sound another pitch, at the velocity of their choosing, while the previous triggered pitch sustains. This process can encompass multiple pitches. Contrarily, if the user decides to bring the trigger back to the reset state, then a MIDI control message is sent that sets all pitches off.

6.5 Debug Modes

The code can be compiled with "debug mode switches" whenever any aspect of the device is questionable. These switches can be turned on/off by simply enabling their definition or prefixing the symbol name with an "x" (or anything to change the symbol name). When such a symbol is on, the debug code becomes exposed to the compiler and outputs data onto the Serial Monitor screen providing insight into the execution. The following debug modes and their purpose are:

Table 16: Debug Modes

DEBUG SYMBOL	COMMENT
`EXECUTION_MODE_DEBUG_CYCLE`	Display activity of the state-machine such as the current state and the current Trigger value.
`EXECUTION_MODE_DEBUG_PITCH`	Display when the Reset state is reached and every attribute of a pitch such as: - Velocity - Fingering - Embouchure sweep - Pitch - Sustain flag
`EXECUTION_MODE_DEBUG_CALIB`	Display data regarding stored EEPROM Calibration such as: - Hardware and Software version - Valves calibration - Trigger calibration - Embouchure calibration
`EXECUTION_MODE_MIDI`	Controls whether MIDI messages are transmitted or not. Useful when debugging.
`EXECUTION_MODE_DEBUG_VALVES`	Displays pitch, valve fingering and valves thresholds

6.6 Calibration and Sampling

To address variable resistances from each flex sensor, a combination of sampling and storing calibration data was implemented. This approach proved effective. As the software requests a particular sensor reading, an average reading is taken instead, based on the number of readings. At the moment the readings per sample are specified as:

```
#define READINGS_PER_SAMPLE 2000
```

This can conveniently be changed, however, this current setting proved to be the minimum value which yields the most consistent results at a standstill.

The next approach stores calibration data to EEPROM memory for each sensor including the Trigger, Embouchure, and Valves. Each sensor has a dedicated Calibration momentary pushbutton to engage the Calibration function. The Calibration function when activated engages the user to customize either a “sweep” for the Embouchure and Trigger, or an “on/off threshold” for the Valves. To engage the Calibration function, the user is instructed (refer to usual manual) to hold down the corresponding Calibration button for 3 seconds. Upon detecting a Calibration being pressed, the software exits the state-machine and tracks a continuous hold for 3 seconds. If at any point the hold is released prior to the 3 seconds, the software returns to the state machine, uninterrupted. A continuous 3 second hold executes the associated Calibration function.

Valves Calibration involves reading the current Valves sensors and storing them to EEPROM. Sweep Calibration for the Embouchure and Trigger sensors involve a continuous reading spread evenly over 3 seconds of 127 readings. 127 readings are chosen due to the MIDI range specifications for velocity, which apply to the Trigger sensor. For engineering simplicity, 127 readings were also applied to the Embouchure sweep, which is partitioned into 6 ranges (as to prior implementation). As the Calibration data is written to EEPROM, it also becomes utilized for current use. When the unit is power-cycled, the software, as part of its initialization, loads the Calibration data from EEPROM and, again, utilizes it for current use. To indicate the Calibration activity, a dedicated LED is provided on the faceplate of the enclosure. Following the 3 second hold of a Calibration button, the LED blinks once for storing the Valves threshold to EEPROM

and blinks 127 times during a 3 second sweep capture, where each blink represents one write to EEPROM memory.

As the software executes and sensor readings are returned, the Calibration data is used as a lookup to "normalize" the data regardless of the sensor. For example, the Trigger velocity according to MIDI specs always ranges between 0-127 and, thus, the Trigger Calibration table is 128 entries. When the Trigger sensor returns a reading, the lookup determines where in the Calibration table the reading "fits", at which point, the entry index is returned. Therefore, a value of 0-127 is always returned for a Calibrated sensor. It is significant to mention that a Calibrated sensor should not be relocated unless re-calibrated, otherwise, unpredictable behaviors will result.

6.7 Embedded Version Numbering

In an effort to debug a *SansTrumpet* device, the version of hardware and software can always be referenced from the first 4 bytes of EEPROM (refer to EEPROM Memory Format below). Every new version of code loaded to the *Teensy* will detect whether the device has ever been calibrated. If the first 2 bytes of each Calibration section of EEPROM contain a 0xffff, the device is considered un-calibrated. When the software boots and is considered un-calibrated, the software writes the Hardware and Software version to the respective locations in EEPROM, in addition to flashing a special LED pattern. Once calibrated, this special processing is no longer performed. Therefore, this device can easily be debugged by simply reading the version from EEPROM and cross-referencing the source code to the device behavior.

6.8 EEPROM Memory Format

The *Arduino* software interface to the EEPROM is byte addressable, but the device data being stored is typically 2 bytes. Therefore, functions were written to assist in the read/write operations to properly format and address the data.

Table 17: EEPROM Memory Format

Address	Data (byte1)	Data (byte2)
0	HW-ver-major-rel-byte1	HW-ver-minor-rel-byte2
2	SW-ver-major-rel-byte1	SW-ver-minor-rel-byte2
4	Valve1-threshold-byte1	Valve1-threshold-byte2
6	Valve2-threshold-byte1	Valve1-threshold-byte2
8	Valve3-threshold-byte1	Valve1-threshold-byte2
10	Trigger-sweep[0]-byte1	Trigger-sweep[0]-byte2
12	Trigger-sweep[1]-byte1	Trigger-sweep[0]-byte2
14	Trigger-sweep[2]-byte1	Trigger-sweep[0]-byte2
…	…	…
264	Trigger-sweep[127]-byte1	Trigger-sweep[127]-byte2
266	Embouchure-sweep[0]-byte1	Embouchure-sweep[0]-byte2
268	Embouchure-sweep[1]-byte1	Embouchure-sweep[1]-byte2
270	Embouchure-sweep[2]-byte1	Embouchure-sweep[2]-byte2
…	…	…
520	Embouchure-sweep[127]-byte1	Embouchure-sweep[127]-byte2
…	…	…

6.9 Sustain Pedal

The Sustain pedal enables multiple pitches to resonate when the pedal switch is on. When it is off, only a single pitch is sounded at a time. This switch is implemented using a digital pin on the Teensy and when reading this pin it simply returns a 0 (off) or 1 (on), depending on the switch state. This value is reflected back to the code in the form of a boolean flag and acquired when the pitch is being formed. When the pitch is triggered, it will either be sustained or not depending on the Sustain flag. Originally, this function was intended to utilize the "All-Notes-Off" MIDI Control message. However, after testing with *Garageband*, it was determined this message was not supported. In an effort to assure every DAW can perform with the device, a specific algorithm was developed which does not depend on the MIDI "All-Notes-Off" message. This algorithm is based on a circular buffer that stores the last 1000 pitches, which can be configured via:

```
#define PITCH_BUFFER_SIZE  1000
```

When in Sustain mode, pitches are stored in the circular buffer and wrap when the buffer overflows. As the pedal switch is pressed and the transition from Sustain to Single-Pitch occurs, the circular buffer is traversed to send a single MIDI "Note-Off" message for every pitch triggered.

Alternatively, during Single-Pitch mode the circular buffer does not apply since the MIDI "Note-Off" message is always sent for the previous pitch that is always tracked.

This technique satisfies the sustain pedal requirements without a "All-Notes-Off" MIDI message since:

1) 1000 pitches may never be played during one performance.
2) If 1000+ pitches did get sounded, every pitch at one point should be accounted for a "Note-Off". In addition, if not accounted for, the pitch sustain would shut-off naturally from the DAW by the time 1000 pitches are sounded.
3) The algorithm is optimized to not send duplicate "Note-Off" messages for the same pitch, so performance is not affected negatively.

7. Physical Design

The physical design and appearance included the initial prototype of the enclosure, flex sensor cabling, and gloves. A requirement of the physical design was to be user friendly. Thus, it was decided the user should not have to be concerned with flex sensor problems from either damage or from wear-and-tear. For this reason, it was determined the wiring occurs all inside the enclosure and that modular cables (with flex sensors) would be developed to allow for simple exchange of damaged flex sensors. The modular cables connect to a sensor on one end and a male jack connector on the other end, for connection to a panel-mounted female jack on the enclosure. Finally, the flex sensors would attach to the hand by means of a glove.

7.1 Initial Physical Design

7.1.1 Enclosure

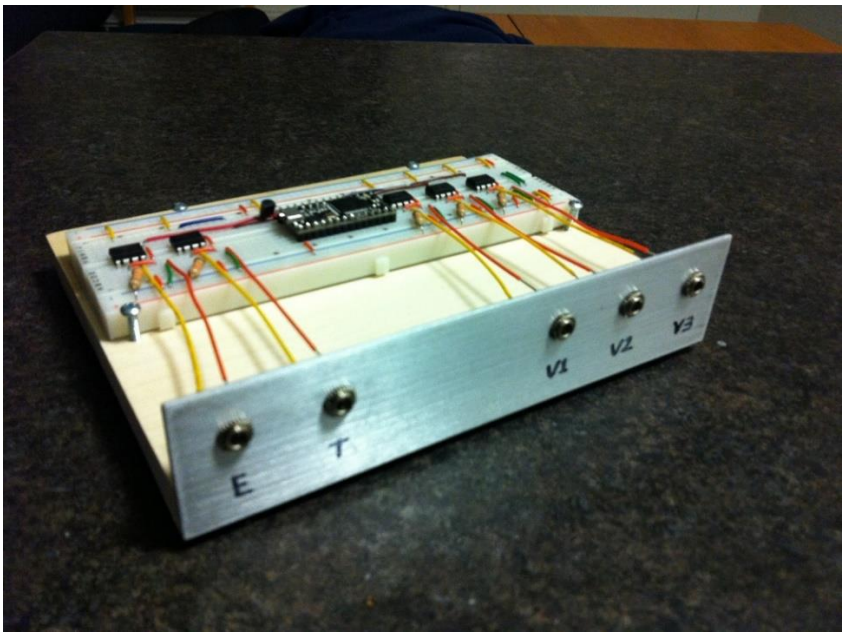

Figure 36: Initial Enclosure Prototype

The concept of the initial enclosure prototype was to hold the breadboard containing the circuitry and panel mount female connectors. The enclosure that was utilized was a scrap piece of a guitar pedal board. The enclosure has a wood base and an adjacent metal plate that is attached

with screws to the wood base. The breadboard was held firmly in place by positioning screws into the wood base, on the sides of the breadboard that would prevent it from moving. Holes were drilled out for the panel mounted female jacks. When the female jacks were placed on the metal plate, wires were then used to solder the leads of the connectors to the appropriate pin locations on the breadboard. Also, labels were written on the metal plate (near the female jacks) that denoted the appropriate inputs, such as embouchure register, trigger, or valves. The initial enclosure prototype is shown above in Figure 36.

7.1.2 Modular Sensor Units

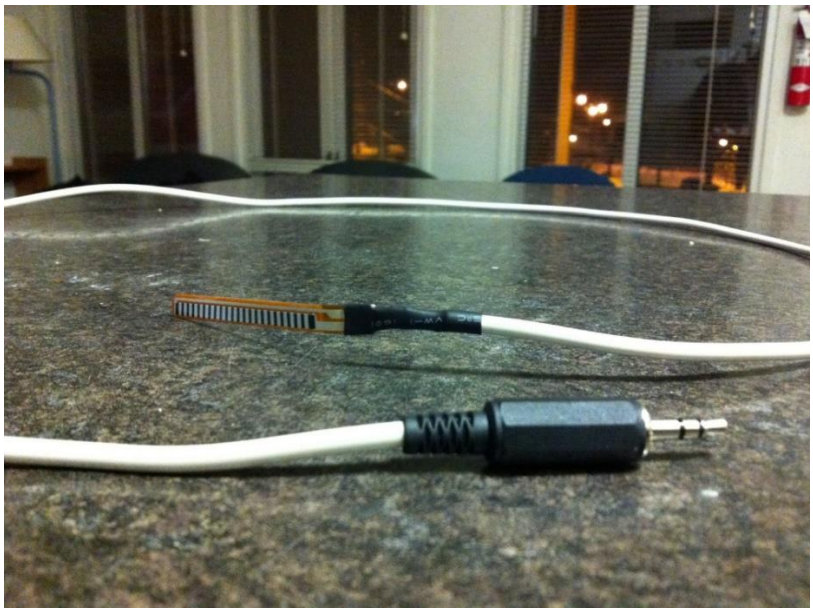

Figure 37: Modular Sensor Unit

The concept of the modular sensor units was to have a cable, whose internal wires would connect two components on each end: the male jack and the flex sensor. A telephone cable was utilized with 4 internal conductors, although, only two of the conductors were used. The cables were soldered onto both the male jack and flex sensor. Finally, heat shrink was applied to the leads of the flex sensor to protect the wiring, particularly the flex sensor leads which are known to be rather fragile when exposed. The modular sensor unit is shown above in Figure 37.

7.1.3 Glove Design

The flex sensors, from the modular sensor units, would then be inserted into the gloves. The initial design for the glove is shown below in Figure 38:

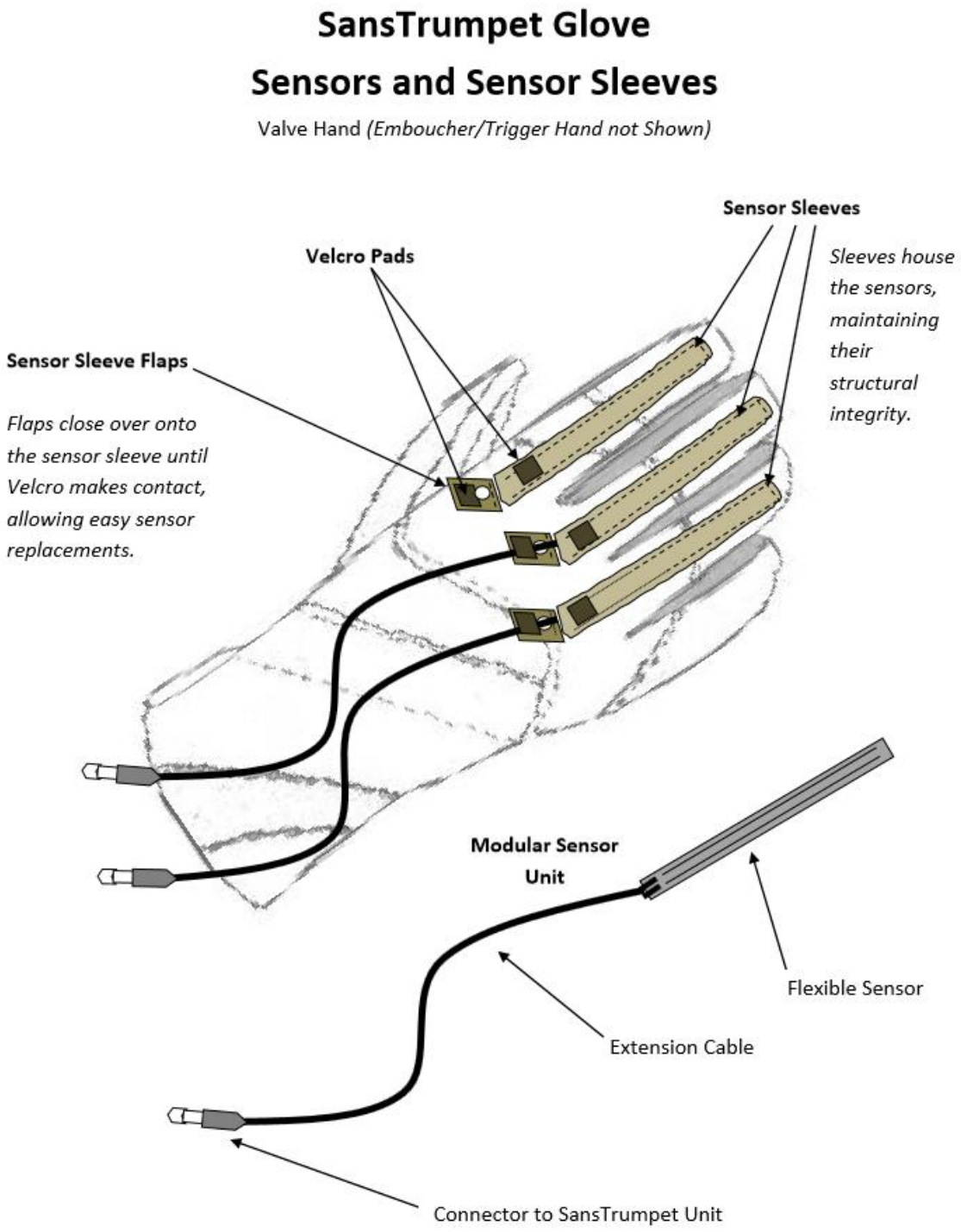

Figure 38: Initial Glove Layout/Design Concept

As shown, a fabric would be stitched on to the top of the finger regions on the glove that would act as the sleeves to house the flex sensors. To secure the modular units in place, the cables would be routed, male jack first, through a sleeve flap that contained Velcro. Initially it was intended to have the cable routed through the flap, the flex sensor would then be inserted into the sleeve and secured in place by positioning the Velcro flap onto the Velcro pad.

Thus, when the concept of the glove was planned accordingly, an *ISOTONER* glove was used as the basis of the glove design. A piece of fine fabric was hand stitched onto the gloves to form the sleeves: one on the left hand (representing the trigger finger) and three on the right hand (representing the valve fingering). Instead of using Velcro as indicated in the initial concept, lace was used to wrap around the flex sensor and secured into place by a button positioned right next to the sensor sleeve. The complete initial prototype set, with the glove included, is shown below in Figure 39:

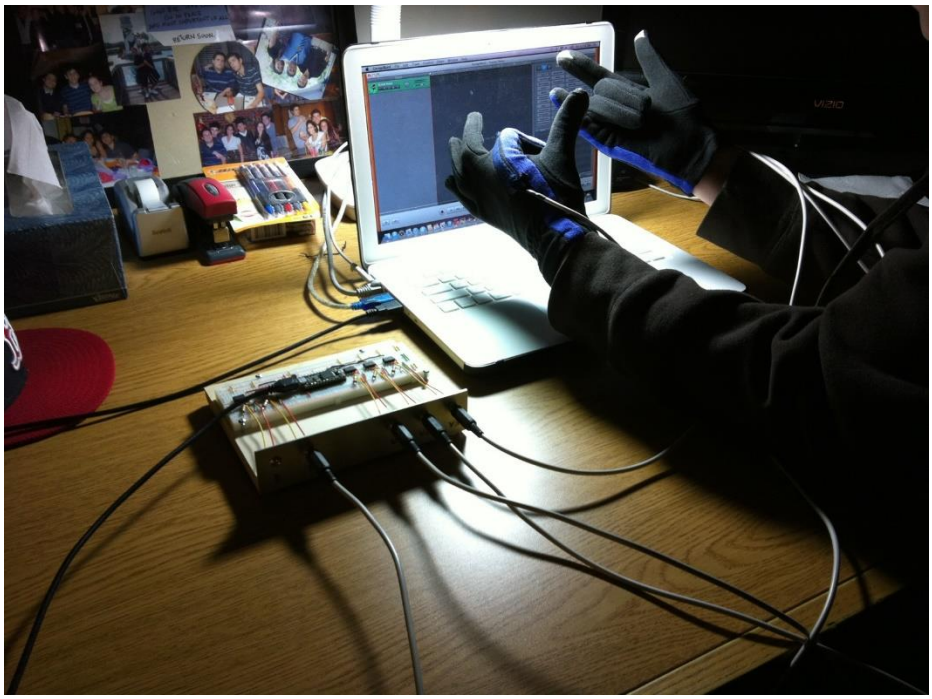

Figure 39: Initial Prototype Set

7.2 Final Physical Design

7.2.1 Enclosure

With the concept of the enclosure prototype that was constructed initially, a transition to a real enclosure had to be implemented once the PCB was obtained. Essentially, the purpose of the new enclosure design was to, first, panel mount the female jack connectors, push buttons, and LED, with their appropriate connections (just as the prototype), but also to house all the wiring associated with the circuit. In this fashion, the entire circuit is protected. The characteristics of the enclosure were that it would allow for ample space to house all electrical components of the design (including the USB adapter), would be ergonomically designed to place the female jack connectors (to insert the modular sensor units, in relation to the body), and that would look sleek, elegant, and light. As a result, an aluminum, trapezoidal design was purchased satisfying this criteria. The enclosure was also conveniently designed to place three female connectors on both sides of it. This was ergonomically significant because three modular sensor units are utilized on each side of the body.

Regarding the wiring inside the enclosure, it was decided the remaining components (LED and calibration buttons) should be placed on the faceplate, while the opening back-plate should be left free, with no wiring. The explanation for this configuration is to allow the user easy access to the inside of the enclosure and PCB. The USB adapter was chosen to be panel mounted on the long side of the trapezoid to allow for the most room to "snake" its cable underneath the PCB, and then angled around the top of the PCB into its appropriate female connection on the *Teensy*. The first task that was performed was drilling the holes for all the components. To accomplish this, a sketch was initially drawn of the exact measurement of the enclosure on graph paper. Then, the components, with their measurements, were also drawn in this sketch. When the desired locations were determined, blue masking tape was placed on the enclosure with the shape of the component drawn into their appropriate position. A drill bit was then utilized (like the prototype) to drill the holes. After, all the components were panel mounted onto their corresponding locations. Next, the PCB had to be populated. Thus, the resistors, op-amps, male headers, *Teensy*, and regulator, were all soldered to their appropriate pins on the PCB. After, the female header connectors (connected to the male headers) had to be constructed. The opposing end of these connectors had to be developed with 24-gauge wire being crimped onto a pin, soldering the pin, and placing it into the

appropriate pin location of the female header. For simple readability, the wires were color coded to represent certain parameters: black signified ground, red signified both the voltage to push button devices (calibration buttons and pedal switch), blue signified the signal from the trigger and embouchure (left glove), and white signified the remaining *Teensy* signals from the LED and pedal, as well as the valves (right glove). The populated PCB with female header connectors is displayed in Figure 40:

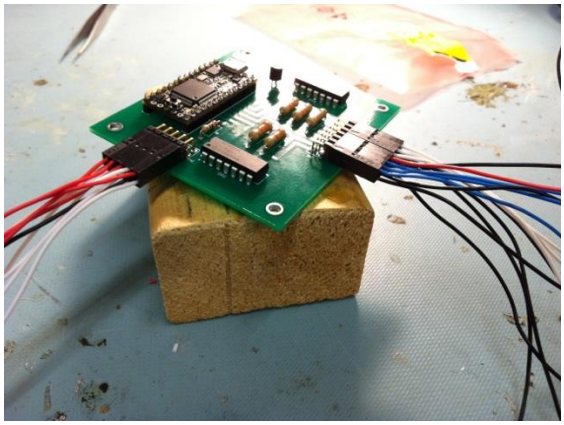

Figure 40: Populated PCB (Components and Female Header Connectors)

The wires from the female header connectors were then soldered to the suitable female jack connections. This configuration is shown in Figure 41 below:

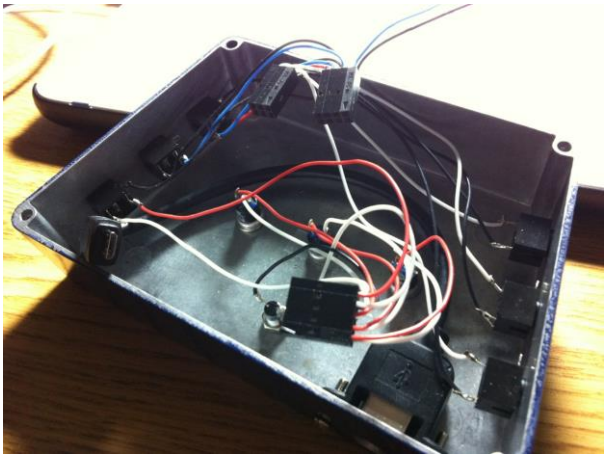

Figure 41: Wiring Configuration in Enclosure

The final task was to place the PCB into the enclosure. Measurements were made for the location of the PCB to allow ample space from the opening of the enclosure and the masking of

the USB adapter. To keep the PCB suspended above the wiring and USB adapter, screw posts were fastened onto the PCB mount holes and epoxied to the back of the faceplate. A screw post is created by cutting off the head of a screw. To hold the PCB level, nuts were placed on the top and bottom of the PCB. The PCB was tested to be level if the bottom of the posts could be placed firmly on the top of a level surface, such as a table, with no shims on either side. Modifications were then made, if appropriate, to the placement of the nuts to level the PCB. When this arrangement was set, the posts were then epoxied to the back of the faceplate. Subsequently, the female header connectors were "snaked" underneath the PCB, into their suitable male headers on the PCB and the male end of the USB adapter was inserted into the USB female connector on the *Teensy*. This configuration is demonstrated in Figure 42 below:

Figure 42: Interior of Enclosure (w/ PCB)

With all of the hardware components of the enclosure complete, the final goal was to find a mechanical clip that could be attached onto the back of user's belt for playability. The clip that was selected was a contractor's tape measure clip found at *Home Depot*. This clip utilizes two magnets to hold the desired object in place. Therefore, a metal plate contains the clip that is attached to the user's belt, while the magnet got screwed onto to the back-plate of the enclosure. Figure 43 displays the original concept of both magnets of the clip, in relationship to the opening of the enclosure as shown below:

Figure 43: Original Concept of Clip

Figure 44: Exterior of Enclosure (Opening)

Figure 45: Exterior of Enclosure (Faceplate)

As seen in Figure 44, one end of the magnet was screwed into the opening of the enclosure. Figure 45 displays the opposing end of the enclosure, the faceplate.

7.2.2 Glove Improvements

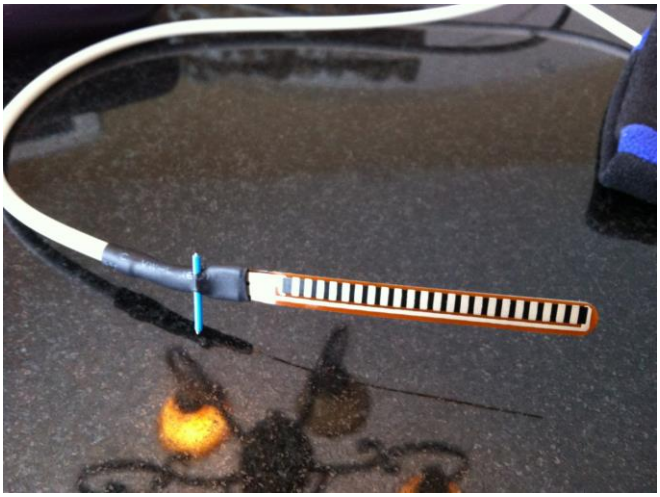

Figure 46: Modular Sensor Unit (w/ Anchor)

Figure 47: Modular Sensor Unit (on Glove)

A major problem of the glove design was found in which the flex sensors were not securely fastened into the sleeves and occasionally would come out. To resolve this issue, the concept of an anchor for each sensor was developed. The idea is to utilize the lace (originally used to secure the sensor) to wrap around the anchor, such as a wire, to improve how the sensor is secured. The placement of the anchor would be approximately at the bottom of the modular sensor unit when placed in the sleeve. To accomplish this, an additional layer of heat shrink was placed around the

sensor and original layer of heat shrink, an anchor (a thick piece of a paper clip) was placed in the appropriate location, and heat was applied to the heat shrink to condense. The unit was tested by applying tug and strain on the cable to examine if the sensor would detach, however, it was confirmed to be firmly secured. Figure 46 displays a modular sensor unit with an anchor, while Figure 47 displays the unit wrapped around the anchor on the glove.

Figure 48: Proximity Sensor Unit (on Glove)

The next task concerning the glove was how to integrate the proximity sensor. First, the modular sensor unit was created with 3 wires from inside the telephone cable: for Vcc, ground, and *Teensy* signal. Because these copper wires were too delicate to solder directly onto the proximity sensor pins, wires from the lab kit were soldered onto the copper wires on the sensor side of the cable. Next, heat shrink was applied around the cable, by just giving enough room for these wires to be clipped onto the proximity sensor for the time being. The next goal was how to fixate the sensor to point down once one's hand was at rest. This was accomplished by sewing the proximity sensor onto the glove via its mounting holes and spare pin-holes. Also, right below the heat shrink of the proximity sensor unit, extra threading was looped around the cable from the glove, which made the sensor more secure. Finally, the wires that were temporarily clipped on the appropriate pin locations on the proximity sensor were soldered. Figure 48 displays the integration of the proximity sensor on the glove.

7.2.3 Sustain Pedal Design

The sustain pedal was constructed by utilizing a single pole single throw illuminated pushbutton switch. The concept of the pedal was for the user to switch between a sustain mode and a single-note mode, where the light inside of the switch would alert the user which mode they were in, in a similar fashion to a guitar pedal. The cable of the pedal was chosen to be approximately twice as long as the rest of the units. The reason for this determination is the team did not want to constrain the user to a fixed location during performance. With this design, the user still has the flexibility to move around, while being able to tap the pedal at their preference.

In consideration of this project, one issue of the switch is that it requires 120 V_{AC} to light the bulb inside as opposed to a low DC voltage, which would be suitable for this design. However, there was an instruction guide online on how to replace the bulb with an LED to resolve this issue for a similar application (a guitar pedal) [79]. In order to accomplish this, the components within the switch must be replaced. The bulb that was being utilized had to be clipped, while the 330kΩ resistor had to be removed. These two components were exchanged with a red LED and a 200Ω resistor. The mini bulb was replaced with the LED and was situated in the same position. Therefore, the switch was reconstructed and the end result is a compatible switch that can function at low DC voltage while being capable of making the LED indication apparent.

Figure 49: Sustain Pedal

For the base of the pedal, a sofa foot was purchased at *Home Depot*. A hole was burrowed into the middle of this foot to suspend the switch from the inside of the foot, as well as a second

hole that was burrowed on the side to "snake" in the cable for the modular pedal unit. Four holes were then drilled into the foot, for placing an aluminum panel that would be screwed onto the top of the foot. Also, a hole was made in the middle of this aluminum plate to panel mount the switch. The modular pedal unit was created with 3 wires from inside the telephone cable. Accordingly, the unit cable was "snaked" into the side of the sofa foot, through the hole of the middle of the foot, and the conductor wires were soldered to their appropriate pins: V_{cc}, ground, and Teensy signal, on the leads of the switch. Next, the aluminum panel (with panel mounted switch) was screwed into the top of the sofa foot. Finally, a tie wrap was placed around the unit cable, near the side opening of the foot, to secure the cable. The resulting sustain pedal is displayed in Figure 49.

7.2.4 Sensor Cable Design

The sensor units were redesigned with sturdier, more flexible cables that could withstand the wear and tear of the playing technique. It was realized during testing that the telephone cables being utilized for the modular sensor units had broken their connections from the male audio jacks. The disconnection was related to the copper conductors inside of the telephone cables being solid conductors, which were too weak for this application. Thus, it became apparent that the cables had to be replaced with cables that contained stranded conductors, being able to endure the flexibility entailed by this application. As a result, stranded conductor ribbon cables with 10 conductors were purchased. While being more optimal for flexibility and sturdiness, ribbon cables would also allow for less clutter during performance. Instead of having an individual cable for each component of the device, there would now be only one ribbon cable on each side of the user's body, since there are 10 conductors in each ribbon cable (enough to satisfy all connections to the audio jacks).

While the ribbon cables would solve the issue of the audio jack disconnections, it was vital to still keep each sensor modular. Therefore, a design was formulated to involve the ribbon cable as an interconnect between the gloves and the device. This interconnect was designed to include male audio jacks for each component on one end (to connect to the device) and female audio jacks for each component on the opposing end (to connect to the gloves). The interconnect was constructed by splicing the desired wires of the ribbon cables and soldering its conductors to their appropriate jacks. Once all connections were made, heat shrink was applied around all of the male and female audio jacks to secure and consolidate the jacks. The flex sensors on the gloves would

keep their connections to the original telephone cable, but the cable would be cut at the wrist region, to allow a male audio jack to be connected. The female audio jacks from the interconnect would then be connected to their respective sensor. As a result, each sensor is still modular, but the improved overall design utilizes studier cables and involves less clutter. The sensor cable design for both the left and right hand is shown in Figure 50 and 51, respectively:

Figure 50: Left-Hand Sensor Cable

Figure 51: Right-Hand Sensor Cable

The telephone cable of the sustain pedal was also exchanged with a more robust cable that contained stranded wire. This eliminated possible disconnections that might have occurred with the original weaker cable.

8. Results

The functionality of the final product was analyzed to determine how well the device performs in accordance to the specifications. Particularly, calibration and the general playing of the *SansTrumpet* were studied. Data pertaining to the calibration of both the trigger and embouchure were gathered and plotted. For each, three trials were conducted to also observe consistency. Data for the playing of the device was gathered as the chromatic scale was played with the device. Similarly, the data was plotted to graphically observe the functionality of the device. However, in this case, multiple plots were created in order to observe each component (valve, embouchure, and trigger) both separately and collectively.

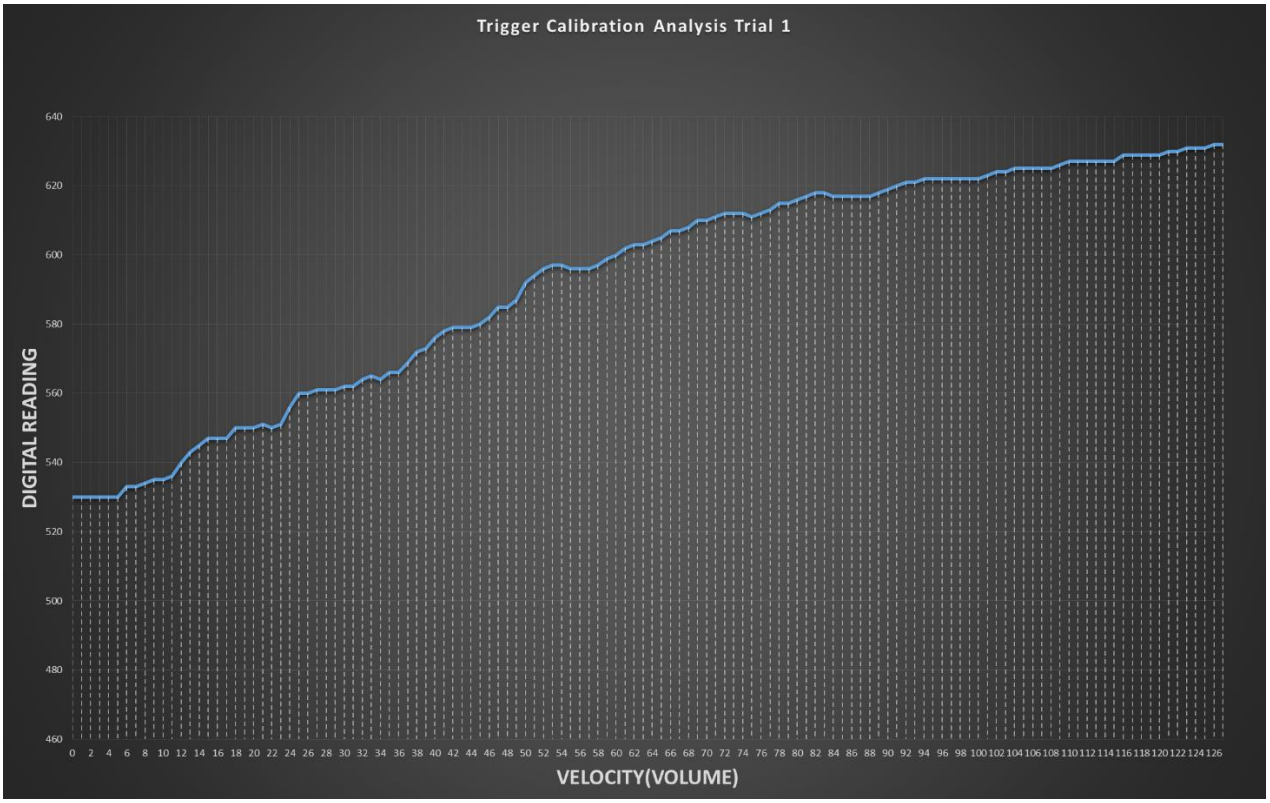

Figure 52: Trial One of Trigger Calibration Analysis

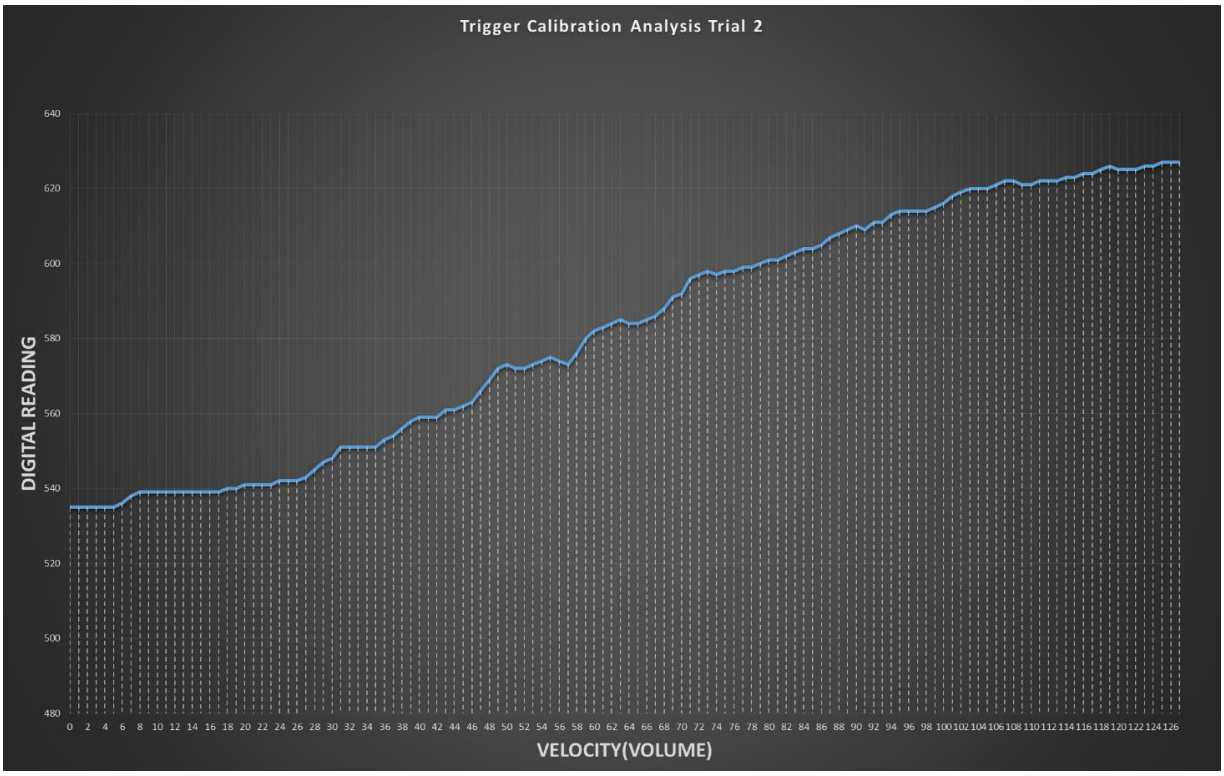

Figure 53: Trial Two of Trigger Calibration Analysis

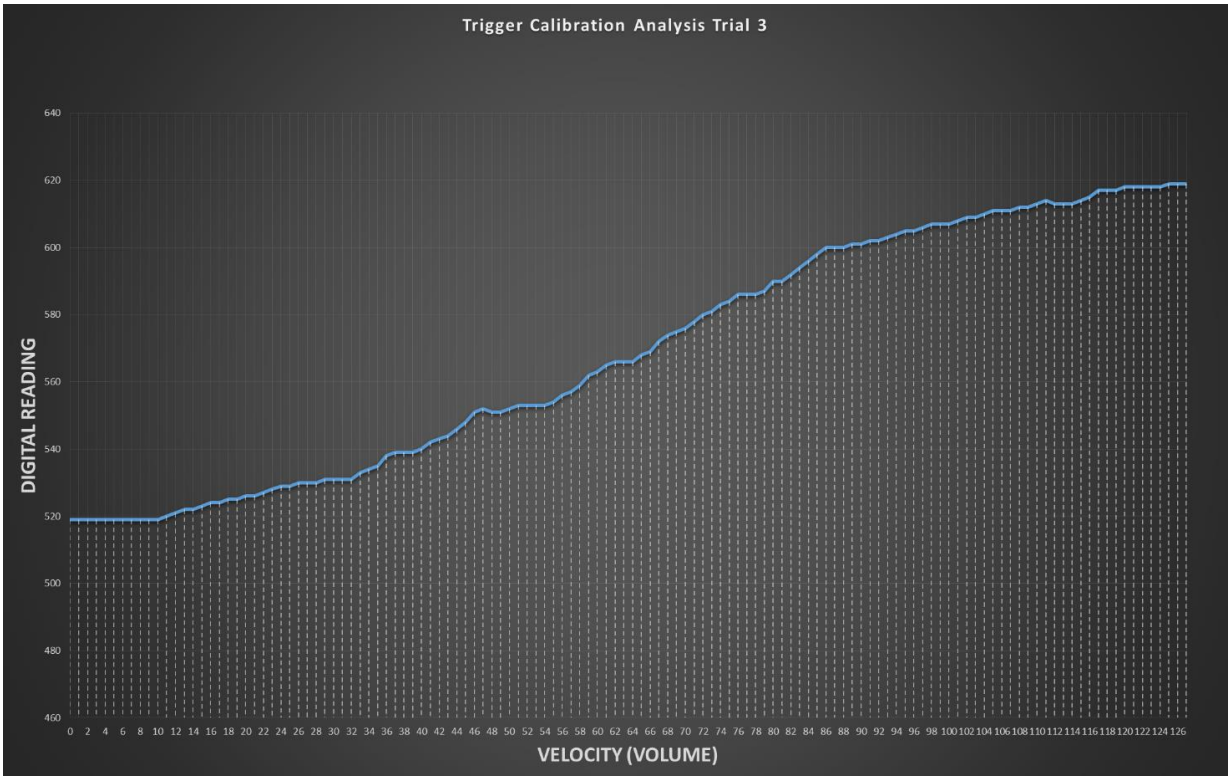

Figure 54: Trial Three of Trigger Calibration Analysis

Figures 52 to 54 depict the resulting plot of the gathered data of the trigger calibration. As shown, a total of three trials were conducted. For each trial, the sweep of the trigger finger (left index finger) was steadily bent within a time span of three seconds. For each of the plots, the velocity (MIDI volume) is shown on the x-axis while the digital value of the analog signal is shown on the y-axis. As indicated in the software section, the each index corresponds to a MIDI velocity number (0 to 126). The trigger calibration indicates that over all three trials, the general sweep method appears to produce expected results where the curve exhibits a generally increasing nature. As shown, the middle portion of the calibration curve over the three trials is linear. Notice the flat portions of the plots which are evident at the beginning and end of the calibration curve. The flat portion at the beginning of the curve relates to the slight delay in reaction of noticing the LED. The flat portion towards the end of the curve correspond to the fact that the finger (specifically knuckle region), has reached near maximum bend. This type of behavior is expected, and therefore is not a concern for the general calibration method.

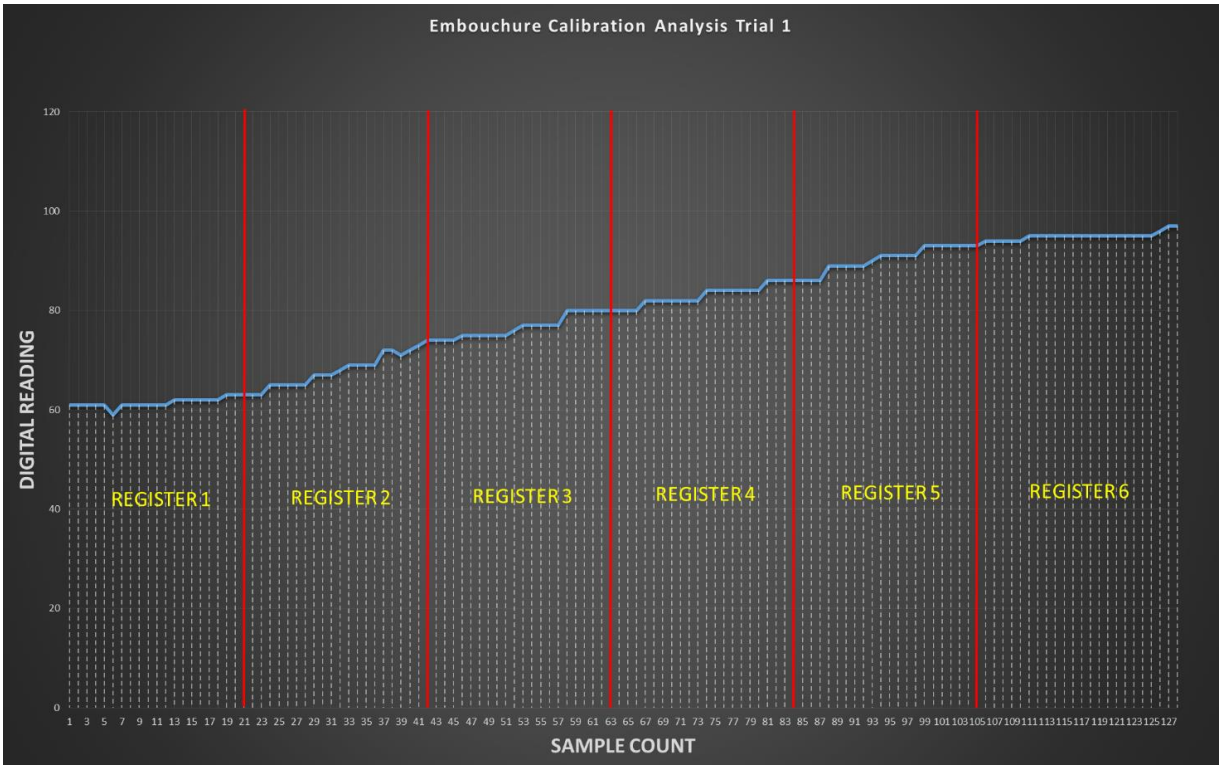

Figure 55: Trial One of Embouchure Analysis

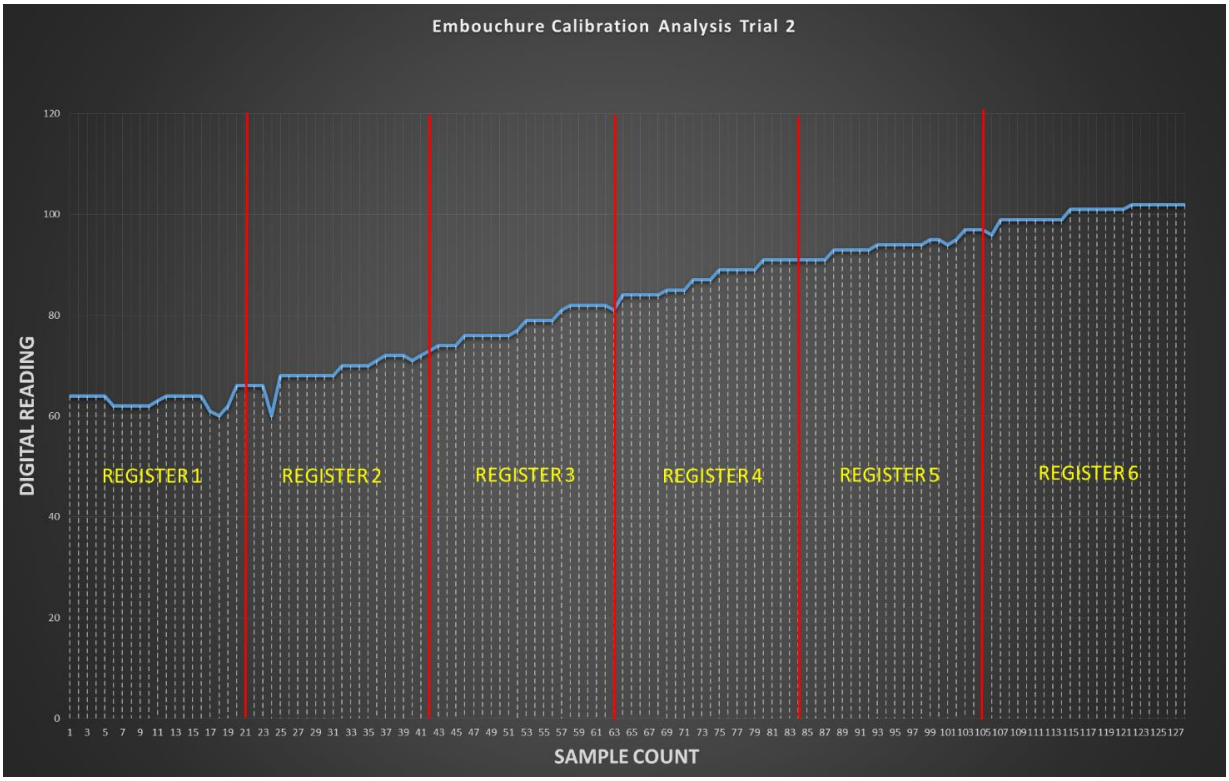

Figure 56: Trial Two of Embouchure Analysis

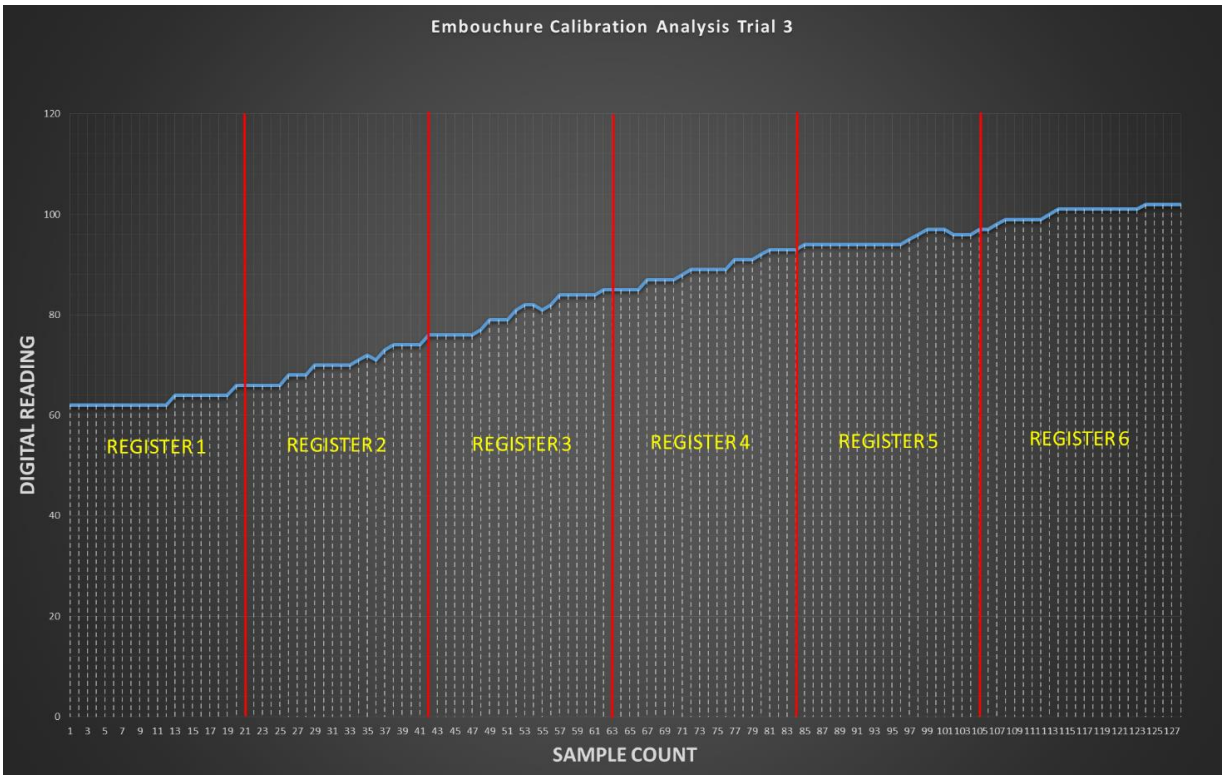

Figure 57: Trial Three of Embouchure Analysis

Figures 55 to 57 depict the calibration analysis plots pertaining to the embouchure. As shown in the plots, the calibration curve exhibits a steadily increasing characteristic in each of the three trials. As described previously, there are a total of six embouchure registers which were divided in segments of 21 indexes for each. Therefore as depicted in the plot, every 21 sample count (index values) of the total 127, correspond to a register. The results indicate that the embouchure functions well. However, notice that there are some fluctuations in trial 2 of embouchure calibration. Such fluctuations result from possible obstacles within the area such as a chair that is too close. This is due to the intrinsic design of the proximity sensor. Therefore, such calibration is optimal when conducted in a generally open area.

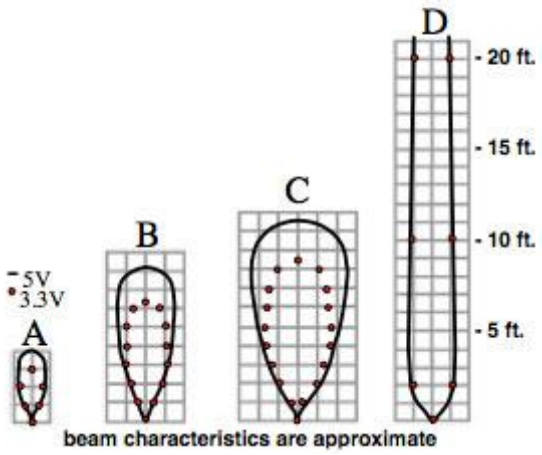

Figure 58: Proximity Sensor Beam Characteristics [80]

Figure 58 displays the beam characteristics of the *MaxBotix Ultrasonic Rangefinder LV-EZ1* proximity sensor. As seen in Figure 58, the form of the ultrasonic beam signal exhibits a different shape depending on both the voltage used as well as the distance the beam is reflecting from. For the *SansTrumpet* implementation, the proximity sensor is utilized from the waist region to possibly above the user's head. Therefore in according to the characteristics shown in Figure 58, the results would correspond to situations A and B. Notice that situation B exhibits a wider characteristic than situation A. This shows that the beam may be wide rather than narrow, and therefore may be more susceptible to interference by objects near the user. These characteristics imply the need to maintain an "open area" in order to avoid interference by nearby objects.

In summary, the results of the calibration analysis plots indicate that the calibration function operates as expected given that the user calibrates the embouchure (proximity sensor) in an open area within the time span and that for the trigger, the user is able to perform a sweep within the dedicated time span.

In order to test the general functionality of the *SansTrumpet* several notes were played in succession in approximately 3 seconds apart. The purpose of this test is to analyze each element of the *SansTrumpet* during its operation.

Table 18: Desired Notes for Test

Note	**Valve 1**	**Valve 2**	**Valve 3**	**Embouchure Register**	**MIDI Velocity**
C4	Open	Open	Open	1	Medium
C#4	Closed	Closed	Closed	2	Medium
D4	Closed	Open	Closed	2	Medium-High
D#4	Open	Closed	Closed	2	High
E4	Closed	Closed	Open	2	Medium

Table 18 shown above indicates the desired notes that were to be played during the test. As seen, a total of 5 notes were to be played. The table also indicates the requirements for the corresponding note in terms of valve state (open or closed) and the embouchure register. As indicated, "Closed" refers to pressing a valve while "Open: refers to un-pressed. For embouchure register, there are a total of 6. The trigger corresponds to the "triggering" of notes as well as selection of the MIDI velocity parameter (volume). For the purpose of this test, the MIDI velocity was intended to increase upon each note and then decrease at the final note.

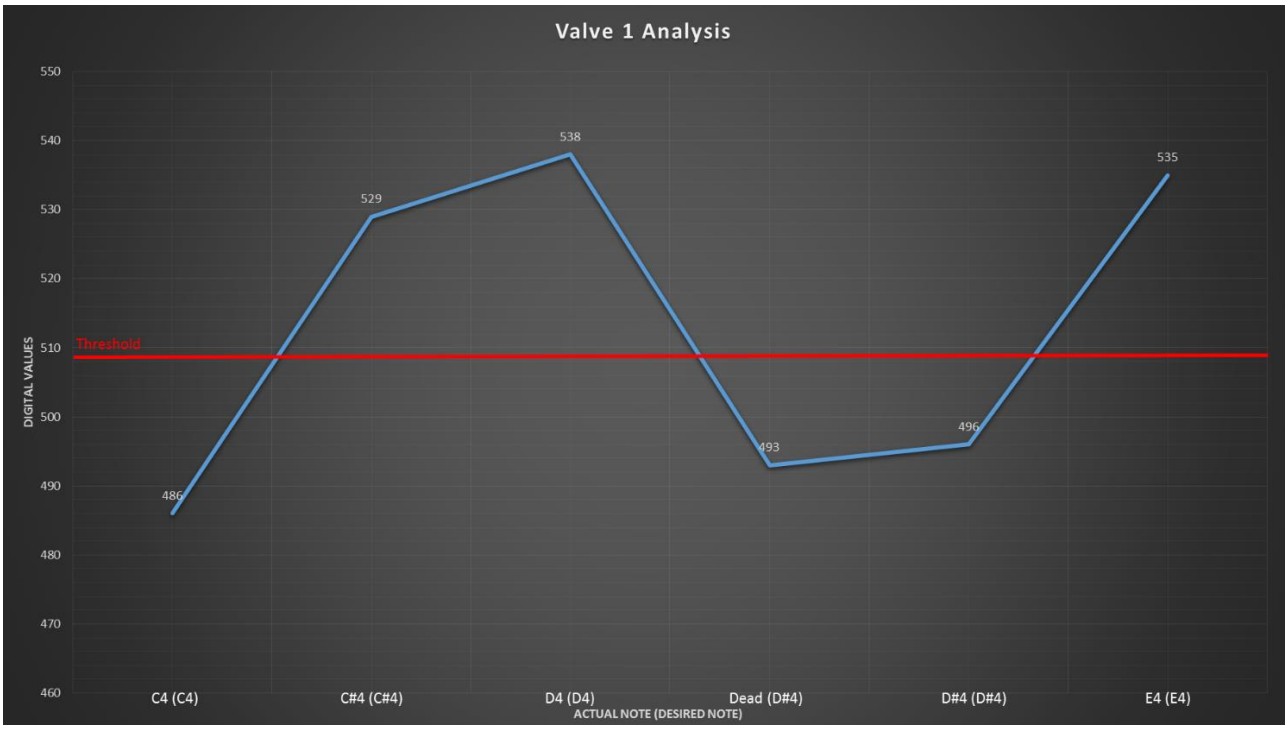

Figure 59: Valve 1 Analysis

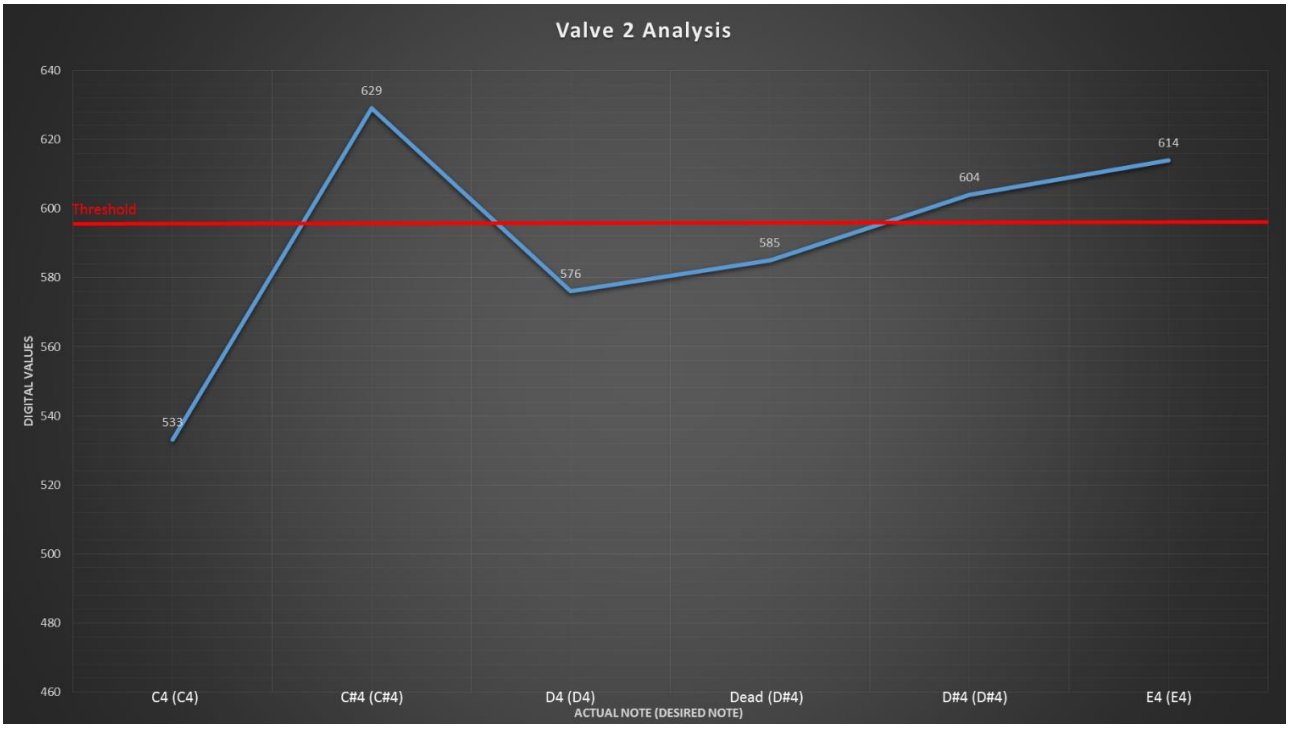

Figure 60: Valve 2 Analysis

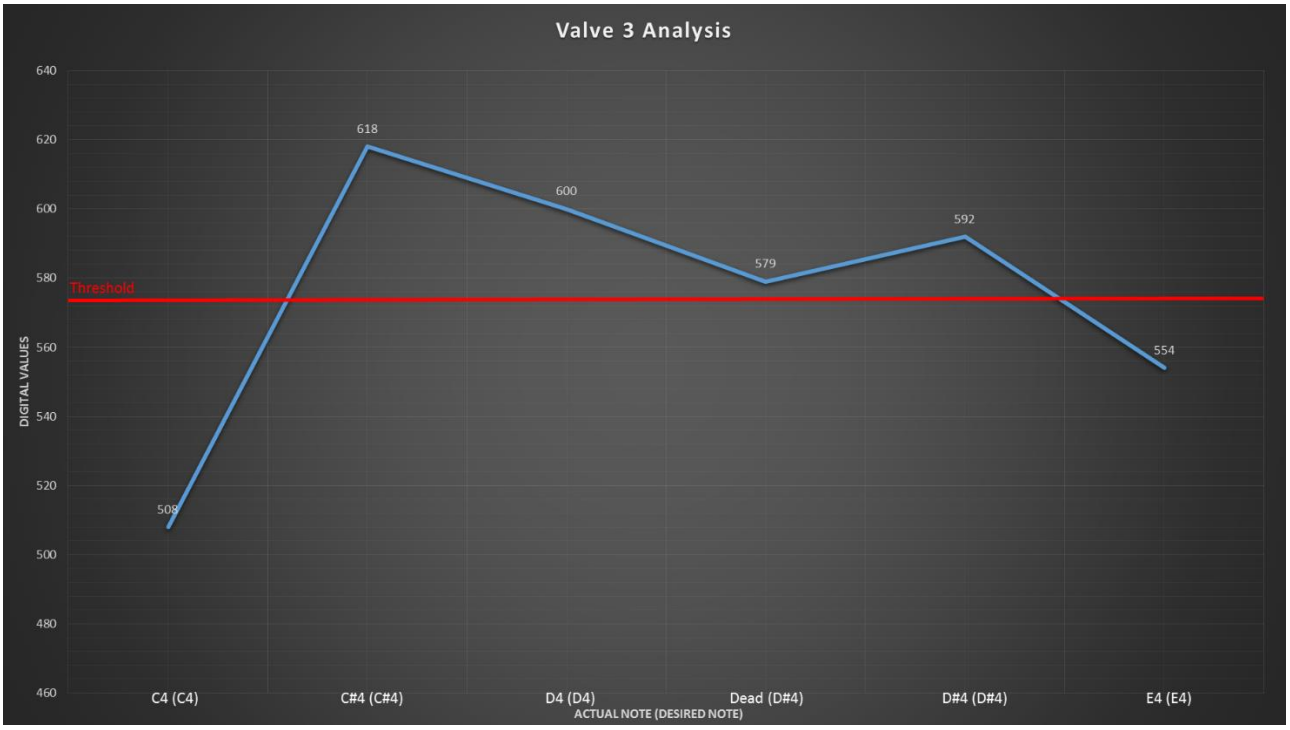

Figure 61: Valve 3 Analysis

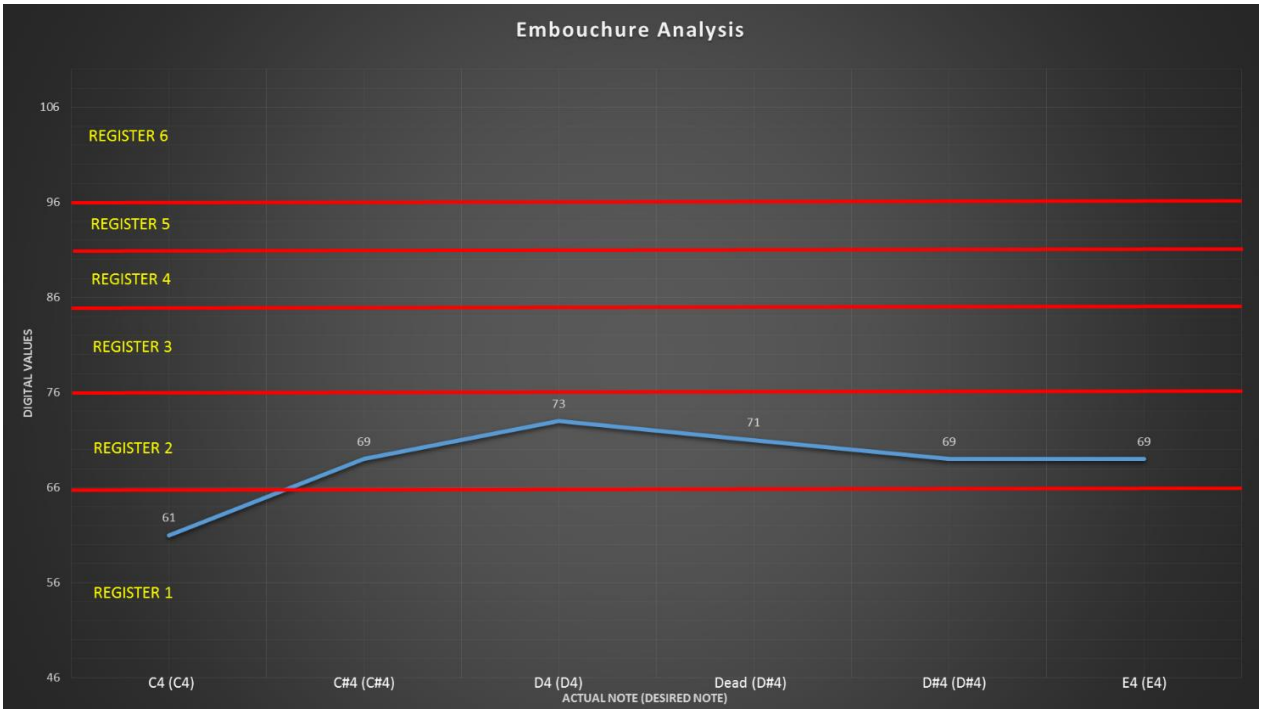

Figure 62: Embouchure Analysis

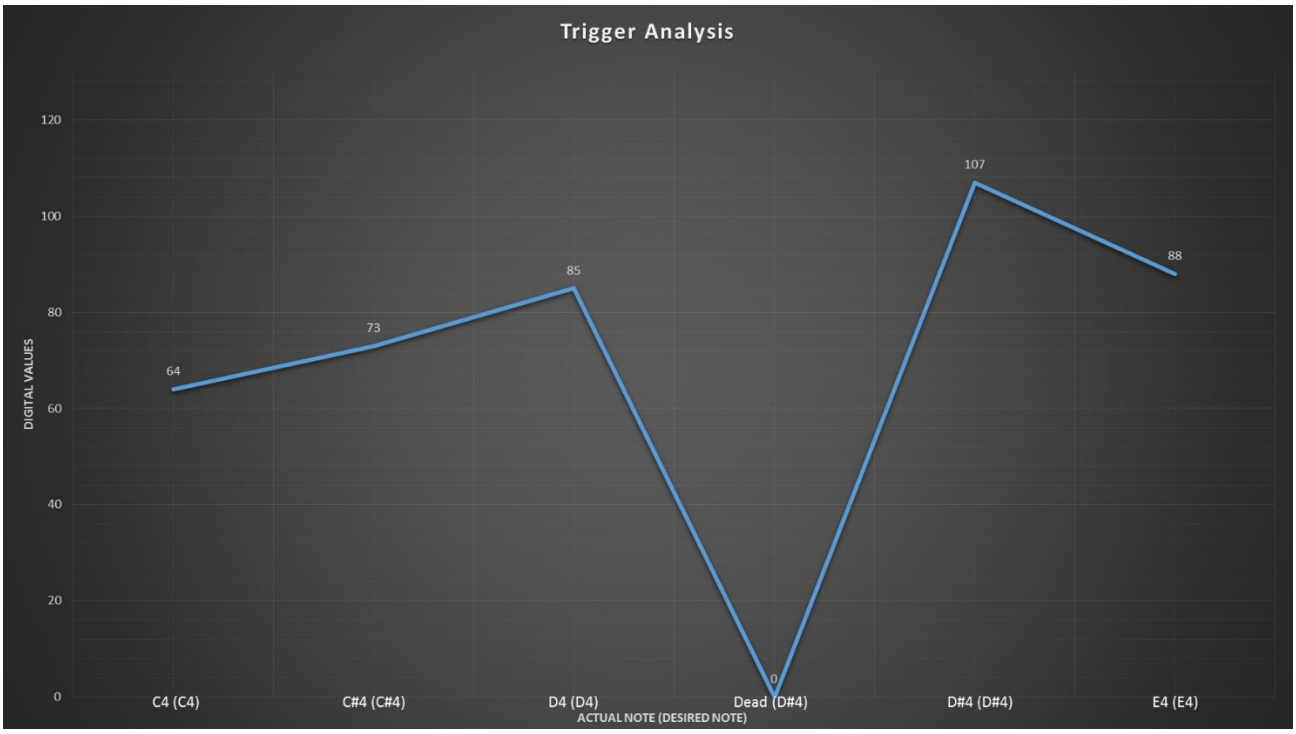

Figure 63: Trigger Analysis

Figures 59 to 63 depict the results of the calibration analysis for each particular element such as the valves, embouchure, and the trigger. As indicated in the plots, the x-axis pertains to the produced note while the desired note is enclosed in parentheses. The y-axis indicates the corresponding digital values. The resulting notes that were produced from the desired notes as shown in the plots, are summarized qualitatively below in Table 19:

Table 19: Actual Notes Produced from Test

Note	Valve 1	Valve 2	Valve 3	Embouchure Register	MIDI Velocity
C4	Open	Open	Open	1	Medium
C#4	Closed	Closed	Closed	2	Medium
D4	Closed	Open	Closed	2	Medium-High
Dead	Open	Open	Closed	2	None
D#4	Open	Closed	Closed	2	High
E4	Closed	Closed	Open	2	Medium

In comparison to the desired notes shown in Table 19, only one of the notes did not function as expected. Specifically, note the D#4 resulted in a “Dead” note as seen in the results. The dead

note is depicted as a 0 MIDI velocity and hence no sound, shown in the trigger analysis plot which is depicted in Figure 63. As indicated previously, the produced note depends on the valve combination as well as the embouchure, while the trigger determines the velocity of the note. In order to play the note D#4, qualitatively, valves 2 and 3 must be closed (pressed) while valve 1 must be left opened. At the same time, the proximity sensor should be positioned such that the digital value is within the region pertaining to register 2. Quantitatively, the valves must exceed their corresponding thresholds that were determined from the calibration. For this test, the calibrated thresholds for valves 1, 2, and 3 were 508, 596, and 574 respectively. For the embouchure, the digital values from the proximity sensor must be between 66 and 76 to be in the second register. As observed in the valve results shown in Figures 59 to 61, the valve combination when attempting to play the first D#4 resulted in a dead note. Notice that the digital value corresponding to valve 1 was 493. This is below the threshold and so, valve 1 is considered open or un-pressed. The value for valve 2 was 585, which was also below its corresponding threshold. However, one of the conditions to play the note D#4 is that valve 2 must be pressed, and therefore the value should exceed the threshold. Valve 3 as shown, resulted in a digital value of 579, which exceeded the threshold. Lastly, the digital value for the embouchure was a 71 which is indeed between the thresholds of 66 and 76. Therefore, the dead note resulted solely because valve 2 did not exceed the threshold to obtain the correct fingering combination. The second attempt to play note D#4 resulted in the correct note when the valve fingers were adjusted to the appropriate positioning. As shown in the analysis plots, valve 1 remained below the threshold while valves 2 and 3 exceeded their corresponding thresholds. The embouchure remained within the second register. In terms of the results pertaining to the trigger, the MIDI velocity was to be increased steadily and then decreased for the final note. As shown in Figure 63, the behavior for the MIDI velocity performed as expected when excluding the dead note.

The results of the test analysis, indicated that the device performed well. However as shown in the results, the first attempt in playing note D#4 resulted in a dead note. As indicated, the dead note was generated due to the fact that valve 2 failed to exceed the threshold value, and therefore an incorrect fingering combination was produced. This issue is a result of the physiology of the human hand such that each person has a different dexterity. When the user bends one finger, there is a tendency to also bend the adjacent finger. The fabric material of the gloves that were

implemented for the *SansTrumpet* also contributed to this problem. When one finger is bent, the fabric would also pull on the regions of the flex sensors. This problem would affect not only the playing of notes, but the calibration. Therefore, perhaps the threshold for valve 2 during the calibration was not as accurate as desired due to the material of the glove and the dexterity of the hand. Otherwise, the other notes were produced as expected as shown in the results. The trigger and embouchure also performed well in which the results met expectations and did not exhibit noticeable concerns during this test.

A video of the *SansTrumpet* being played was also recorded and submitted along with this document. The video exhibits the playing of the chromatic scale as well as the ability to utilize the MIDI capabilities to play different instruments. The results from testing the *SansTrumpet* indicate that the reliability can be improved upon. In particular, the playing of notes with certain fingerings, exhibit reliability issues. This was evident in the test analysis when valve 2 failed to exceed its threshold, resulting in a dead note. However, for the majority of the fingering combinations, the *SansTrumpet* performs reliably. Reliability issues may be due to the mechanical aspects of the *SansTrumpet* such as the flex sensors, human physiology, and the gloves. The flex sensors exhibit high tolerance in which the resistance values differ significantly from sensor to sensor. Although calibration addresses this issue, the calibration function itself is susceptible to issues posed by the glove material as well as the dexterity of the hand. As described, the bending of one's finger may cause the adjacent fingers to also bend. Similarly, the fabric material of the glove also contributes to this problem. However, the implemented playing technique appears to function well. As seen in the performance video demo, the user is able to visually convey their performance to some degree with this playing technique, while being able to utilize the capability of MIDI.

9. Future Considerations

Various aspects of the *SansTrumpet* can be improved upon, either through software or mechanical alterations. Due to the fact that the flex sensors have high tolerance, improved flex sensors could be implemented. In order to improve reliability, one possible method to address this is to reposition the flex sensors in such a way, that the valve finger movements are isolated from one another. A more complex triggering scheme can also be implemented to improve reliability. Perhaps different sensors could be implemented either to complement or replace the current sensors. For example, hall effect sensors may be a viable replacement for the flex sensors pertaining to the valves. Other considerations are to improve the *SansTrumpet* in terms of features. In example, musical aspects could be added such as pitch bend. Mechanically, a wireless interface between the gloves, central unit, as well as the computer could be implemented. This would replace the ribbon cable interconnections, allowing users to perform and move around without the hindrance of cables. Since the user would not be tethered by the USB cable, this would also increase the distance and area that the user could use. Among the various possible changes, the team believes that the general playing technique should remain the same. The current playing technique appears to be intuitive to users while at the same time allow users to provide the visual aspect. Therefore improvements should build upon and reinforce this playing technique.

10. Conclusion

The purpose of this project was to develop a hand gesture-based MIDI controller. The visual aspects were found to play a critical role in the audiences' perception of the performance. Through hand gestures, such aspects can be conveyed to the audience, while performing music. The team chose the trumpet to serve as a foundation to the *SansTrumpet* due to the fact that the playing technique posed to be intuitive to hand movement. The *SansTrumpet* capabilities include having MIDI functionality, monophonic and polyphonic modes, and the full key range of an acoustic trumpet. These elements expand the capabilities of an acoustic trumpet by allowing the user to play and record virtual instruments in the DAW of their choice.

The final results of the *SansTrumpet* indicated that the device functions well, as seen in the video demonstration of the C chromatic scale, however, the reliability can be improved upon. Mainly, the combination of the hand physiology and the glove's mechanical design has contributed to unreliability of the valves. Despite these issues, they are only evident in particular valve combinations. However, for the majority, the instrument performs satisfactory. Future directions to consider include improving upon the reliability of the *SansTrumpet*, as well as implementing additional features. Through software, a complex valve triggering scheme can be implemented to resolve the certain cases of unreliability. Mechanically, the positioning of the sensors can be relocated to better isolate the bend of one finger from another. Finally, additional MIDI features, such as pitch bend, can be incorporated to expand its capabilities as a MIDI controller.

11. References

[1] C. Tsay. Sight over sound in the judgment of music performance. *Proceedings of the National Academy of Sciences* 2013. . DOI: 10.1073/pnas.1221454110.

[2] Christian Geiger, Holger Reckter, David Paschke, and Florian Schulz. (2008, Mar.). "Poster: Evolution of a Theremin-Based 3D-Interface for Music Synthesis," *IEEE Symposium on 3D User Interfaces 2008*, [Online] Available: IEEE Xplore [Sept. 10, 2013]

[3] Andrei Smirnov. "Music and Gesture: Sensor Technologies in Interactive Music and the Theremin based Space Control Systems," [Online] Available: http://www.theremin.ru/asmir/articles/ICMC2000_AMotion.pdf [Sept. 10, 2013]

[4] Theremin Society (2008). [Online] Available: http://thereminsociety.com/Leon_Theremin_Playing_Theremin.jpg Theremin Society

[5] Louis E. Garner, Jr. (1967). "For that different sound, Music a la Theremin," *Popular Electronics*, Available: http://www.studio250.fr/docs/divers%20synthese/theremin2.pdf [Sept. 10, 2013]

[6] Kenneth D. Skeldon, Lindsay M. Reid, Viviene McInally, Brendan Dougan, and Craig Fulton. (1998). "Physics of the Theremin," *American Journal of Physics*, [Online], vol. 66 (11), pp. 945 Available: AAPT [Sept. 10, 2013]

[7] Kenneth D. Skeldon, Lindsay M. Reid, Viviene McInally, Brendan Dougan, and Craig Fulton. (1998). "Figure 14" in "Physics of the Theremin," *American Journal of Physics*, [Online], vol. 66 (11), pp. 945 Available: AAPT [Sept. 10, 2013]

[8] STEIM. "3971518724_97a62caeb8.jpg," *in Remembering MW* [Online]. Available: http://steim.org/2009/10/remembering-mw/?utm_source=rss&utm_medium=rss&utm_campaign=remembering-mw [Sept. 9, 2013]

[9] Michel Waisvisz. "The Hands." [Online]. Available: http://www.crackle.org/TheHands.htm

[10] Crackle.org. "1984 - 1989 The Hands (first version)." [Online]. Available: http://www.crackle.org/The%20Hands%201984.htm

[11] A. J. Bongers. Tactual display of sound properties in electronic musical instruments. *Displays 18(3)*, pp. 129-133. 1998. . DOI: http://dx.doi.org/10.1016/S0141-9382(98)00013-4.

[12] Laetitia Sonami. "27_lady-1.jpg," in *The Lady's Glove, a brief history* [Online]. Available: http://www.sonami.net/works/ladys-glove/ [Sept. 8, 2013]

[13] Laetitia Sonami. "The Lady's Glove, a Brief History." [Online]. Available: http://www.sonami.net/works/ladys-glove/ [Sept. 8, 2013]

[14] Laetitia Sonami. ”27_lady-4d.jpg,” in *The Lady's Glove, a brief history* [Online]. Available: http://www.sonami.net/works/ladys-glove/ [Sept. 8, 2013]

[15] Laetitia Sonami. ”27_lady-5c.jpg,” in *The Lady's Glove, a brief history* [Online]. Available: http://www.sonami.net/works/ladys-glove/ [Sept. 8, 2013]

[16] Bean. (1998). “Electronic Musician.” *Tech: A Soft Touch* [Online]. Available: http://www.sonami.net/LS-reviews/rev_EM6_98.htm [Sept. 8, 2013]

[17] Elena Jessop. “vamp_glove2.jpg,” in *The Vocal Augmentation and Manipulation Prosthesis (VAMP)* [Online]. Available: http://web.media.mit.edu/~ejessop/vamp.html

[18] E. Jessop. The vocal augmentation and manipulation prosthesis (vamp): A conducting-based gestural controller for vocal performance. Demonstration, NIME Pittsburgh 2009.

[19] Music Gloves Ltd. (2012). “in Brief.” [Online]. Available: http://imogenheap.com/thegloves/ [Sept. 14, 2013]

[20] WiredVideoUK. Poster, “Imogen Heap Performance with Musical Gloves Demo: Full Wired Talk 2012” [Online]. Available: http://www.youtube.com/watch?v=6btFObRRD9k

[21] Music Gloves Ltd. (2012). “closeup4.jpeg,” in *the Story* [Online]. Available: http://imogenheap.com/thegloves/

[22] Arabella Seer. (Dec. 28, 2012). “heap-5-520x292.jpg,” in *Imogen Heap’s ‘Gloves’: When motion becomes music* [Online]. Available: http://www.humansinvent.com/?_escaped_fragment_=/10261/imogen-heaps-gloves-when-motion-becomes-music/#!/10261/imogen-heaps-gloves-when-motion-becomes-music/

[23] Arabella Seer. (Dec. 28, 2012). “Imogen Heap’s ‘Gloves’: When motion becomes music.” [Online]. Available: http://www.humansinvent.com/?_escaped_fragment_=/10261/imogen-heaps-gloves-when-motion-becomes-music/#!/10261/imogen-heaps-gloves-when-motion-becomes-music/

[24] 5DT. (2005). “5DT Data Glove 14 Ultra.” [Online]. Available: http://www.5dt.com/products/pdataglove14.html

[25] Tom Cheshire. (Sept. 29, 2011). “How to make music with gestures.” [Online]. Available: http://www.wired.co.uk/magazine/archive/2011/10/how-to/how-to-make-music-with-gestures

[26] WiredUK. (Oct. 29, 2012). “Kinect, ‘Magical Gloves’ Let Imogen Heap Turn Movement Into Music.” [Online]. Available: http://www.wired.com/underwire/2012/10/imogen-heap-magical-gloves/

[27] Leap Motion. “what2_desktop-2d07bc0b0592c4d0ea7e845f18f2c81f.jpg.” [Online]. Available: https://www.leapmotion.com/product [Sept. 18, 2013]

[28] Leap Motion. “Airspace Store.” [Online]. Available: https://airspace.leapmotion.com/ [Sept. 17, 2013]

[29] Michael Gorman. (July 22, 2013). “Leap Motion Controller Review.” [Online]. Available: http://www.engadget.com/2013/07/22/leap-motion-controller-review/

[30] Fingertapps. “Fingertapps Piano.” [Online]. Available: https://airspace.leapmotion.com/apps/fingertapps-piano/windows [Sept. 17, 2013]

[31] Olympia Noise Co. “Chordion Conductor.” [Online]. Available: https://airspace.leapmotion.com/apps/chordion-conductor/osx [Sept. 17, 2013]

[32] Dave Roos. “How Stuff Works: How MIDI Works” [Online]. Available: http://entertainment.howstuffworks.com/midi.htm [Sep. 24, 2013]

[33] Juan P. Bello. “MIDI Code” [Online]. Available: http://www.nyu.edu/classes/bello/FMT_files/9_MIDI_code.pdf [Sep. 24, 2013]

[34] Jeffrey Hass. (2010). “How does the MIDI system work?” [Online]. Available: http://www.indiana.edu/~emusic/etext/MIDI/chapter3_MIDI4.shtml [Sep. 24, 2013]

[35] MIDI Note Numbers for Different Octaves. “MIDI Mountain” [Online]. Available: http://www.midimountain.com/midi/midi_note_numbers.html [Sep. 25, 2013]

[36] Juan P. Bello. “Chromatic Western Music Scale,” in *MIDI Code* [Online]. Available: http://www.nyu.edu/classes/bello/FMT_files/9_MIDI_code.pdf [Sep. 24, 2013]

[37] cplai. Poster, “James Morrison’s Digital Trumpet” [Online]. Available: http://www.youtube.com/watch?v=BxLlym502bI

[38] Morrison Digital Instrument. “MDT-angle.jpg,” in *INVENTERS BLOW THEIR OWN TRUMPET* [Online]. Available: http://www.digitaltrumpet.com.au/press.htm [Sept. 18, 2013]

[39] pabslondon. Poster, “Yamaha EZ-TP digital trumpet. Interview with Ted Furman” [Online]. Available: http://www.youtube.com/watch?v=mAFm-k2468g

[40] Japan Trend Shop. “yamaha-ez-tp-eztp-trumpet.jpg,” in *EZ TP Electronic Trumpet by Yamaha* [Online]. Available: http://www.japantrendshop.com/ez-tp-electronic-trumpet-by-yamaha-p-181.html [Sept. 18, 2013]

[41] Thomas Craig, Bradley Factor. “notefreq.png,” in *Music Theory* [Online]. Available: https://instruct1.cit.cornell.edu/courses/ee476/FinalProjects/s2008/twc22_bef23/twc22_bef23/ [Oct. 12, 2013]

[42] Thomas Craig, Bradley Factor. “Cornell University School of Electrical and Computer Engineering: Trumpet MIDI Controller” [Online]. Available: https://instruct1.cit.cornell.edu/courses/ee476/FinalProjects/s2008/twc22_bef23/twc22_bef23/#design [Oct. 12, 2013]

[43] Phil Burk, Larry Polansky, Douglas Repetto, Mary Roberts, Dan Rockmore. (2011, Spring). “Music and Computers: A Theoretical and Historical Approach” [Online]. Available: http://music.columbia.edu/cmc/MusicAndComputers/chapter3/03_01.php [Oct. 5, 2013]

[44] Jeff Bordogna. (2013). “Amplitude Envelope,” [Online]. Available: http://www.jeffbordogna.com/wp-content/uploads/2013/04/amplitude-envelope.png [Oct. 5, 2013]

[45] SparkFun. “10264-01.jpg,” in *Flex Sensor 2.2"* [Online]. Available: https://www.sparkfun.com/products/10264 [Sept. 28, 2013]

[46] SpectraSymbol. “Flex Sensor” [Online]. Available: https://www.sparkfun.com/datasheets/Sensors/Flex/flex22.pdf [Sept. 28, 2013]

[47] Spoocter. (Oct. 15, 2012). “Arduino based MIDI controller” in *Arduino Forums* [Online]. Available: http://forum.arduino.cc/index.php/topic,127390.0.html [Oct. 6, 2013]

[48] PJRC. “Using USB MIDI” [Online]. Available: http://www.pjrc.com/teensy/td_midi.html [Oct. 9, 2013]

[49] Eric Ma. (Feb. 28, 2011). “arduino_uno_test.jpg,” in *Integrating Technology in Science Projects - the Arduino Uno* [Online]. Available: http://if.youthscience.ca/blog/integrating-technology-science-projects-arduino-uno [Sept. 28, 2013]

[50] Arduino. “Introduction.” [Online]. Available: http://arduino.cc/en/Guide/Introduction [Sept. 28, 2013]

[51] Arduino. “Download the Arduino Software.” [Online]. Available: http://arduino.cc/en/main/software [Sept. 28, 2013]

[52] Arduino. “Arduino Uno.” [Online]. Available: http://arduino.cc/en/Main/arduinoBoardUno [Sept. 28, 2013]

[53] Jimmie Rodgers. (Sept. 22, 2011). “arduino-leonardo-600x450.jpg,” in *Arduino Leonardo Opens Doors to Product Development*. [Online]. Available: http://makezine.com/2011/09/22/arduino-leonardo-opens-doors-to-product-development/ [Sept. 28, 2013]

[54] Arduino. “Arduino Leonardo.” [Online]. Available: http://arduino.cc/en/Main/arduinoBoardLeonardo [Sept. 28, 2013]

[55] E. Spitler. (2013, Sept. 23). “RE MIDI Shield Compatibility.” Personal e-mail.

[56] Paul Maunders. “DSC_1085.jpg” in *Raspberry Pi Measurements & Dimensions* [Online]. Available: http://www.pyrosoft.co.uk/blog/2012/01/15/raspberry-pi-measurements-dimensions/

[57] Paul Maunders. “DSC_1095.jpg” in *Raspberry Pi Measurements & Dimensions* [Online]. Available: http://www.pyrosoft.co.uk/blog/2012/01/15/raspberry-pi-measurements-dimensions/

[58] RasperryPi. “Faqs.” [Online]. Available: http://www.raspberrypi.org/faqs [Sept. 28, 2013]

[59] Mikey Sklar. “Overview” [Online]. Available: http://learn.adafruit.com/reading-a-analog-in-and-controlling-audio-volume-with-the-raspberry-pi/overview [Sept. 28, 2013]

[60] PJRC. “tpp_main.jpg,” in *Teensy++ USB Development Board* [Online]. Available: http://www.pjrc.com/store/teensypp.html [Sept. 28, 2013]

[61] PJRC. Teensy++ USB Development Board [Online]. Available: http://www.pjrc.com/store/teensypp.html [Sept. 28, 2013]

[62] PJRC. Teensy USB Development Board [Online]. Available: http://www.pjrc.com/teensy/index.html [Sept. 28, 2013]

[63] SparkFun. “Teensy 3.0” [Online]. Available: https://www.sparkfun.com/products/11780 [Sept. 29, 2013]

[64] John Arne Birkeland. “Teensy3.jpg” in *Teensy 3.0* [Online]. Available: http://diydrones.com/profiles/blogs/teensy-3-0 [Sept. 28, 2013]

[65] PJRC. “Using USB MIDI” [Online]. Available: http://www.pjrc.com/teensy/td_midi.html [Oct. 14, 2013]

[66] S. Kolokowsky, T. Davis. (June 2006). “Common USB Development Mistakes – You Don’t Have To Make Them All Yourself!” [Online]. Available: http://www.cypress.com/?docID=9184 [Oct. 5, 2013]

[67] MicroChip. (2007). “Low Quiescent Current LDO” in *MCP1700*. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/21826b.pdf [Oct. 6, 2013]

[68] MicroChip. (2010). “250 mA Low Quiescent Current LDO Regulator” in *MCP1702*. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/22008E.pdf [Oct. 6, 2013]

[69] Diodes Incorporated. (2010). “1.5A Low Dropout Positive Adjustable or Fixed-Mode Regulator” in *AP1086* [Online]. Available: http://www.diodes.com/datasheets/AP1086.pdf [Oct .6, 2013]

[70] Atmel. (Oct. 2009). “8-bit AVR Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash” [Online]. Available: http://www.atmel.com/Images/doc8161.pdf [Oct. 6, 2013]

[71] MicroChip. (2007). 2.7V to 6.0V Single Supply CMOS Op Amps in *MCP601/1R/2/3/4* [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/21314G.pdf [Oct. 6, 2013]

[72] MicroChip. (2009). 2.5V to 6.0V Micropower CMOS Op Amp in *MCP606/7/8/9* [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/11177f.pdf [Oct. 6, 2013]

[73] SparkFun. “Triple Axis Magnetometer Breakout” [Online]. Available: https://www.sparkfun.com/products/10619 [Mar. 8, 2014]

[74] AdaFruit Industries. “MaxSonar-EZ1_LRG.gif” in *Maxbotix Ultrasonic Rangefinder – LV-EZ1* [Online]. Available: http://www.adafruit.com/products/172 [Mar. 8, 2014]

[75] MaxBotix. (2012). “LV-MaxSonar-EZ1 High Performance Sonar Range Finder” [Online]. Available: http://www.maxbotix.com/documents/MB1010_Datasheet.pdf [Mar. 8, 2014]

[76] PJRC. “Lowering Power & Extending Battery Life” [Online]. Available: http://www.pjrc.com/teensy/low_power.html [Oct. 8, 2013]

[77] P. Stoffregen. “Teensy 2.0 ++ or 3.0” in *PJRC Forums* [Online]. Available: http://forum.pjrc.com/threads/18939-Teensy-2-0-or-3-0 [Oct. 8, 2013]

[78] FreeScale Semiconductor. (May 2012). “Data Sheet: Technical Data” in *K20P64M50SF0* [Online]. Available: http://www.pjrc.com/teensy/K20P64M50SF0.pdf [Oct. 8, 2013]

[79] Instructables. “Radioshack Illuminated Switch Hack” [Online]. Available: http://www.instructables.com/id/Radioshack-Illuminated-Switch-Hack/?ALLSTEPS [Mar. 8, 2014]

[80] AdaFruit Industries. “ez1-beam-characteristics.jpg” in *Maxbotix Ultrasonic Rangefinder – LV-EZ1* [Online]. Available: http://www.adafruit.com/products/172 [Apr. 23, 2014]

12. Appendix

12.1 Appendix A: Basic Serial Print Code

```
void setup()
{
  Serial.begin(38400);
  analogReadRes(10);
}

int val;

void loop()
{
  val = analogRead(0);
  Serial.print("analog 0 is: ");
  Serial.println(val);
  delay(250);
}
```

12.2 Appendix B: Initial Flex Sensor Testing Code

```
void setup()
{
  pinMode (0, INPUT);
  analogReadResolution(10);
}

int val;

void loop()
{
  val = analogRead(0);

  if (val > 550)
  {
     usbMIDI.sendNoteOn(36, 90, 1);
  }
  else
  {
     usbMIDI.sendNoteOff(36, 90, 1);
  }

  delay(1800);
}
```

12.3 Appendix C: Version 1 of *SansTrumpet* Algorithm

```
void setup()
{
  pinMode (0, INPUT);
  analogReadResolution(10);
}

int val;
int i;
int flag = 0;
int sim_amplitude = 1;

void loop()
{
  val = analogRead(0);
```

```
  //Serial.print("analog 0 is: ");
  //Serial.println(val);

  if (val > 550)
  {
    if (flag == 0)
    {
      if (sim_amplitude == 1)
      {
        for (i=40; i < 90; i++)  // attack
           usbMIDI.sendNoteOn(24, i, 1);
        for (i=90; i > 60; i--)  // decay
           usbMIDI.sendNoteOn(24, i, 1);
        for (i=1; i < 20; i++)   // sustain
           usbMIDI.sendNoteOn(24, 60, 1);
      }
      else
      {
        usbMIDI.sendNoteOn(24, 90, 1);
      }
      flag = 1;
    }
  }
  else
  {
    if (flag == 1)
    {
      // release
      usbMIDI.sendNoteOff(24, 90, 1);
      flag = 0;
    }
  }

  //delay(1800);
}
```

12.4 Appendix D: *SansTrumpet* Algorithm (v2.0)

```
// PURPOSE:
// To test the Trigger sensor and Amplitude Envolope algorithms.  The Trigger sounds the pitch at first
// detection of the Trigger sweep transitioning back to the reset position.  The Amplitude Envelope is
// determined as follows:
//    1.  The Trigger sweep determines the Velocity of the pitch (Attack Peak).
//    2.  The speed (via a simple timecounter) at which the pitch is reached determines the Sustain.
//    3.  The Sustain is represented by playing the pitch followed by a delay() sustaining the pitch.
//    4.  Once the delay expires a MIDI NoteOff is issued representing the Release.
//
// DATE:     11/10/13
//
// AUTHOR:    Anthony Mastromattei and Thomas Nguyen
//

#define CYCLE_RESOLUTION        100
#define SUSTAIN_FACTOR          80

#define TRIGGER_RESET_POSITION   500
#define TRIGGER_MAX_POSITION     627

#define STATE_TRIGGER_WAIT       0
#define STATE_TRIGGER_RESET      1
#define STATE_TRIGGER_ADVANCING  2

int pin0_curr   = 0;
int pin0_prev   = 0;
int timeCounter = 0;
int state       = STATE_TRIGGER_WAIT;

void genAmplitudeEnvelope(byte pitch, byte velocity);
```

```
void setup()
{
  pinMode(0, INPUT);
  analogReadResolution(10);
}


void loop()
{
  switch (state)
  {
    case STATE_TRIGGER_WAIT:
      pin0_curr = analogRead(0);

      // Serial.print("STATE_TRIGGER_WAIT:  pin0 = ");
      // Serial.println(pin0_curr);

      if (pin0_curr <= TRIGGER_RESET_POSITION)
      {
        state = STATE_TRIGGER_RESET;
      }

      pin0_prev = pin0_curr;

      delay(CYCLE_RESOLUTION);
      break;

    case STATE_TRIGGER_RESET:
      pin0_curr   = analogRead(0);
      timeCounter = 0;

      // Serial.print("STATE_TRIGGER_RESET:  pin0 = ");
      // Serial.println(pin0_curr);

      if ((pin0_curr > pin0_prev) && (pin0_curr > TRIGGER_RESET_POSITION))
      {
        state = STATE_TRIGGER_ADVANCING;
      }

      pin0_prev = pin0_curr;

      delay(CYCLE_RESOLUTION);
      break;

    case STATE_TRIGGER_ADVANCING:
      pin0_curr = analogRead(0);

      // Serial.print("STATE_TRIGGER_ADVANCING:  pin0 = ");
      // Serial.println(pin0_curr);

      if (pin0_curr >= pin0_prev)
      {
        timeCounter++;
      }
      else  // first retracting detection
      {
        genAmplitudeEnvelope(62, (pin0_curr - TRIGGER_RESET_POSITION));
        state = STATE_TRIGGER_WAIT;
      }

      pin0_prev = pin0_curr;

      delay(CYCLE_RESOLUTION);
      break;
  }
}


void genAmplitudeEnvelope(byte pitch, byte velocity)
```

```
{
 if (velocity >  127)
 {
  velocity = 127;
 }

 Serial.print("===== GENERATE AMPLITUDE ENVELOPE / pitch = ");
 Serial.print(pitch);
 Serial.print(" / velocity = ");
 Serial.print(velocity);
 Serial.print(" / timeCounter = ");
 Serial.print(timeCounter);
 Serial.println(" =====");

 usbMIDI.sendNoteOn(pitch, velocity, 1);
 delay(timeCounter * SUSTAIN_FACTOR);
 usbMIDI.sendNoteOff(pitch, velocity, 1);
}
```

12.5 Appendix E: *SansTrumpet* Algorithm (v3.0)

```
// PURPOSE:
// SansTrumpet implementation, including representations of:
//              - Valve functionality
//              - Trigger functionality
//              - Embouchure register and
//              - Amplitude envelope
//
// VERSION:             3.0
// DATE:   12/8/13
//
// AUTHOR:              Anthony Mastromattei and Thomas Nguyen


// Constants:
//
// Execution
#define EXECUTION_MODE_DEBUG                    // Expose debug code (x-out to undefine)
#define xEXECUTION_MODE_MIDI                    // Expose MIDI code (x-out to undefine)

#ifdef EXECUTION_MODE_DEBUG
#define CYCLE_RESOLUTION        300             // Cycle delay (ms) to improve sensor predictability
#else
#define CYCLE_RESOLUTION        100             // Cycle delay (ms) to improve sensor predictability
#endif

// Trigger
#define TRIGGER_THRESHOLD       540             // Trigger Reset threshold sweep from on to off and vice-versa
#define TRIGGER_POSITION_MAX    (TRIGGER_THRESHOLD + 127)  // Trigger maximum position range
#define TRIGGER_POSITION_RESET(s) ((s) < (TRIGGER_THRESHOLD))          // Trigger reset condition
#define TRIGGER_REVERSE_DIR(c,p)  (((p) - (c)) > 10)                   // Trigger fluctuation large enough for reverse direction



// Valves
#define VALVE_1_THRESHOLD       520                     // Valve one threshold from on to off and vice-versa
#define VALVE_2_THRESHOLD       600                     // Valve two threshold from on to off and vice-versa
#define VALVE_3_THRESHOLD       600                     // Valve three threshold from on to off and vice-versa

#define VALVE_1_POSITION_OFF(s)   ((s) <  (VALVE_1_THRESHOLD)) // Valve 1 position off condition
#define VALVE_1_POSITION_ON(s)    ((s) >= (VALVE_1_THRESHOLD))  // Valve 1 position on condition
#define VALVE_2_POSITION_OFF(s)   ((s) <  (VALVE_2_THRESHOLD)) // Valve 2 position off condition
#define VALVE_2_POSITION_ON(s)    ((s) >= (VALVE_2_THRESHOLD))  // Valve 2 position on condition
#define VALVE_3_POSITION_OFF(s)   ((s) <  (VALVE_3_THRESHOLD)) // Valve 3 position off condition
#define VALVE_3_POSITION_ON(s)    ((s) >= (VALVE_3_THRESHOLD))  // Valve 3 position on condition

#define VALVE_1_BIT             4                       // Valve 1 bit position
#define VALVE_2_BIT             2                       // Valve 2 bit position
```

```
#define VALVE_3_BIT             1                                       // Valve 4 bit position

// Embouchure
#define EMBOUCHURE_NUM_ENTRIES    8                     // Maximum number of Embouchure entries
#define EMBOUCHURE_NUM_REG        6                     // Number of Embouchure Registers
#define EMBOUCHURE_MIN_SWEEP      500                   // Minimum embouchure sweep range
#define EMBOUCHURE_MAX_SWEEP      650                   // Maximum embouchure sweep range

#define EMBOUCHURE_REG_1          1                     // Embouchure Register range 1
#define EMBOUCHURE_REG_2          2                     // Embouchure Register range 2
#define EMBOUCHURE_REG_3          3                     // Embouchure Register range 3
#define EMBOUCHURE_REG_4          4                     // Embouchure Register range 4
#define EMBOUCHURE_REG_5          5                     // Embouchure Register range 5
#define EMBOUCHURE_REG_6          6                     // Embouchure Register range 6

// States
#define TRIGGER_STATE_INITIAL     1             // Initial state
#define TRIGGER_STATE_RESET       2             // Reset state (all pitch off)
#define TRIGGER_STATE_FORWARD     3             // State of trigger in forward motion
#define TRIGGER_STATE_REVERSE     4             // State of trigger in reverse motion
#define NEXT_STATE(s)             state = (s)   // Next state transition

// Analog pins
#define TRIGGER_PIN             8                       // Trigger pin value
#define VALVE_1_PIN             2                       // Valve 1 pin value
#define VALVE_2_PIN             1                       // Valve 2 pin value
#define VALVE_3_PIN             0                       // Valve 3 pin value
#define EMBOUCHURE_PIN          9             // Embouchure pin value

// MIDI
#define PRIMARY_CHANNEL           1           // Primary MIDI channel
#define CONTROL_NOTE_OFF_CMD      123         // Control change all-notes-off command
#define CONTROL_NOTE_OFF_VALUE    0           // Control change all-notes-off value
#define DEAD_PITCH              255           // Unplayable pitch


// Datatypes:
//
typedef struct
{
  byte  midiPitch;
  char *pitchStr;
} embouchureType;


// Globals:
//
int state       = TRIGGER_STATE_INITIAL;      // Begin at the Initial state
int currTrigger = 0;                          // Current Trigger analog read
int prevTrigger = 0;                          // Current Trigger analog read

embouchureType embouchureReg1[EMBOUCHURE_NUM_ENTRIES] =
{                                                       // Valve pos 123   MIDI pitch F#3(42)-C4(48)
  {48, "C4"},                                           // Fingering 000 = MIDI pitch C4 in Register 1
  {DEAD_PITCH, "DEAD"},                         // Fingering 001 = MIDI pitch -- in Register 1
  {47, "B3"},                                           // Fingering 010 = MIDI pitch B3 in Register 1
  {44, "G#3"},                                          // Fingering 011 = MIDI pitch G#3 in Register 1
  {46, "A#3"},                                          // Fingering 100 = MIDI pitch A#3 in Register 1
  {43, "G3"},                                           // Fingering 101 = MIDI pitch G3 in Register 1
  {45, "A3"},                                           // Fingering 110 = MIDI pitch A3 in Register 1
  {42, "F#3"}                                           // Fingering 111 = MIDI pitch F#3 in Register 1
};

embouchureType embouchureReg2[EMBOUCHURE_NUM_ENTRIES] =
{                                                       // Valve pos 123   MIDI pitch C#4(49)-G4(55)
  {55, "G4"},                                           // Fingering 000 = MIDI pitch G4 in Register 2
  {DEAD_PITCH, "DEAD"},                         // Fingering 001 = MIDI pitch -- in Register 2
  {54, "F#4"},                                          // Fingering 010 = MIDI pitch F#4 in Register 2
  {51, "D#4"},                                          // Fingering 011 = MIDI pitch D#4 in Register 2
  {53,      "F4"},                                      // Fingering 100 = MIDI pitch F4 in Register 2
```

```
  {50, "D4"},                                   // Fingering 101 = MIDI pitch D4 in Register 2
  {52, "E4"},                                   // Fingering 110 = MIDI pitch E4 in Register 2
  {49, "C#4"}                                   // Fingering 111 = MIDI pitch C#4 in Register 2
};

embouchureType embouchureReg3[EMBOUCHURE_NUM_ENTRIES] =
{                                               // Valve pos 123   MIDI pitch G#4(56)-C5(60)
  {60, "C5"},                                   // Fingering 000 = MIDI pitch C5 in Register 3
  {DEAD_PITCH, "DEAD"},                 // Fingering 001 = MIDI pitch -- in Register 3
  {59, "B4"},                                   // Fingering 010 = MIDI pitch B4 in Register 3
  {56, "G#4"},                                  // Fingering 011 = MIDI pitch G#4 in Register 3
  {58, "A#4"},                                  // Fingering 100 = MIDI pitch A#4 in Register 3
  {DEAD_PITCH, "DEAD"},                 // Fingering 101 = MIDI pitch -- in Register 3
  {57, "A4"},                                   // Fingering 110 = MIDI pitch A4 in Register 3
  {DEAD_PITCH, "DEAD"}                  // Fingering 111 = MIDI pitch -- in Register 3
};

embouchureType embouchureReg4[EMBOUCHURE_NUM_ENTRIES] =
{                                               // Valve pos 123   MIDI pitch C#5(61)-E5(64)
  {64, "E5"},                                   // Fingering 000 = MIDI pitch E5 in Register 4
  {DEAD_PITCH, "DEAD"},                 // Fingering 001 = MIDI pitch -- in Register 4
  {63, "D#5"},                                  // Fingering 010 = MIDI pitch D#5 in Register 4
  {DEAD_PITCH, "DEAD"},                 // Fingering 011 = MIDI pitch -- in Register 4
  {62, "D5"},                                   // Fingering 100 = MIDI pitch D5 in Register 4
  {DEAD_PITCH, "DEAD"},                 // Fingering 101 = MIDI pitch -- in Register 4
  {61, "C#5"},                                  // Fingering 110 = MIDI pitch C#5 in Register 4
  {DEAD_PITCH, "DEAD"}                  // Fingering 111 = MIDI pitch -- in Register 4
};

embouchureType embouchureReg5[EMBOUCHURE_NUM_ENTRIES] =
{                                               // Valve pos 123   MIDI pitch F5(65)-A5(69)
  {67, "G5"},                                   // Fingering 000 = MIDI pitch G5 in Register 5
  {DEAD_PITCH, "DEAD"},                 // Fingering 001 = MIDI pitch -- in Register 5
  {66, "F#5"},                                  // Fingering 010 = MIDI pitch F#5 in Register 5
  {68, "G#5"},                                  // Fingering 011 = MIDI pitch G#5 in Register 5
  {65, "F5"},                                   // Fingering 100 = MIDI pitch F5 in Register 5
  {DEAD_PITCH, "DEAD"},                 // Fingering 101 = MIDI pitch -- in Register 5
  {69, "A5"},                                   // Fingering 110 = MIDI pitch A5 in Register 5
  {DEAD_PITCH, "DEAD"}                  // Fingering 111 = MIDI pitch -- in Register 5
};

embouchureType embouchureReg6[EMBOUCHURE_NUM_ENTRIES] =
{                                               // Valve pos 123   MIDI pitch A#5(70)-C6(72)
  {72, "C6"},                                   // Fingering 000 = MIDI pitch C6 in Register 5
  {DEAD_PITCH, "DEAD"},                 // Fingering 001 = MIDI pitch -- in Register 5
  {71, "B5"},                                   // Fingering 010 = MIDI pitch B5 in Register 5
  {DEAD_PITCH, "DEAD"},                 // Fingering 011 = MIDI pitch -- in Register 5
  {70, "A#5"},                                  // Fingering 100 = MIDI pitch A#5 in Register 5
  {DEAD_PITCH, "DEAD"},                 // Fingering 101 = MIDI pitch -- in Register 5
  {DEAD_PITCH, "DEAD"},                 // Fingering 110 = MIDI pitch -- in Register 5
  {DEAD_PITCH, "DEAD"}                  // Fingering 111 = MIDI pitch -- in Register 5
};

#ifdef EXECUTION_MODE_DEBUG
char *stateStr[5] =                             // Debug state strings
{
  "RESERVED",                                   // Reserved for now/unused
  "INITIAL",                                    // Initial state
  "RESET  ",                                    // Reset state (all pitch off)
  "FORWARD",                                    // State of trigger in forward motion
  "REVERSE"                                     // State of trigger in reverse motion
};

char *fingeringStr[8] =                                         // Debug fingering strings
{
  "000",                                                // valve1=off, valve2=off, valve3=off
  "001",                                                // valve1=off, valve2=off, valve3=on
  "010",                                                // valve1=off, valve2=on, valve3=off
  "011",                                                // valve1=off, valve2=on, valve3=on
  "100",                                                // valve1=on, valve2=off, valve3=off
```

```
  "101",                                                    // valve1=on, valve2=off, valve3=on
  "110",                                                    // valve1=on, valve2=on, valve3=off
  "111"                                                     // valve1=on, valve2=on, valve3=on
};
#endif


// Forwards:
//
void pitchOn();
int getEmbouchureReg(int sweep);


// ========================================================================
// ================================= SETUP ================================
// ========================================================================
//
void setup()
{
  pinMode(TRIGGER_PIN, INPUT);
  pinMode(VALVE_1_PIN, INPUT);
  pinMode(VALVE_2_PIN, INPUT);
  pinMode(VALVE_3_PIN, INPUT);
  pinMode(EMBOUCHURE_PIN, INPUT);

  analogReadResolution(10);

  NEXT_STATE(TRIGGER_STATE_INITIAL);
}


// ========================================================================
// ================================= LOOP =================================
// ========================================================================
//
void loop()
{
#ifdef EXECUTION_MODE_DEBUG
  Serial.print("CYCLE:  ");
  Serial.print(" state=");
  Serial.print(stateStr[state]);
  Serial.print(" currTrigger=");
  Serial.println(currTrigger);
#endif

  currTrigger = analogRead(TRIGGER_PIN);

  switch (state)
  {
    case TRIGGER_STATE_INITIAL:
      if (TRIGGER_POSITION_RESET(currTrigger))
      {
        NEXT_STATE(TRIGGER_STATE_RESET);
      }
      else  // not in Trigger reset
      {
        NEXT_STATE(TRIGGER_STATE_FORWARD);
      }
      break;

    case TRIGGER_STATE_RESET:
      if (! TRIGGER_POSITION_RESET(currTrigger))
      {
        NEXT_STATE(TRIGGER_STATE_FORWARD);
      }
      break;

    case TRIGGER_STATE_FORWARD:
      if (currTrigger < prevTrigger)  // maybe in reverse direction?
      {
```

```
          if (TRIGGER_REVERSE_DIR(currTrigger, prevTrigger))
          {
            // Trigger fluctuation indicates reverse direction - sound the pitch
            NEXT_STATE(TRIGGER_STATE_REVERSE);
            pitchOn();
          }
        }
        break;

      case TRIGGER_STATE_REVERSE:
        if (TRIGGER_POSITION_RESET(currTrigger))
        {
          NEXT_STATE(TRIGGER_STATE_RESET);

#ifdef EXECUTION_MODE_MIDI
          // All pitch off
          usbMIDI.sendControlChange(CONTROL_NOTE_OFF_CMD, CONTROL_NOTE_OFF_VALUE, PRIMARY_CHANNEL);
#endif
        }
        else if (currTrigger >= prevTrigger)
        {
          NEXT_STATE(TRIGGER_STATE_FORWARD);
        }
        break;

      default: break;
    }

    prevTrigger = currTrigger;

    delay(CYCLE_RESOLUTION);
}


// Functions:
//
// ********************************************************************************
// * pitchOn()
// * Retrieve all the resources needed to sound the selected pitch.
// ********************************************************************************
//
void pitchOn()
{
  byte  velocity   = 0;
  int   valve1     = 0;
  int   valve2     = 0;
  int   valve3     = 0;
  int   fingering  = 0;
  int   embouchure = 0;
  int   pitch      = 0;
  char *pitchStr   = '\0';

  // ------------------------------------
  // Calculate Velocity
  // ------------------------------------

  if (currTrigger > TRIGGER_POSITION_MAX)
    currTrigger = TRIGGER_POSITION_MAX;

  velocity = currTrigger - TRIGGER_THRESHOLD;

  // ------------------------------------
  // Calculate Valve Index Mapping
  // ------------------------------------

  if (VALVE_1_POSITION_ON(analogRead(VALVE_1_PIN)))
    valve1 = VALVE_1_BIT;

  if (VALVE_2_POSITION_ON(analogRead(VALVE_2_PIN)))
    valve2 = VALVE_2_BIT;
```

```
  if (VALVE_3_POSITION_ON(analogRead(VALVE_3_PIN)))
    valve3 = VALVE_3_BIT;

  fingering = valve1 | valve2 | valve3;

  // ------------------------------------
  // Calculate Embouchure Register
  // ------------------------------------

  embouchure = getEmbouchureReg(analogRead(EMBOUCHURE_PIN));

  switch (embouchure)
  {
    case EMBOUCHURE_REG_1:
      pitch    = embouchureReg1[fingering].midiPitch;
      pitchStr = embouchureReg1[fingering].pitchStr;
      break;

    case EMBOUCHURE_REG_2:
      pitch    = embouchureReg2[fingering].midiPitch;
      pitchStr = embouchureReg2[fingering].pitchStr;
      break;

    case EMBOUCHURE_REG_3:
      pitch    = embouchureReg3[fingering].midiPitch;
      pitchStr = embouchureReg3[fingering].pitchStr;
      break;

    case EMBOUCHURE_REG_4:
      pitch    = embouchureReg4[fingering].midiPitch;
      pitchStr = embouchureReg4[fingering].pitchStr;
      break;

    case EMBOUCHURE_REG_5:
      pitch    = embouchureReg5[fingering].midiPitch;
      pitchStr = embouchureReg5[fingering].pitchStr;
      break;

    case EMBOUCHURE_REG_6:
      pitch    = embouchureReg6[fingering].midiPitch;
      pitchStr = embouchureReg6[fingering].pitchStr;
      break;
  }

  // ------------------------------------
  // Ignore a dead pitch
  // ------------------------------------

  if (pitch == DEAD_PITCH)
  {
#ifdef EXECUTION_MODE_DEBUG
    // ------------------------------------
    // Debug
    // ------------------------------------

    Serial.println("PITCH:   DEAD");
#endif
    return;
  }

#ifdef EXECUTION_MODE_MIDI
  // ------------------------------------
  // Sound the Pitch in polyphonic
  // (allow notes to overlay each other)
  // ------------------------------------

  usbMIDI.sendNoteOn(pitch, velocity, PRIMARY_CHANNEL);
  usbMIDI.sendPolyPressure(pitch, velocity, PRIMARY_CHANNEL);
#endif
```

```
#ifdef EXECUTION_MODE_DEBUG
  // ------------------------------------
  // Debug
  // ------------------------------------

  Serial.print("PITCH:  ");
  Serial.print(" velocity=");
  Serial.print(velocity);
  Serial.print(" fingering=");
  Serial.print(fingeringStr[fingering]);
  Serial.print(" embouchure=");
  Serial.print(embouchure);
  Serial.print(" pitch=");
  Serial.println(pitchStr);
#endif
}


// ********************************************************************************
// * getEmbouchureReg()
// * Determine which Embouchure Register the sensor sweep is currently in.
// ********************************************************************************
//
int getEmbouchureReg(int sweep)
{
  int increment = (EMBOUCHURE_MAX_SWEEP - EMBOUCHURE_MIN_SWEEP) / EMBOUCHURE_NUM_REG;
  int delta     = (sweep - EMBOUCHURE_MIN_SWEEP);
  int result    = 0;

  // ???
  return (EMBOUCHURE_REG_1);

  if (delta < 0)
    return (EMBOUCHURE_REG_1);

  result = (delta / increment) + 1;

  if (result > EMBOUCHURE_NUM_REG)
    result = EMBOUCHURE_NUM_REG;

  return (result);
}
```

12.6 Appendix F: Current Source Code (v4.3)

```
// PURPOSE:
// SansTrumpet implementation, including representations of:
//      - Valve functionality
//      - Trigger functionality
//      - Embouchure register and
//      - Amplitude envelope
//
// HW VERSION:  1.0
// SW VERSION: 4.3
//
// DATE:       2/1/14
//
// AUTHOR:     Anthony Mastromattei and Thomas Nguyen
//
// REVISION HISTORY:
//      1.0     Initial version with only Trigger implementation
//      2.0     Valve implementation
//      3.0     Added debug functionality to the Serial Monitor
//      4.0     Added calibration/eeprom and sampling functionality
//      4.1     Implemented embouchure register selection algorithm
//       4.2     MIDI Control-Change-All-Notes-Off doesn't work, simulate with "All Pitches Off"
//       4.3     PCB Integration
//
```

```
#include <EEPROM.h>

// Constants:
//
// Versions
#define VERSION_HW                  0x0100                          // HW version (1st byte major
ver, 2nd byte minor)
#define VERSION_SW                  0x0403                          // SW version (1st byte major
ver, 2nd byte minor)

// Execution (x-out to undefine)
#define xEXECUTION_MODE_DEBUG_CYCLE                                 // Expose cycle debug code
#define EXECUTION_MODE_DEBUG_PITCH                                  // Expose pitch debug code
#define EXECUTION_MODE_DEBUG_CALIB                                  // Expose calibration debug
code
#define EXECUTION_MODE_DEBUG_VALVES                                 // Expose valve debug code
#define EXECUTION_MODE_MIDI                                         // Expose MIDI release code

// Tweaks
#define READINGS_PER_SAMPLE         2000                            // Number of sensor readings
per sample
#define CYCLE_RESOLUTION            5                               // Cycle delay (ms) to
improve flexSensor predictability

// MIDI
#define PRIMARY_CHANNEL             1                                   // Primary MIDI channel
#define MIDI_RANGE                  128                                 // Common MIDI range
for velocity, etc.
#define CONTROL_NOTE_OFF_CMD        123                                 // Control change all-
notes-off command
#define CONTROL_NOTE_OFF_VALUE      0                               // Control change all-notes-
off value
#define DEAD_PITCH                  255                                 // Unplayable pitch

// Trigger
#define TRIGGER_POSITION_MAX        (calibTriggerSweep[MIDI_RANGE-1])       // Trigger maximum
position range
#define TRIGGER_POSITION_RESET(s)   ((s) < (calibTriggerSweep[0]))  // Trigger reset condition
#define TRIGGER_REVERSE_DIR(c,p)    (((p) - (c)) > 2)                   // Trigger fluctuation
large enough for reverse direction

// Valves
#define VALVE_1_POSITION_OFF(s)     ((s) <  (calibValve1Threshold)) // Valve 1 position off
condition
#define VALVE_1_POSITION_ON(s)      ((s) >= (calibValve1Threshold)) // Valve 1 position on
condition
#define VALVE_2_POSITION_OFF(s)     ((s) <  (calibValve2Threshold)) // Valve 2 position off
condition
#define VALVE_2_POSITION_ON(s)      ((s) >= (calibValve2Threshold)) // Valve 2 position on
condition
#define VALVE_3_POSITION_OFF(s)     ((s) <  (calibValve3Threshold)) // Valve 3 position off
condition
#define VALVE_3_POSITION_ON(s)      ((s) >= (calibValve3Threshold)) // Valve 3 position on
condition

#define VALVE_1_BIT                 4                               // Valve 1 bit position
#define VALVE_2_BIT                 2                               // Valve 2 bit position
#define VALVE_3_BIT                 1                               // Valve 4 bit position

// Embouchure
#define EMBOUCHURE_NUM_ENTRIES      8                               // Maximum number of
Embouchure entries
#define EMBOUCHURE_NUM_REG          6                               // Number of Embouchure
Registers
#define EMBOUCHURE_MIN_SWEEP        (calibEmbouchureSweep[0])           // Minimum embouchure
sweep range
#define EMBOUCHURE_MAX_SWEEP        (calibEmbouchureSweep[MIDI_RANGE-1])// Maximum embouchure
sweep range
```

```
#define EMBOUCHURE_REG_1            1                               // Embouchure Register range
1
#define EMBOUCHURE_REG_2            2                               // Embouchure Register range
2
#define EMBOUCHURE_REG_3            3                               // Embouchure Register range
3
#define EMBOUCHURE_REG_4            4                               // Embouchure Register range
4
#define EMBOUCHURE_REG_5            5                               // Embouchure Register range
5
#define EMBOUCHURE_REG_6            6                               // Embouchure Register range
6

// Sustain
#define SUSTAIN_ON(s)              ((s) == 0)                           // Sustain pedal on?

// States
#define TRIGGER_STATE_INITIAL       1                               // Initial state
#define TRIGGER_STATE_RESET         2                               // Reset state (all pitch
off)
#define TRIGGER_STATE_FORWARD       3                               // State of trigger in
forward motion
#define TRIGGER_STATE_REVERSE       4                               // State of trigger in
reverse motion
#define NEXT_STATE(s)               state = (s)                             // Next state
transition

// Analog pins
#define TRIGGER_PIN                 5                               // Trigger pin value
#define VALVE_1_PIN                 7                               // Valve 1 pin value
#define VALVE_2_PIN                 8                               // Valve 2 pin value
#define VALVE_3_PIN                 9                               // Valve 3 pin value
#define EMBOUCHURE_PIN              6                               // Embouchure pin value
#define SUSTAIN_PIN                 5                                  // Sustain pedal pin value
#define VALVES_CALIB_BUTTON_PIN     0                               // Valves calibration button
pin
#define TRIGGER_CALIB_BUTTON_PIN    2                               // Trigger calibration button
pin
#define EMBOUCHURE_CALIB_BUTTON_PIN 1                               // Embouchure calibration
button pin
#define CALIB_LED_PIN               17                                     // Calibration LED pin

// EEPROM
#define EEPROM_VERSION_HW           0                               // EEPROM address for HW
version
#define EEPROM_VERSION_SW           2                               // EEPROM address for SW
version
#define EEPROM_VALVE_1_THRESHOLD    4                               // EEPROM address for Valve 1
threshold
#define EEPROM_VALVE_2_THRESHOLD    6                               // EEPROM address for Valve 2
threshold
#define EEPROM_VALVE_3_THRESHOLD    8                               // EEPROM address for Valve 3
threshold
#define EEPROM_TRIGGER_SWEEP        10                                     // EEPROM address for
Trigger sweep
#define EEPROM_EMBOUCHURE_SWEEP     266                                    // EEPROM address for
Embouchure sweep
#define EEPROM_UNCALIB_FLAG         0xffff                          // EEPROM flag indicating not
calibrated

// Control-Change-All-Notes-Off simulation (All-Pitches-Off)
#define PITCH_BUFFER_SIZE           1000                            // Number of pitches to track
in a circular buffer

// Calibration
#define CALIB_BUTTON_PRESSED        1023                            // Analog read value for
calibration buttons
#define IS_DEVICE_DATA_CALIB (                                             \
   (calibValve1Threshold    != EEPROM_UNCALIB_FLAG) &&                     \
   (calibTriggerSweep[0]    != EEPROM_UNCALIB_FLAG) &&                     \
   (calibEmbouchureSweep[0] != EEPROM_UNCALIB_FLAG)                  \
```

```
)


// Datatypes:
//
typedef struct                                                  // Embouchure datatype
{
   byte  midiPitch;                                             // Midi pitch value
   char *pitchStr;                                              // Associated pitch string
for debug
} embouchureType;


// Globals:
//
int state                = TRIGGER_STATE_INITIAL;               // Begin at the Initial state
int currTrigger          = 0;                                   // Current Trigger analog
read
int prevTrigger          = 0;                                   // Previous Trigger analog
read
int currPitch            = 0;                                       // Current pitch
bool isDeviceCalib       = false;                               // Device calibrated flag
int calibValve1Threshold = 0;                                   // Calibrated Valve 1 on/off
threshold
int calibValve2Threshold = 0;                                   // Calibrated Valve 2 on/off
threshold
int calibValve3Threshold = 0;                                   // Calibrated Valve 3 on/off
threshold
int calibTriggerSweep[MIDI_RANGE];                              // Calibrated Trigger sweep
int calibEmbouchureSweep[MIDI_RANGE];                           // Calibrated Embouchure
sweep
int pitchBuffer[PITCH_BUFFER_SIZE];                                 // Track pitches sounded
in circular buffer (All-Pitches-Off)
bool isPitchOff[MIDI_RANGE];                                        // Track which pitches
are reset for efficiency
int pitchIndex = 0;                                                 // Pitch buffer index

embouchureType embouchureReg1[EMBOUCHURE_NUM_ENTRIES] =
{                                                               // Valve pos 123   MIDI pitch
F#3(42)-C4(48)
   {48, "C4"},                                                  // Fingering 000 = MIDI pitch
C4 in Register 1
   {DEAD_PITCH, "DEAD"},                                        // Fingering 001 = MIDI pitch
-- in Register 1
   {47, "B3"},                                                  // Fingering 010 = MIDI pitch
B3 in Register 1
   {44, "G#3"},                                                        // Fingering 011 =
MIDI pitch G#3 in Register 1
   {46, "A#3"},                                                        // Fingering 100 =
MIDI pitch A#3 in Register 1
   {43, "G3"},                                                  // Fingering 101 = MIDI pitch
G3 in Register 1
   {45, "A3"},                                                  // Fingering 110 = MIDI pitch
A3 in Register 1
   {42, "F#3"}                                                  // Fingering 111 = MIDI pitch
F#3 in Register 1
};

embouchureType embouchureReg2[EMBOUCHURE_NUM_ENTRIES] =
{                                                               // Valve pos 123   MIDI pitch
C#4(49)-G4(55)
   {55, "G4"},                                                  // Fingering 000 = MIDI pitch
G4 in Register 2
   {DEAD_PITCH, "DEAD"},                                        // Fingering 001 = MIDI pitch
-- in Register 2
   {54, "F#4"},                                                        // Fingering 010 =
MIDI pitch F#4 in Register 2
   {51, "D#4"},                                                        // Fingering 011 =
MIDI pitch D#4 in Register 2
   {53, "F4"},                                                  // Fingering 100 = MIDI pitch
F4 in Register 2
```

```
   {50, "D4"},                                                  // Fingering 101 = MIDI pitch
D4 in Register 2
   {52, "E4"},                                                  // Fingering 110 = MIDI pitch
E4 in Register 2
   {49, "C#4"}                                                  // Fingering 111 = MIDI pitch
C#4 in Register 2
};

embouchureType embouchureReg3[EMBOUCHURE_NUM_ENTRIES] =
{                                                               // Valve pos 123   MIDI pitch
G#4(56)-C5(60)
   {60, "C5"},                                                  // Fingering 000 = MIDI pitch
C5 in Register 3
   {DEAD_PITCH, "DEAD"},                                        // Fingering 001 = MIDI pitch
-- in Register 3
   {59, "B4"},                                                  // Fingering 010 = MIDI pitch
B4 in Register 3
   {56, "G#4"},                                                         // Fingering 011 =
MIDI pitch G#4 in Register 3
   {58, "A#4"},                                                         // Fingering 100 =
MIDI pitch A#4 in Register 3
   {DEAD_PITCH, "DEAD"},                                        // Fingering 101 = MIDI pitch
-- in Register 3
   {57, "A4"},                                                  // Fingering 110 = MIDI pitch
A4 in Register 3
   {DEAD_PITCH, "DEAD"}                                                 // Fingering 111 =
MIDI pitch -- in Register 3
};

embouchureType embouchureReg4[EMBOUCHURE_NUM_ENTRIES] =
{                                                               // Valve pos 123   MIDI pitch
C#5(61)-E5(64)
   {64, "E5"},                                                  // Fingering 000 = MIDI pitch
E5 in Register 4
   {DEAD_PITCH, "DEAD"},                                        // Fingering 001 = MIDI pitch
-- in Register 4
   {63, "D#5"},                                                         // Fingering 010 =
MIDI pitch D#5 in Register 4
   {DEAD_PITCH, "DEAD"},                                        // Fingering 011 = MIDI pitch
-- in Register 4
   {62, "D5"},                                                  // Fingering 100 = MIDI pitch
D5 in Register 4
   {DEAD_PITCH, "DEAD"},                                        // Fingering 101 = MIDI pitch
-- in Register 4
   {61, "C#5"},                                                         // Fingering 110 =
MIDI pitch C#5 in Register 4
   {DEAD_PITCH, "DEAD"}                                                 // Fingering 111 =
MIDI pitch -- in Register 4
};

embouchureType embouchureReg5[EMBOUCHURE_NUM_ENTRIES] =
{                                                               // Valve pos 123   MIDI pitch
F5(65)-A5(69)
   {67, "G5"},                                                  // Fingering 000 = MIDI pitch
G5 in Register 5
   {DEAD_PITCH, "DEAD"},                                        // Fingering 001 = MIDI pitch
-- in Register 5
   {66, "F#5"},                                                         // Fingering 010 =
MIDI pitch F#5 in Register 5
   {68, "G#5"},                                                         // Fingering 011 =
MIDI pitch G#5 in Register 5
   {65, "F5"},                                                  // Fingering 100 = MIDI pitch
F5 in Register 5
   {DEAD_PITCH, "DEAD"},                                        // Fingering 101 = MIDI pitch
-- in Register 5
   {69, "A5"},                                                  // Fingering 110 = MIDI pitch
A5 in Register 5
   {DEAD_PITCH, "DEAD"}                                                 // Fingering 111 =
MIDI pitch -- in Register 5
};
```

```
embouchureType embouchureReg6[EMBOUCHURE_NUM_ENTRIES] =
{                                                                   // Valve pos 123   MIDI pitch
A#5(70)-C6(72)
   {72, "C6"},                                                      // Fingering 000 = MIDI pitch
C6 in Register 5
   {DEAD_PITCH, "DEAD"},                                            // Fingering 001 = MIDI pitch
-- in Register 5
   {71, "B5"},                                                      // Fingering 010 = MIDI pitch
B5 in Register 5
   {DEAD_PITCH, "DEAD"},                                            // Fingering 011 = MIDI pitch
-- in Register 5
   {70, "A#5"},                                                            // Fingering 100 =
MIDI pitch A#5 in Register 5
   {DEAD_PITCH, "DEAD"},                                            // Fingering 101 = MIDI pitch
-- in Register 5
   {DEAD_PITCH, "DEAD"},                                            // Fingering 110 = MIDI pitch
-- in Register 5
   {DEAD_PITCH, "DEAD"}                                                    // Fingering 111 =
MIDI pitch -- in Register 5
};

#ifdef EXECUTION_MODE_DEBUG_CYCLE
char *stateStr[5] =                                                 // Debug state strings
{
   "RESERVED",                                                      // Reserved for now/unused
   "INITIAL",                                                       // Initial state
   "RESET  ",                                                       // Reset state (all pitch
off)
   "FORWARD",                                                       // State of trigger in
forward motion
   "REVERSE"                                                        // State of trigger in
reverse motion
};
#endif


#if defined(EXECUTION_MODE_DEBUG_PITCH) || defined(EXECUTION_MODE_DEBUG_VALVES)
char *fingeringStr[8] =                                                    // Debug fingering
strings
{
   "000",                                                          // valve1=off, valve2=off,
valve3=off
   "001",                                                          // valve1=off, valve2=off,
valve3=on
   "010",                                                          // valve1=off, valve2=on,
valve3=off
   "011",                                                          // valve1=off, valve2=on,
valve3=on
   "100",                                                          // valve1=on,  valve2=off,
valve3=off
   "101",                                                          // valve1=on,  valve2=off,
valve3=on
   "110",                                                          // valve1=on,  valve2=on,
valve3=off
   "111"                                                           // valve1=on,  valve2=on,
valve3=on
};
#endif


// Forwards:
//
int pitchOn();
int getEmbouchureReg(int sweep);
int readFlexSensor(int pin);
bool loadCalibData();
bool checkCalibButtons();
int readEepromInt16(int addr);
void writeEepromInt16(int addr, int value);
void blinkCalibLED(int ms);
void uncalibLEDpattern();
```

```
void calibValves();
void calibSweep(int *sweep, int calibPin, int sensorPin, int eepromAddr);
int lookupTriggerVelocity();
void allPitchesOff();


// ============================================================================
// ================================== SETUP ===================================
// ============================================================================
//
void setup()
{
   int i;

   // Wait for Teensy to stabilize
   delay(500);

   // Establish pin modes and read resolution
   pinMode(TRIGGER_PIN, INPUT);
   pinMode(VALVE_1_PIN, INPUT);
   pinMode(VALVE_2_PIN, INPUT);
   pinMode(VALVE_3_PIN, INPUT);
   pinMode(EMBOUCHURE_PIN, INPUT);
   pinMode(SUSTAIN_PIN, INPUT);
   pinMode(CALIB_LED_PIN, OUTPUT);

   analogReadResolution(10);

   // Assure Calibration LED is off
   digitalWrite(CALIB_LED_PIN, LOW);  // LED off

   // Read calibrations from EEPROM
   isDeviceCalib = loadCalibData();

   // Initialize All-Pitches-Off data
   for (i=0; i < PITCH_BUFFER_SIZE; i++) pitchBuffer[i] = -1;
   for (i-0; i < MIDI_RANGE; i++)        isPitchOff[i]  = false;
   pitchIndex = 0;

   // Initialize state machine
   NEXT_STATE(TRIGGER_STATE_INITIAL);
}


// ============================================================================
// ================================== LOOP ====================================
// ============================================================================
//
void loop()
{
   // ----- Is device calibrated? -----

   if (! isDeviceCalib)
   {
      isDeviceCalib = checkCalibButtons();
      if (isDeviceCalib)
      {
         // Device fully calibrated, record hw/sw versions
         writeEepromInt16(EEPROM_VERSION_HW, VERSION_HW);
         writeEepromInt16(EEPROM_VERSION_SW, VERSION_SW);
      }
      else  // device not calibrated yet
      {
         return;
      }
   }

   // ----- Allow a re-calibration -----

   checkCalibButtons();
```

```
   // ----- Run the state machine -----

   currTrigger = readFlexSensor(TRIGGER_PIN);

#ifdef EXECUTION_MODE_DEBUG_CYCLE
   Serial.print("CYCLE:  ");
   Serial.print(" state=");
   Serial.print(stateStr[state]);
   Serial.print(" currTrigger=");
   Serial.println(currTrigger);
#endif

   switch (state)
   {
      case TRIGGER_STATE_INITIAL:
         if (TRIGGER_POSITION_RESET(currTrigger))
         {
            NEXT_STATE(TRIGGER_STATE_RESET);
#ifdef EXECUTION_MODE_DEBUG_PITCH
            Serial.println("PITCH:   Trigger Reset1");
#endif

         }
         else  // not in Trigger reset
         {
            NEXT_STATE(TRIGGER_STATE_FORWARD);
         }
         break;

      case TRIGGER_STATE_RESET:
         if (! TRIGGER_POSITION_RESET(currTrigger))
         {
            NEXT_STATE(TRIGGER_STATE_FORWARD);
         }
         break;

      case TRIGGER_STATE_FORWARD:
         if (currTrigger < prevTrigger)  // maybe in reverse direction?
         {
            if (TRIGGER_REVERSE_DIR(currTrigger, prevTrigger))
            {
               // Trigger fluctuation indicates reverse direction - sound the pitch
               NEXT_STATE(TRIGGER_STATE_REVERSE);
               currPitch = pitchOn();
            }
         }
         break;

      case TRIGGER_STATE_REVERSE:
         if (TRIGGER_POSITION_RESET(currTrigger))
         {
            NEXT_STATE(TRIGGER_STATE_RESET);
#ifdef EXECUTION_MODE_DEBUG_PITCH
            Serial.println("PITCH:   Trigger Reset2");
#endif

#ifdef EXECUTION_MODE_MIDI
            // All-Pitches-Off
            allPitchesOff();
#endif
         }
         else if (currTrigger >= prevTrigger)
         {
            NEXT_STATE(TRIGGER_STATE_FORWARD);
         }
         break;

      default: break;
   }

   prevTrigger = currTrigger;
```

```
   delay(CYCLE_RESOLUTION);
}


// ============================================================================
// ============================== PRIMITIVES ==================================
// ============================================================================
//
// ****************************************************************************
// * pitchOn()
// * Retrieve all the resources needed to sound the selected pitch.
// ****************************************************************************
//
int pitchOn()
{
   byte  velocity        = 0;
   int   valve1          = 0;
   int   valve2          = 0;
   int   valve3          = 0;
   int   v1              = 0;
   int   v2              = 0;
   int   v3              = 0;
   int   fingering       = 0;
   int   embouchure      = 0;
   int   embouchureSweep = 0;
   int   pitch           = 0;
   char *pitchStr        = '\0';
   int   sustain         = 0;

   // ------------------------------------
   // Lookup Trigger Velocity
   // ------------------------------------

   velocity = lookupTriggerVelocity();

   // ------------------------------------
   // Calculate Valve Index Mapping
   // ------------------------------------

   v1=readFlexSensor(VALVE_1_PIN);
   v2=readFlexSensor(VALVE_2_PIN);
   v3=readFlexSensor(VALVE_3_PIN);

   if (VALVE_1_POSITION_ON(v1)) valve1 = VALVE_1_BIT;
   if (VALVE_2_POSITION_ON(v2)) valve2 = VALVE_2_BIT;
   if (VALVE_3_POSITION_ON(v3)) valve3 = VALVE_3_BIT;

   fingering = valve1 | valve2 | valve3;

   // ------------------------------------
   // Calculate Embouchure Register
   // ------------------------------------

   embouchureSweep = readFlexSensor(EMBOUCHURE_PIN);
   embouchure      = getEmbouchureReg(embouchureSweep);

   switch (embouchure)
   {
      case EMBOUCHURE_REG_1:
         pitch    = embouchureReg1[fingering].midiPitch;
         pitchStr = embouchureReg1[fingering].pitchStr;
         break;

      case EMBOUCHURE_REG_2:
         pitch    = embouchureReg2[fingering].midiPitch;
         pitchStr = embouchureReg2[fingering].pitchStr;
         break;

      case EMBOUCHURE_REG_3:
         pitch    = embouchureReg3[fingering].midiPitch;
```

```
         pitchStr = embouchureReg3[fingering].pitchStr;
         break;

      case EMBOUCHURE_REG_4:
         pitch    = embouchureReg4[fingering].midiPitch;
         pitchStr = embouchureReg4[fingering].pitchStr;
         break;

      case EMBOUCHURE_REG_5:
         pitch    = embouchureReg5[fingering].midiPitch;
         pitchStr = embouchureReg5[fingering].pitchStr;
         break;

      case EMBOUCHURE_REG_6:
         pitch    = embouchureReg6[fingering].midiPitch;
         pitchStr = embouchureReg6[fingering].pitchStr;
         break;
   }

   // ------------------------------------
   // Ignore a dead pitch
   // ------------------------------------

   if (pitch == DEAD_PITCH)
   {
#ifdef EXECUTION_MODE_DEBUG_PITCH
      // ------------------------------------
      // Debug Deadnote Pitch
      // ------------------------------------

      Serial.print("PITCH:   DEAD [fingering=");
      Serial.print(fingeringStr[fingering]);
      Serial.print("/reg=");
      Serial.print(embouchure);
      Serial.println("]");

#endif

#ifdef EXECUTION_MODE_DEBUG_VALVES
   // ------------------------------------
   // Debug Deadnote Valves
   // ------------------------------------

   Serial.print("VALVES:  ");
   Serial.print("pitch=DEAD");
   Serial.print(" fingering=");
   Serial.print(fingeringStr[fingering]);
   Serial.print(" v1[");
   Serial.print(calibValve1Threshold);
   Serial.print("]=");
   Serial.print(v1);
   Serial.print(" v2[");
   Serial.print(calibValve2Threshold);
   Serial.print("]=");
   Serial.print(v2);
   Serial.print(" v3[");
   Serial.print(calibValve3Threshold);
   Serial.print("]=");
   Serial.println(v3);
#endif

      return (pitch);
   }

   sustain = digitalRead(SUSTAIN_PIN);
   sustain = (SUSTAIN_ON(sustain) ? 1 : 0);

#ifdef EXECUTION_MODE_MIDI
   // ------------------------------------
   // Sound the pitch
   // ------------------------------------
```

```
   // Store the pitch in circular buffer for when All-Pitches-Off occurs
   pitchBuffer[pitchIndex] = pitch;
   if (pitchIndex++ == PITCH_BUFFER_SIZE)
   {
      pitchIndex = 0;
   }

   usbMIDI.sendNoteOn(pitch, velocity, PRIMARY_CHANNEL);

   if (! sustain)
   {
      delay(200);
      usbMIDI.sendNoteOff(pitch, 0, PRIMARY_CHANNEL);
   }
#endif

#ifdef EXECUTION_MODE_DEBUG_PITCH
   // ------------------------------------
   // Debug Pitch
   // ------------------------------------

   Serial.print("PITCH:  ");
   Serial.print(" velocity=");
   Serial.print(velocity);
   Serial.print(" fingering=");
   Serial.print(fingeringStr[fingering]);
   Serial.print(" embouchure=[sweep:");
   Serial.print(embouchureSweep);
   Serial.print("/reg:");
   Serial.print(embouchure);
   Serial.print("] pitch=");
   Serial.print(pitchStr);
   Serial.print("/");
   Serial.print(pitch);
   Serial.print(" sustain=");
   Serial.println(sustain);
#endif

#ifdef EXECUTION_MODE_DEBUG_VALVES
   // ------------------------------------
   // Debug Valves
   // ------------------------------------

   Serial.print("VALVES:  ");
   Serial.print("pitch=");
   Serial.print(pitchStr);
   Serial.print(" fingering=");
   Serial.print(fingeringStr[fingering]);
   Serial.print(" v1[");
   Serial.print(calibValve1Threshold);
   Serial.print("]=");
   Serial.print(v1);
   Serial.print(" v2[");
   Serial.print(calibValve2Threshold);
   Serial.print("]=");
   Serial.print(v2);
   Serial.print(" v3[");
   Serial.print(calibValve3Threshold);
   Serial.print("]=");
   Serial.println(v3);
#endif

   return (pitch);
}


// ************************************************************************
// * getEmbouchureReg()
// * Determine which Embouchure Register the sensor sweep is currently in.
// * The embouchure calibration sweep array is equally divided into 6 "buckets",
```

```
// * and the sweep passed in is simply compared to each bucket range boundary.
// * Once a range is matched, that register is returned.
// ************************************************************************
//
int getEmbouchureReg(int sweep)
{
   int spread = MIDI_RANGE / EMBOUCHURE_NUM_REG;
   int index  = spread;
   int reg    = EMBOUCHURE_REG_1;

   // First bucket
   if (sweep < calibEmbouchureSweep[index])
   {
      return (reg);
   }

   // Middle buckets
   for (reg = EMBOUCHURE_REG_2; reg < EMBOUCHURE_NUM_REG; reg++)
   {
      if ((sweep >= calibEmbouchureSweep[index]) && (sweep < calibEmbouchureSweep[index+spread]))
      {
         return (reg);
      }
      index += spread;
   }

   // Last bucket assumed
   return (reg);
}


// ************************************************************************
// * readFlexSensor()
// * This function returns a sampling of a FlexSensor reading.
// ************************************************************************
//
int readFlexSensor(int pin)
{
   int sampling = 0;
   int reading;

   for (reading = 0; reading < READINGS_PER_SAMPLE; reading++)
   {
      sampling += analogRead(pin);
   }

   sampling /= READINGS_PER_SAMPLE;

   return(sampling);
}


// ************************************************************************
// * loadCalibData()
// * This function loads all the calibration data from EEPROM flash memory, it
// * returns a 'true' value when successful, otherwise a 'false' indicates an
// * uncalibrated device.  The Teensy EEPROM interface is byte based but the
// * sensor readings require at least two bytes.  Therefore, the structure for
// * the SansTrumpet EEPROM is:
// *
// * EEPROM Address    EEPROM Calibration Data
// * --------------    -----------------------
// *    0              | hw-ver-major-rel-byte1   | hw-ver-minor-rel-byte2   |
// *    2              | sw-ver-major-rel-byte1   | sw-ver-minor-rel-byte2   |
// *    4              | valve1-threshold-byte1   | valve1-threshold-byte2   |
// *    6              | valve2-threshold-byte1   | valve2-threshold-byte2   |
// *    8              | valve3-threshold-byte1   | valve3-threshold-byte2   |
// *    10             | trigger-sweep[0]-byte1   | trigger-sweep[0]-byte2   |
// *    12             | trigger-sweep[1]-byte1   | trigger-sweep[1]-byte2   |
// *    14             | trigger-sweep[2]-byte1   | trigger-sweep[2]-byte2   |
// *    . . .                  . . .
```

```
// *    264            | trigger-sweep[127]-byte1 | trigger-sweep[127]-byte2 |
// *    266            | embouch-sweep[0]-byte1   | embouch-sweep[0]-byte2   |
// *    268            | embouch-sweep[1]-byte1   | embouch-sweep[1]-byte2   |
// *    270            | embouch-sweep[2]-byte1   | embouch-sweep[2]-byte2   |
// *    . . .                  . . .
// *    520            | embouch-sweep[127]-byte1 | embouch-sweep[127]-byte2 |
// *
// * Blink LED for one second after each calibrated data set is loaded (Valve
// * thresholds, Trigger sweep and Embouchure sweep).
// ************************************************************************
//
bool loadCalibData()
{
   int addr, i;

   // Load the Valves calibrated on/off thresholds
   calibValve1Threshold = readEepromInt16(EEPROM_VALVE_1_THRESHOLD);
   calibValve2Threshold = readEepromInt16(EEPROM_VALVE_2_THRESHOLD);
   calibValve3Threshold = readEepromInt16(EEPROM_VALVE_3_THRESHOLD);

   blinkCalibLED(1000);  // notify Valves calibration data loaded ok

   // Load the Trigger calibrated sweep
   for (i = 0, addr = EEPROM_TRIGGER_SWEEP;
        i < MIDI_RANGE;
       i++, addr += 2)
   {
      calibTriggerSweep[i] = readEepromInt16(addr);
   }

   blinkCalibLED(1000);  // notify Trigger calibration data loaded ok

   // Load the Embouchure calibrated sweep
   for (i = 0, addr = EEPROM_EMBOUCHURE_SWEEP;
        i < MIDI_RANGE;
       i++, addr += 2)
   {
      calibEmbouchureSweep[i] = readEepromInt16(addr);
   }

   blinkCalibLED(1000);  // notify Embouchure calibration data loaded ok

#ifdef EXECUTION_MODE_DEBUG_CALIB
   byte hwverMaj, hwverMin, swverMaj, swverMin;

   // Load the Valves calibrated on/off thresholds
   hwverMaj = EEPROM.read(EEPROM_VERSION_HW);
   hwverMin = EEPROM.read(EEPROM_VERSION_HW+1);
   swverMaj = EEPROM.read(EEPROM_VERSION_SW);
   swverMin = EEPROM.read(EEPROM_VERSION_SW+1);

   Serial.println("-----EEPROM DUMP-----");

   // Dump HW/SW Versions
   Serial.print("HW Version "); Serial.print(hwverMaj); Serial.print(".");
Serial.println(hwverMin);
   Serial.print("SW Version "); Serial.print(swverMaj); Serial.print(".");
Serial.println(swverMin);

   // Dump Valves Calibration
   Serial.print("Valve 1 Calibrated Threshold: "); Serial.println(calibValve1Threshold);
   Serial.print("Valve 2 Calibrated Threshold: "); Serial.println(calibValve2Threshold);
   Serial.print("Valve 3 Calibrated Threshold: "); Serial.println(calibValve3Threshold);

   // Dump Trigger Calibration
   for (i = 0; i < MIDI_RANGE; i++)
   {
      Serial.print("Trigger Calibrated Sweep: [index="); Serial.print(i);
      Serial.print(", value="); Serial.print(calibTriggerSweep[i]);
      Serial.println("]");
   }
```

```
   // Dump Embouchure Calibration
   for (i = 0; i < MIDI_RANGE; i++)
   {
      Serial.print("Embouchure Calibrated Sweep: [index="); Serial.print(i);
      Serial.print(", value="); Serial.print(calibEmbouchureSweep[i]);
      Serial.println("]");
   }
#endif

   return (IS_DEVICE_DATA_CALIB);  // calibration status
}


// ***************************************************************************
// * readEepromInt16()
// * This function reads and returns a 16bit integer from EEPROM at a specified
// * address.  Shifts and or's two bytes.
// ***************************************************************************
//
int readEepromInt16(int addr)
{
   int  value = 0;
   byte byte1 = 0, byte2 = 0;

   byte1 = EEPROM.read(addr);
   byte2 = EEPROM.read(addr+1);
   value = (byte1 << 8) | byte2;

   return (value);
}


// ***************************************************************************
// * writeEepromInt16()
// * This function writes a 16bit integer to EEPROM at a specified address.
// * Shifts and or's two bytes.
// ***************************************************************************
//
void writeEepromInt16(int addr, int value)
{
   byte byte1 = 0, byte2 = 0;

   byte1 = value >> 8;
   byte2 = value;
   EEPROM.write(addr, byte1);
   EEPROM.write(addr+1, byte2);
}


// ***************************************************************************
// * blinkCalibLED()
// * This function blinks the Calibration LED for a specified amount of ms.
// ***************************************************************************
//
void blinkCalibLED(int ms)
{
   digitalWrite(CALIB_LED_PIN, HIGH);  // LED on
   delay(ms);
   digitalWrite(CALIB_LED_PIN, LOW);  // LED off
   delay(ms);
}


// ***************************************************************************
// * checkCalibButtons()
// * This function checks whether the user intends to calibrate by checking
// * the corresponding calibration buttons for Trigger, Valves or Embouchure.
// * Calls the appropriate calibration function when applicable.
// * Returns 'true' when all data is calibrated, otherwise returns 'false'.
// * Note - only one button can be active at one time.
```

```
// ****************************************************************************
//
bool checkCalibButtons()
{
   // Have any calibration buttons been pressed?
   if (analogRead(VALVES_CALIB_BUTTON_PIN) == CALIB_BUTTON_PRESSED)
      calibValves();

   if (analogRead(TRIGGER_CALIB_BUTTON_PIN) == CALIB_BUTTON_PRESSED)
      calibSweep(calibTriggerSweep, TRIGGER_CALIB_BUTTON_PIN, TRIGGER_PIN, EEPROM_TRIGGER_SWEEP);

   if (analogRead(EMBOUCHURE_CALIB_BUTTON_PIN) == CALIB_BUTTON_PRESSED)
      calibSweep(calibEmbouchureSweep, EMBOUCHURE_CALIB_BUTTON_PIN, EMBOUCHURE_PIN,
EEPROM_EMBOUCHURE_SWEEP);

   // Is all device data calibrated now?
   if (IS_DEVICE_DATA_CALIB)
   {
      return (true);
   }
   else  // device data still not all calibrated
   {
      uncalibLEDpattern();
      return (false);
   }
}


// ****************************************************************************
// * uncalibLEDpattern()
// * This function blinks an LED sequence to indicate the device is not
// * calibrated yet.
// ****************************************************************************
//
void uncalibLEDpattern()
{
   blinkCalibLED(500);
   blinkCalibLED(500);
   blinkCalibLED(2000);
}


// ****************************************************************************
// * calibValves()
// * This function calibrates the valves but assures the calibration button
// * has first been pressed for 3 seconds.  Once calibration is complete, the
// * calibration LED blinks for 1 second.
// ****************************************************************************
//
void calibValves()
{
   int i;

   // Assure the calibration button is being held steady 3sec for an intended
   // calibration - in 10ms increments.  Otherwise assume mistakenly pressed.
   //
   for (i = 0; i < 300; i++)
   {
      if (analogRead(VALVES_CALIB_BUTTON_PIN) != CALIB_BUTTON_PRESSED)
         return;  // abort calibration
      delay(10);
   }

   // Perform the Valves calibration by locking in the Valves sensor readings
   calibValve1Threshold = readFlexSensor(VALVE_1_PIN);
   calibValve2Threshold = readFlexSensor(VALVE_2_PIN);
   calibValve3Threshold = readFlexSensor(VALVE_3_PIN);

   // Write the calibrations to EEPROM
   writeEepromInt16(EEPROM_VALVE_1_THRESHOLD, calibValve1Threshold);
   writeEepromInt16(EEPROM_VALVE_2_THRESHOLD, calibValve2Threshold);
```

```
   writeEepromInt16(EEPROM_VALVE_3_THRESHOLD, calibValve3Threshold);

   // Alert calibration complete by blinking the LED
   blinkCalibLED(1000);  // one second blink

#ifdef EXECUTION_MODE_DEBUG_CALIB
   Serial.println("-----VALVES CALIBRATION DUMP-----");

   // Dump Valves Calibration
   Serial.print("Valve 1 Calibrated Threshold: "); Serial.println(calibValve1Threshold);
   Serial.print("Valve 2 Calibrated Threshold: "); Serial.println(calibValve2Threshold);
   Serial.print("Valve 3 Calibrated Threshold: "); Serial.println(calibValve3Threshold);
#endif
}


// ***************************************************************************
// * calibSweep()
// * This function calibrates the Embouchure and Trigger sweep but assures
// * the calibration button has first been pressed for 3 seconds.  During
// * calibration the LED blinks continuously showing calibration is in progress.
// ***************************************************************************
//
void calibSweep(int *sweep, int calibPin, int sensorPin, int eepromAddr)
{
   int i;

   // Assure the calibration button is being held steady 3sec for an intended
   // calibration - in 10ms increments.  Otherwise assume mistakenly pressed.
   //
   for (i = 0; i < 300; i++)
   {
      if (analogRead(calibPin) != CALIB_BUTTON_PRESSED)
         return;  // abort calibration
      delay(10);
   }

   // Perform the Valves calibration by locking in the sweep sensor readings.
   // Iterate 127 time with a 24ms delay between each iteration, resulting in
   // a 3sec calibration recording sweep.  Blink the calibration 'in progress'
   // LED pattern to also accomodate the delay (12ms on + 12ms off).
   //
   for (i = 0; i < MIDI_RANGE; i++)
   {
      sweep[i] = readFlexSensor(sensorPin);            // read current sweep
      writeEepromInt16(eepromAddr, sweep[i]);          // write calibration to EEPROM
      eepromAddr += 2;                                 // next 2byte EEPROM location
      blinkCalibLED(12);                               // calibration in-progress blink
   }

#ifdef EXECUTION_MODE_DEBUG_CALIB
   Serial.println("-----SWEEP CALIBRATION DUMP-----");

   // Dump Trigger Calibration
   for (i = 0; i < MIDI_RANGE; i++)
   {
      Serial.print("Calibrated Sweep: [index="); Serial.print(i);
      Serial.print(", value="); Serial.print(sweep[i]);
      Serial.println("]");
   }
#endif
}


// ***************************************************************************
// * lookupTriggerVelocity()
// * This function returns the Trigger velocity by searching for the closest
// * approximation of the current Trigger in the Trigger Sweep Calibration
// * array.  Once found, the index of that entry represents the pitch velocity
// * and returned.
// * Note - if response time too slow, change to binary search (from linear).
```

```
// ****************************************************************************
//
int lookupTriggerVelocity()
{
   int i;

   // Trigger position greater than calibrated max?
   if (currTrigger > TRIGGER_POSITION_MAX)
      return (MIDI_RANGE-1);

   // Trigger position less than calibrated min?
   if (TRIGGER_POSITION_RESET(currTrigger))
      return (0);

   // Trigger position should be between calibrated min to max
   for (i = 0; i < (MIDI_RANGE-1); i++)
   {
      if ((currTrigger >= calibTriggerSweep[i]) &&
          (currTrigger <= calibTriggerSweep[i+1]))
         return (i);
   }
}


// ****************************************************************************
// * allPitchesOff()
// * This function simulates the Control-Change-All-Notes-Off functionality
// * (All-Pitches-Off) since it doesn't perform to spec.  Traverse the pitch
// * buffer array, shutting each pitch off.  Also, track the pitches shut off
// * to avoid redundant MIDI messages.
// ****************************************************************************
//
void allPitchesOff()
{
   int i;
   int pitch;

   // Shutdown all pitches sounded
   for (i=0;
        ((i < PITCH_BUFFER_SIZE) && (pitchBuffer[i] != -1));
        i++)
   {
      pitch = pitchBuffer[i];

      if (! isPitchOff[pitch])
      {
#ifdef EXECUTION_MODE_DEBUG_PITCH
         Serial.print("PITCH-OFF: "); Serial.println(pitch);
#endif
         usbMIDI.sendNoteOff(pitch, 0, PRIMARY_CHANNEL);
         isPitchOff[pitch] = true;
      }
   }

   // Re-initialize data
   pitchIndex = 0;
   for (i=0; i < PITCH_BUFFER_SIZE; i++) pitchBuffer[i] = -1;
   for (i=0; i < MIDI_RANGE; i++)        isPitchOff[i]  = false;
}
```