GloveSense MIDI-capable Dataglove

Presented to the faculty of

Worcester Polytechnic Institute

In partial fulfillment of the requirements for the Degree of Bachelor of Science by

Mert Erad ____________________

Russell Hedlund ____________________

Nathan Wells ____________________

Nathan Wiegman ____________________

Advised by Professor William Michalson ____________________

On the date March 27, 2015

Figure 1: GloveSense Dataglove

Abstract

The goal of this project is to provide a wearable device that enables the user to control MIDI (Musical Instrument Digital Interface) devices with gestures. Achieving this goal will allow performing artists to better control their music. The dataglove and accompanying software will be able to detect gestures through the use of piezoresistive flex sensors, accelerometers, gyroscopes, and magnetometers. These gestures will then be interpreted into audio modulation, which is used to create audio output.

Acknowledgements

We would like to express the utmost gratitude to Professor William Michalson for his guidance throughout this project. His insights saved the team from numerous pitfalls, and were a major contribution to the success of this project.

Additionally, we would like to thank Bob Boisse for his help with parts acquisition and making the shop available for our hardware needs.

Contents

1 Introduction **7**

2 Customer Requirements **10**

2.1 Derived Specifications 11

3 Literature Review **16**

3.1 Prior Art 16

3.2 MIDI - Musical Instrument Digital Interface 21

3.3 Max/MSP/Jitter 22

3.4 Attitude and Heading Reference System 23

3.5 Summary 24

4 Methodology **25**

4.1 Arudino Development 25

4.1.1 Flex Sensors 27

4.1.2 Orientation Measurement 29

4.1.3 Bluetooth Serial Communication 31

4.1.4 3D-Printed Case 32

4.2 Max Software 33

4.2.1 Flex Sensor Interpretation and Gesture Recognition 34

4.2.2 AHRS Interpretation 37

4.2.3 Ableton Live Interface . . . 38
4.3 Design Summary . . . 39

5 Results **40**

5.1 Hardware Limitations and Improvements . . . 40
5.1.1 Mi.Mu Glove . . . 40
5.1.2 Accelerometer . . . 42
5.1.3 Printed Circuit Board . . . 42
5.2 Software Limitations and Improvements . . . 43
5.2.1 Accelerometer Interpretation . . . 43
5.3 Gesture Recognition Accuracy . . . 43
5.4 Conclusion . . . 45

6 Appendices **46**

6.1 Hardware Schematic . . . 47
6.2 Arduino Code . . . 48
6.3 Max Code . . . 56
6.4 Survey Results . . . 59

List of Figures

1 GloveSense Dataglove 1

2.1 Initial Hardware Block Diagram 14
2.2 Initial Software Block Diagram 15

3.1 T9 Predictive Text 18
3.2 Dvorak Simplified Keyboard for left hand 18
3.3 Enable Talk Dataglove [Ena] 19
3.4 NED Glove used by UFRN Team 20

4.1 Flex Sensor Circuit Diagram 27
4.2 Flex Sensor Block Diagram 28
4.3 IMU Block Diagram 29
4.4 Bluetooth Block Diagram 31
4.5 Forearm Component Case 33
4.6 Max Serial Object Implementation 34
4.7 Gesture Recognition State Diagram 36
4.8 GUI with Gesture Map 37
4.9 MIDI Map Mode in Ableton Live 38
4.10 All MIDI Map in Ableton Live 39

5.1 Mi.Mu Dataglove [Perner-Wilson] 41

6.1 Hardware Schematic 47
6.2 Main Max Code Modules 56
6.3 *Datain* Max Code Module 57
6.4 Gesture Recognition Max Code Module 58
6.5 Survey Results, Glove Interest 59
6.6 Survey Results, Prior Use 59
6.7 Survey Results, Music Application 59
6.8 Survey Response, Concert Application 60
6.9 Survey Response, Gathering Application 60
6.10 Survey Results, Feedback Device 60
6.11 Survey Results, Cost 61

List of Tables

3.1 Dataglove Comparison 21
3.2 Quaternion Multiplication 23
4.1 Flex Sensor Pin Assignment 28
5.1 Gesture Accuracy 44

Chapter 1

Introduction

Human-Computer Interaction (HCI) is a growing field of study of the interface between a human user and computers. There are many different methods of HCI seen in modern devices, such as a cell phone touch screen, or a computer keyboard. In music, HCI devices are are limited to a few, very similar devices such as knobs and sliders. The input devices currently in use enable the user to interact with an object, rather than directly detecting human movement. This is primarily because these objects are more consistent in their movements, which contributes to a higher accuracy. Because of the fear of inaccuracy, there are not many datagloves in use in any industry. The music industry is a potential application of a dataglove where accuracy is not as paramount as user experience. Music may be the application best suited to dataglove use.

The purpose of this project is to create a dataglove that detects various human movements and interprets them into MIDI data. The dataglove creates a new, more interactive way for musicians to perform at a concert. This sort of wearable device will help performers by providing a more fluid connection with the music they make. The dataglove will assist the artist's individuality, since every artist has their own unique way of reacting to the music they play and a wearable device allows the artist modify music with muscle movements. Therefore, this technology will allow artists to be connected with their music on a more

advanced level.

The team conducted a survey at Worcester Polytechnic Institute (WPI) among students and a few faculty members, seen in appendix 6.4. The survey group consisted of 47 participants with diverse backgrounds, yielding representative results. Additionally, the team conducted personal interviews with two musicians and two music professors, which gave a more focused response than the survey. Dr. Manzo, one of the interviewees, suggested that the team create a use case that would enable an artist to do what may be possible with a traditional instrument or device, but is made easier through the use of an alternate controller.

The team designed towards a specific use case, in the knowledge that the final design could be used in many different applications with little to no modification of the basic hardware and software design. To best exemplify the capabilities of GloveSense, the team chose a live performance emulating a DJ a turntable mixing board as the use case. The typical turntable mixing board is capable of playing two tracks at once, and controlling different aspects of each track simultaneously.

To transition between the two songs, a performing DJ will use tempo controls to match the tempos of the song currently playing and the next song to play. This is known as "beatmatching." Once the tracks are beatmatched, the DJ can then use the track-to-track fade to simultaneously reduce the volume of the old song and increase the volume of the new song. An especially astute DJ will constantly modify the equalizer values of each track according to the response of the audience. The DJ can also add in audio effects, such as flanging or echo, and play audio samples from the sample bank.

The team chose five effects to emulate for this use case, to demonstrate the potential of the glove while still being easy for the user to learn. Five effects the user can map the glove gestures to are listed below:

- Tempo Control
- Track-to-track Fade

- Three-band Equalizer
- Audio Effect Modulator
- Sample Bank

The team believes GloveSense can emulate these effects of a DJ turntable mixing board with enough accuracy to be usable in a live performance setting.

The results of an interest survey and accompanying derived specifications are seen in Chapter 2, Customer Requirements. The information presented in Chapter 3, Literature Review, provides the necessary background to understand the system design presented in Chapter 2 and the specific Methodology seen in Chapter 4. The results the team gathered from this project are seen in Chapter 5, Results. Finally, the team presents concluding statements in Chapter 5.4.

Chapter 2

Customer Requirements

The survey initially described in chapter 1 assessed the interest and the experience that participants had with similar devices. Among all the people surveyed only a small portion, nine percent, were experienced with using similar wearable devices. Out of all the people surveyed, 91 percent of the participants were interested in an interactive controller glove, and 96 percent of the participants stated they would like to see these devices being used by famous musicians at large concerts. The survey data shows a generally positive response to the idea of a glove, as well as a willingness to pay for the described product. This shows that there is interest, and a niche in the market for the glove.

The team interviewed Dr. Manzo and Dr. Bianchi, who provided valuable insight into the use of alternate controllers for musical applications. The interviewees both suggested the team use the MIDI format, as it is the industry standard. Dr. Manzo suggested targeting the glove for use with disabled persons, specifically music for the deaf. The team felt that this could be a good alternate use for the glove, but decided to design for the average person, as that would create the largest possible audience.

The team derived a number of requirements from the survey results, as well as other feedback given by the survey takers. The team added to these requirements with ideas from the interviews described above. The lists of implicit and explicit requirements are seen below:

Implicit Requirements:

- The glove should be easy to use.
- The glove should be comfortable.
- The glove should be usable by a wide range of people.
- The glove should be safe to use in an appropriate environment.

Explicit Requirements:

- The glove should accurately recognize gestures.
- The glove should be system (Windows, Mac) independent.
- The glove should allow the user to create or modulate music.

These requirements are fairly broad, and will allow the team room for development. The team condensed these requirements into a the product design specifications discussed in section 2.1.

2.1 Derived Specifications

The team created a design to conform to the customer requirements discussed in Chapter 2. Each specification was designed for individually, but the team attempted to maximize overlap between the requirements to minimize the potential design aspects. A simple design is not only easier for the team to implement, it will likely be easier for the end user to enjoy as well. The specifications are seen below:

- The glove will allow the user to create or modulate music.
- The glove will use the MIDI standard and software compatible with both Windows OS and Max OSX.

- The glove will be comfortable and safe to use.
- Gestures detected by the glove will be sensible and memorable.
- Gesture detection will be user independent.
- Gestures will only be detected when the user wants.

The first specification is the most broad of them all: "the glove will allow the user to create or modulate music." This specification must be adhered to throughout the project; it is the primary goal of the team. The glove must therefore be interfaced with a synthesizer, soundboard, or computer program. Since this goal is so broad, the team chose a specific use case, described in Chapter 1, to design to. This allowed the team to design hardware and software capable of completing this overall goal, while still having a demonstrable software goal to achieve. The team will describe the choices made to adhere to this design specification throughout Chapter 4.

The second specification is reliant on both hardware and software: "the glove will use the MIDI standard and software compatible with both Windows and Mac OSX." This specification is derived from the system independence customer requirement. It necessitates the use of communication hardware and processing software that are compatible with a majority of most computers today. The team will describe the choices made to adhere to this design specification throughout Chapter 4.

The third specification is primarily reliant on hardware: "the glove will be comfortable and safe to use." This specification is derived from the comfort and safety customer requirements. It requires consideration of safety and comfort in all aspects of the hardware design. The team will discuss the choices made to adhere to this design specification in Chapter 4, throughout section 4.1.

The fourth specification is reliant on both hardware and software: "the gestures detected by the glove should be sensible and memorable." This specification is primarily derived from the ease of use customer requirement. It will dictate the choice of motion detection hardware

employed in the glove, as well as the gestures that the team selects for use from the pool of gestures available due to the hardware selection. The team will discuss the choices made to adhere to this design specification throughout section 4.1.

The fifth specification is primarily reliant on software: "the gesture detection should be user independent." This specification is derived from the ease of use and user independence customer requirements. It requires the use of some form of user calibration, whether it be a learning program or literal calibration, through software and hardware, respectively. The team will discuss the choices made to adhere to this design specification in Chapter 4, primarily in section 4.2.1.

The sixth and final requirement is reliant on software, and possibly reliant on hardware: "gestures should only be recognized when the user wants." This specification is derived from the ease of use customer requirement. It necessitates that the glove have some sort of activation feature, whether it is a hardware switch or something detected by software. The team will discuss the choices made to adhere to this design specification in Chapter 4, primarily in section 4.2.1.

The team then designed hardware and software block diagrams that would fulfill the specifications listed above. These diagrams are seen in figures 2.1 and 2.2, respectively.

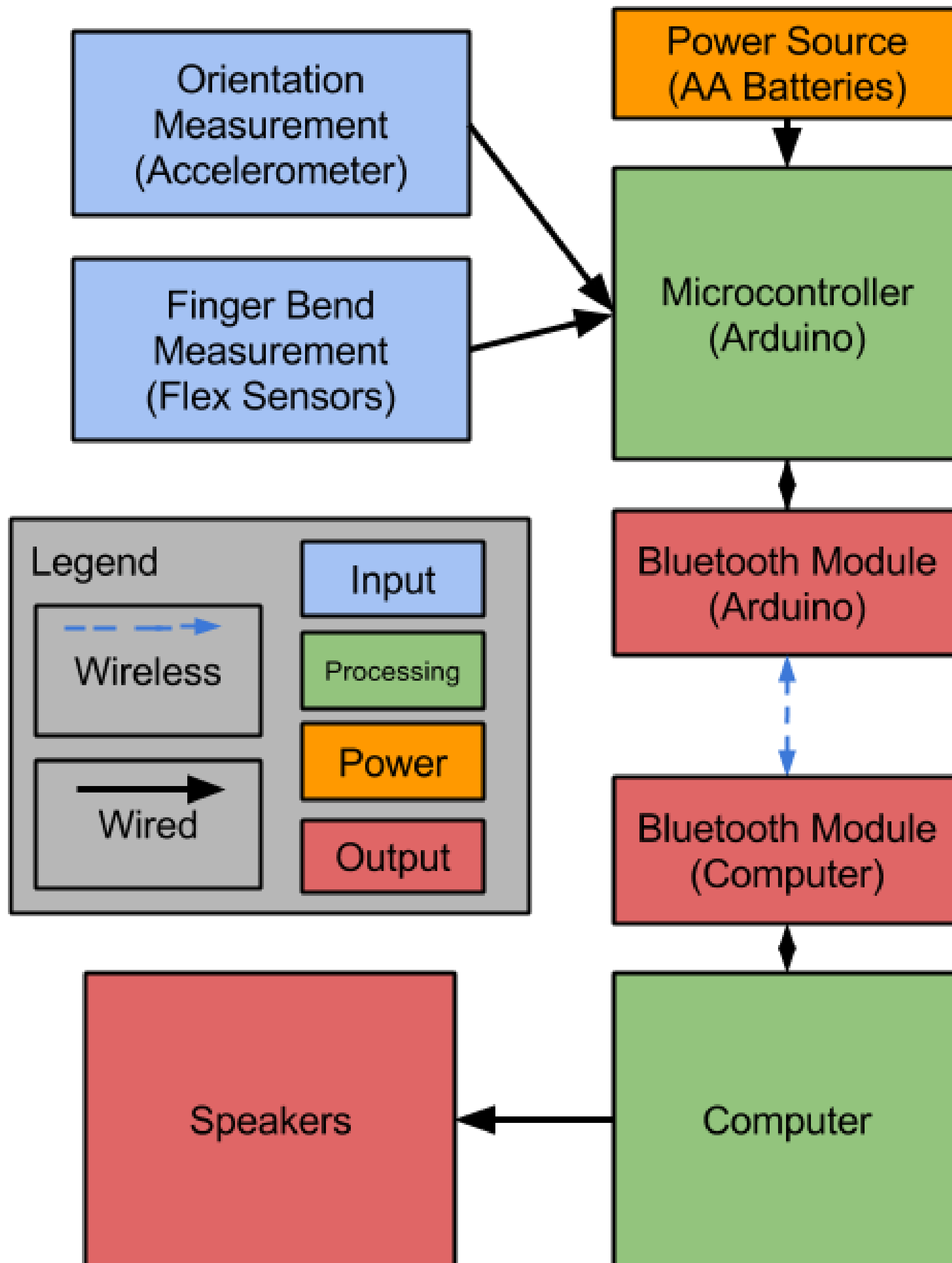


Figure 2.1: Initial Hardware Block Diagram

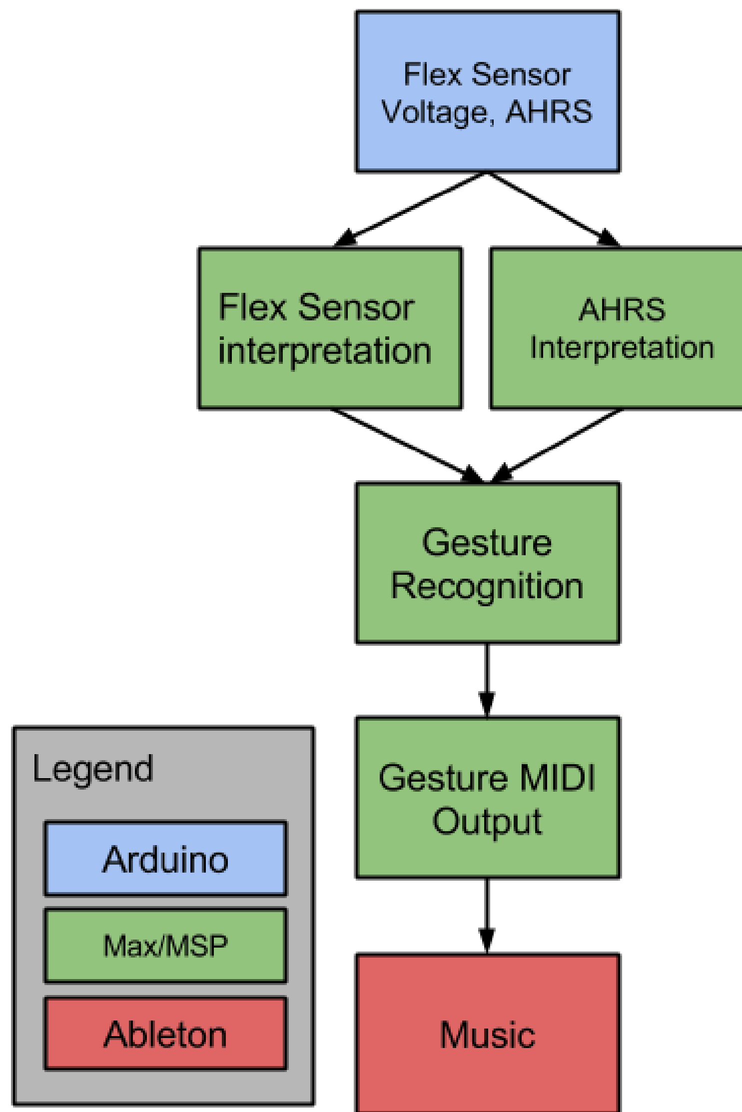


Figure 2.2: Initial Software Block Diagram

Chapter 3

Literature Review

This Chapter presents the relevant background research to build a frame of reference for this project. In section 3.1, we discuss other dataglove projects, and the intended audience of those projects. In section 3.2, we discuss the Musical Instrument Digital Interface (MIDI) communication standard. In section 3.3, we discuss the Max programming family, and many musical uses of the language. In section 3.4, we discuss Attitude and Heading Reference Systems (AHRS) and their use in orientation and position measurement.

3.1 Prior Art

There are many existing datagloves on the market at this time. Each dataglove is tailored to a specific application, and is held a high price point, thus limiting the target market of the glove. One such glove, the EnableTalk sign language interpretation glove, is highly specialized in gesture recognition, and outputs text-to-speech interpretation of the user's sign language inputs. This glove fills a very specific niche in the dataglove market. However, there are many traits that this glove has that are useful to other datagloves. With such exceptional prior art to build off of, our team hopes to include as many of the positive aspects as possible, and as few of the negative.

In the past, engineers have tried many different implementations of single-handed input

devices in an attempt to increase ease of use of computers. These attempts range from the T9 predictive phone keyboard to the Dvorak Simplified keyboard for one hand. These are displayed in figures 3.1 and 3.2, respectively. Both of these options suffer from a lack of input information that one hand is able to provide. Ty ping with one hand is simply slower and less accurate than typing with two. For intensive, high-accuracy typing applications, a full keyboard is still most applicable. A team from Cornell University created a dataglove, dubbed "Mr. Gloves," in another attempt to replace the keyboard and mouse as input devices into a computer. Mr. Gloves accomplishes computer input by using a combination of gestures and on-board buttons to create a tenkeyless keyboard with one hand, and a mouse with the other. The intended use of this device is in "computer applications that are not particularly keystroke-intensive, such as surfing the web and playing video games"[Chen and Levine, 2010]. The pair of gloves have a single flex sensor on the right hand, as well as two accelerometers and a number of buttons. The buttons–primarily mounted on the left hand along the proximal and metacarpal joints–and accelerometers serve as the keyboard. Each unique input has an associated hand position, measured by the accelerometers, and button. This format of input is inefficient and necessitates a large number of movements to type a simple string of characters. The right hand simulates mouse input using the accelerometers for 2-D positioning, as well as the flex sensor and a button to simulate left and right clicks, respectively. This data is transferred from the gloves, to the wireless base station, and then through USB to the PC. The PC interprets this data using the USB standard, interpreting the output as a Human Interface Device (HID). The team managed to make the gloves compliant with Windows HID using no additional drivers. Mr. Gloves is an interesting interpretation of an existing HID. It accomplishes its task, but arguably not as well as the existing mouse and keyboard. Mr. Gloves may not be the best HID, but the team can learn from the negative aspects of the design, and incorporate positive aspects, such as wireless capability.

Figure 3.1: T9 Predictive Text

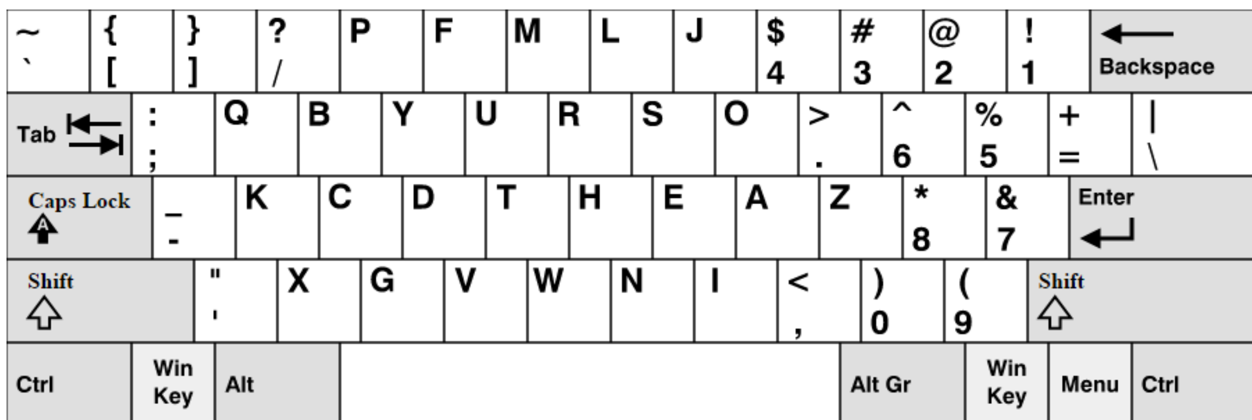


Figure 3.2: Dvorak Simplified Keyboard for left hand

The EnableTalk glove won the the 2012 Microsoft Imagine Cup in Sydney, Australia [Ena]. Unfortunately, the glove is not publicly documented well, but there are pictures of the prototype used in the competition. These pictures show a solar charging panel, as well as flex sensors and contact sensors. The glove uses an unknown software on a mobile phone to convert sign-language gestures into speech. The EnableTalk glove has many desirable features, namely standalone capability and multiple gesture recognition. These features are necessary in sign language interpretation, but also are quite applicable to a musical application.

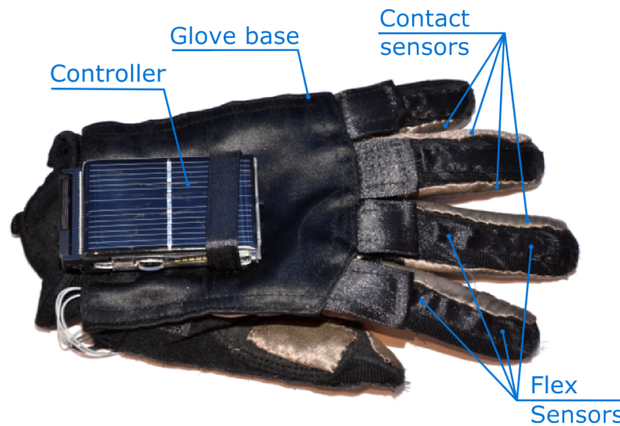


Figure 3.3: Enable Talk Dataglove [Ena]

A team at the Federal University of Rio Grande do Norte (UFRN), created a data-glove using the Arduino platform for virtual reality applications. This glove, dubbed the "NED glove," is seen in figure 3.4. The team discusses the programming languages they used, namely C++, as well as the implementation of on-glove positive user feedback, including physical proprioceptive and auditory feedback. The glove uses a tethered design, communicating with and drawing power from a computer USB connection. This limits the functionality of the glove in many practical applications. The data passed to the computer had a high amount of noise, which the team eliminated by averaging the five most recent sensor readings. This does incur a small delay, but the UFRN team believed that this did not negatively impact the fluidity and usability of the glove. The processed data was used

to create a simulation of the hand in the Unity game engine. The simulated hand mirrored movement of the dataglove, but did not allow the user to interact with objects in the Unity engine, as there was an issue with mesh collisions in the simulator. The methods the UFRN team employed to reduce measurement noise and to provide feedback are very useful in designing a state of the art dataglove.

Figure 3.4: NED Glove used by UFRN Team

[Dantas et al., 2013]

The datagloves discussed in this section provide a good idea of what is necessary to create a successful glove. A comparison of the reviewed datagloves can be seen in table 3.1. The different aspects of each glove will be considered when manufacturing the GloveSense dataglove.

	EnableTalk	Mister Gloves	NED Glove
Wireless	✓	✓	×
Bend Sensors	?	1	5
Position Sensors	?	Accelerometer	Accelerometer
Method of Connection	Bluetooth	FHSS Wireless	USB
Power Supply	Solar/battery	AAA battery	USB

Table 3.1: Dataglove Comparison

3.2 MIDI - Musical Instrument Digital Interface

Musical Instrument Digital Interface is the industry standard for digitization of musical data, including pitch, velocity, and note modifiers [Swift, May 1997]. The simplest form of MIDI assigns each note a pitch value and a velocity (or volume) value, each an unsigned 7-bit integer ranging from 0-127. Furthermore, the MIDI protocol is separated into two types of information, status bytes and data bytes. The status byte will carry commands from the MIDI controller, such as note-on or note-off, while the data byte will carry parameters, such as pitch and velocity. To make the two distinguishable, the first bit of a status byte is 1 and the first bit of a data byte is a 0, similar to a signed integer [Huber, 1991]. This allows each note to be coded in two bytes of data. These simple note bytes are often associated with a more complex string of data that encodes the note's timbre and sustain as well. Changing these values allows for easy manipulation of the note sound.

Many instruments, analog or synthesized, have the ability to be patched through a MIDI device to modify the sound prior to amplification. In addition to note control, many digital audio workstations allow for MIDI controllers to be mapped to certain parameters, such as filter cutoff frequency, or a signal modulator [Abl]. This allows performing artists to modify their sound in real time.

3.3 Max/MSP/Jitter

Max/MSP/Jitter is a visual programming suite, developed by Cycling '74 [C74]. Max is the primary module of the program, and contains the majority of the functions necessary for its use. MSP, the acronym for Max Signal Processing, is an addition to the Max library consisting mainly of signal processing objects. Finally, Jitter is the video processing add-on to Max. The pseudonym 'Max' may be used to refer to the entire programming suite Max/MSP/Jitter.

The Max programming language is comprised of a library of "objects," which are in reality standalone functions that can take input, give output, or both. These objects are connected using "patch chords," which transfer six different types of messages between objects. These message types are: int, float, list, symbol, bang, and signal. All these aspects make up the basis of the Max syntax, which is quite simple for the user to program with. The visual nature of the Max language is also used as the end-user Graphical User Interface (GUI), where different objects are displayed, such as a signal plot or button. This allows the user to build a visually appealing GUI while coding.

One of the objects in Max, *ctlout*, allows the user to create a MIDI device from data in Max. This data can be of type int, float or list. Max supports creation of multiple MIDI devices in this manner, which can be mapped to numerous MIDI ports. Similarly, the *serial* object allows Max to take in serial data via USB or Bluetooth, and can interpret this information using other built-in objects. Thus, Max can allow any system capable of serial data transfer to be used as a MIDI controller.

Max allows the user to do any number of things, ranging from simple signal modulation, to coordinating music and lights through a MIDI interface. Many different devices can be interpreted by the software into a MIDI controller.

3.4 Attitude and Heading Reference System

An attitude and heading reference system consists of sensors on three axes and interpretation software that can measure heading, roll, and pitch. On each axis, there is an accelerometer, gyroscope, and magnetometer to measure acceleration, rotation, and direction, respectively. The key difference between an AHRS and an inertial measurement unit, which consists of the same sensors, is that an AHRS provides attitude and heading information rather than just sensor data[Various].

When interpreting sensor data, AHRS often use quaternions, an extension of the complex number system, to describe rotation in three-dimensional space. As seen in table 3.2, multiplication of quaternions is non-commutative. According to Euler's rotation theorem, any rotation of a body about a fixed point is equivalent to a single rotation about a fixed axis that runs through the fixed point. Therefore, any rotation can be represented by a position vector and a angle scalar. Any rotation about three-dimensional vector u, as seen in equation 3.1, can be represented by the quaternion q, as seen in equation 3.2. Quaternions allow orientation information to be stored in four quaternion numbers, Θ, u_x, u_y, and u_z[Hamilton].

Multiplier \ Multiplicand	**1**	**i**	**j**	**k**
1	1	i	j	k
i	i	-1	k	-j
j	j	-k	-1	i
k	k	j	-i	-1

Table 3.2: Quaternion Multiplication

$$\vec{u} = (u_x, u_y, u_z) = u_x\mathbf{i} + u_y\mathbf{j} + u_z\mathbf{k} \tag{3.1}$$

$$\mathbf{q} = e^{\frac{\Theta}{2}(u_x\mathbf{i}+u_y\mathbf{j}+u_z\mathbf{k})} = cos\frac{\Theta}{2} + (u_x\mathbf{i} + u_y\mathbf{j} + u_z\mathbf{k})sin\frac{\Theta}{2} \tag{3.2}$$

The final output of the AHRS is three angles, with respect to a reference axis, and three accelerations, one for each axis. The reference axis most commonly used is magnetic north, which stays approximately constant for small movements. In some cases, gravity can also be used as the reference axis, but acceleration in the spacial z-axis will detract from the accuracy of this method. It is possible to use double integration of the acceleration values to calculate an approximate position in three-dimensional space, however there is drift associated with this method. An AHRS using quaternions is more stable and less complex than most other methods of representing orientation and rotation, which is extremely useful when attempting to implement an AHRS on a low-cost microprocessor.

3.5 Summary

This chapter has displayed some of the prior art and knowledge that have been used throughout the project. These background topics provide valuable insight into the advantages and disadvantages of numerous approaches to different aspects of this project.

Chapter 4

Methodology

This Chapter presents the methods the team used when designing the glove and accompanying software. Using the information in Chapter 3, the team created a prototype that conforms to the design specifications seen in Chapter 2. In section 4.1, we discuss different modules of the Arduino hardware and software design. In section 4.2, we discuss different modules of the Max software design. Finally, in section 4.3, we summarize all the modules discussed in this Chapter.

4.1 Arudino Development

The team constructed a very simple prototype near the beginning of the project, consisting of an Arduino Uno with two flex sensors connected to it. The Arduino sent the readings from the flex sensor over USB as serial data to a Max patch using the same process as in the final design, which will be described below. Once this basic functionality was set up, the team began planning how to add the additional flex sensors, accelerometer, and Bluetooth. Even in the simplest prototype possible many of the hardware design modules, seen in figure 2.1, are represented in some minimum form.

For the initial prototype, all modules used were already functioning, standard implementations. These include: Arduino microcontroller; power source and communication method,

both USB; and the flex sensor circuitry. This allowed the team to rapidly prototype software, alongside hardware development.

Once the team ensured software was able to retrieve the data, a new prototype could be developed. This prototype included all of the modules outlined in 2.1. The team chose to use an Arduino Mini as the microcontroller to receive and process the information coming from the sensors. At first the team used the Arduino UNO for this purpose, however, the Arduino UNO only has six analog pins, while the design required seven, five for the flex sensor voltage readings and two for the accelerometer I^2C bus. The Arduino Mini has eight analog pins, thereby complying with the hardware design. In addition, the Arduino Mini is physically small at only 30mm by 15mm, so it will easily be able to fit on the forearm. The team decided not to create a custom printed circuit board with microcontroller for the final prototype, since the Arduino is a well established platform that can be reprogrammed. Additionally, the team wanted to allow the end user freedom to modify the glove software, instead of limiting the user to our design. Programming is much easier with the Arduino IDE, as opposed to programming the Atmega 328 directly. Due to these reasons the team decided to use an Arduino Mini for the final prototype.

The team initially used USB as the Arduino power source. The Arduino can then distribute power to the hardware blocks. The USB power was only used in the initial prototyping phase of development. As the design specifications require the glove to be wireless, the design must employ a battery power solution. Initially, the team used six AA batteries wired in series to provide power to the Arduino. The team found this to be difficult to mount on the forearm, and decided to use a 9V 600mAh battery to power the microcontroller instead. The team found the Arduino Mini and connected hardware only draws about 50mA of DC current, allowing a performer to use the glove for approximately 12 hours on one 9V battery. This battery can theoretically power the glove for much longer than most performing artists require, and if another solution is desired, the team left room in the hardware box for such expansion.

This design allowed for rapid prototyping of other hardware modules to be used in the final prototype. A full schematic of the final design is seen in appendix 6.1. The modules discussed below are employed in the final hardware design, unless otherwise specified.

4.1.1 Flex Sensors

The flex sensors used in the initial prototype included a voltage divider circuit that allowed for a three pin format. This allowed for reliable, plug-and-play use on the Arduino board, which can both supply voltage to and read voltage from this flex sensor circuit. However, these pre-made flex sensors were expensive, and to reduce cost, the team opted to use a simple sensor and employ self designed circuitry. The circuit designed to read the sensor is seen in figure 4.1, and the general data flow block diagram is seen in 4.2. The team designed a voltage divider circuit that allowed for maximum possible accuracy in measurement, as well as a low power draw of approximately $165\mu W$ per sensor.

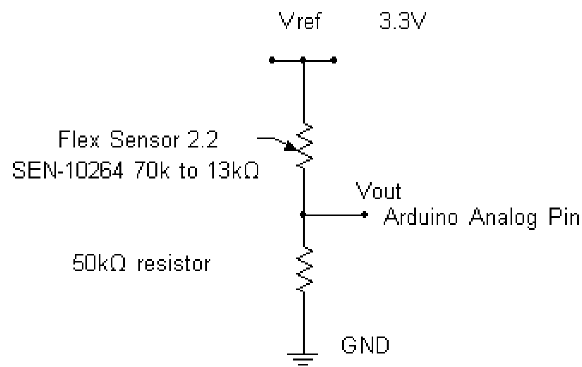


Figure 4.1: Flex Sensor Circuit Diagram

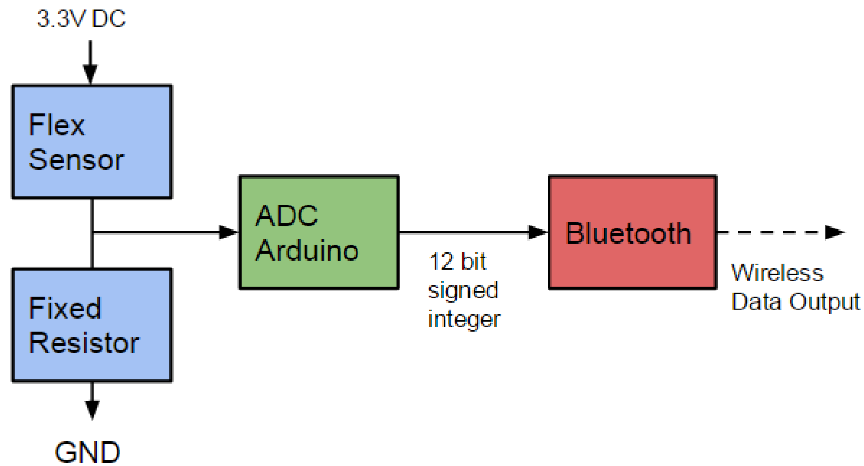


Figure 4.2: Flex Sensor Block Diagram

The team connected the data wires of the flex sensors to analog pins as seen in table 4.1. At the beginning of the main loop, the *analogRead* function takes a voltage reading at each analog pin. This reading is translated into an integer by the Arduino signed 12-bit ADC, with a value of $5V$ equal to 1023, and $0V$ equal to 0. These integers are then sent over Bluetooth as serial data according to section 4.1.3, where they can be interpreted in Max as seen in section 4.2.

Finger	Number	Analog Pin
Little	4	A0
Ring	3	A1
Middle	2	A2
Index	1	A3
Thumb	0	A7

Table 4.1: Flex Sensor Pin Assignment

4.1.2 Orientation Measurement

The team decided to use the MPU-9150, an inertial measurement unit, in the dataglove. It is stated to be the world's first integrated 9-axis MotionTracking™ device that combines a 3-axis gyroscope, a 3-axis accelerometer, a 3-axis magnetometer and a Digital Motion Processor™ (DMP™) hardware accelerator engine[Spa]. Three 16 bit Analog to Digital Converters (ADCs) are used for digitizing the outputs of accelerometer and gyroscope and three 13 bit ADCs are used for the magnetometer outputs. Accelerometer, gyroscope and magnetometer are all user programmable for precision tracking of both fast and slow motions. Therefore, this product will provide us with the high precision and low cost in the prototype design.

The accelerometer is be directly mounted on the glove and connected to the Arduino. The accelerometer pins VCC, GND, SDA, SCL, and INT are connected to the Arduino pins $5V$, GND, $A4$, $A5$, and 2, respectively. The team setup the IMU in the Arduino code using the $mpu.setDLPFMode()$ command, using the default setting of 4. Then, communication was started by using the $mpu.setIntDataReadyEnabled(true)$ command.

The general data flow block diagram is seen in 4.3.

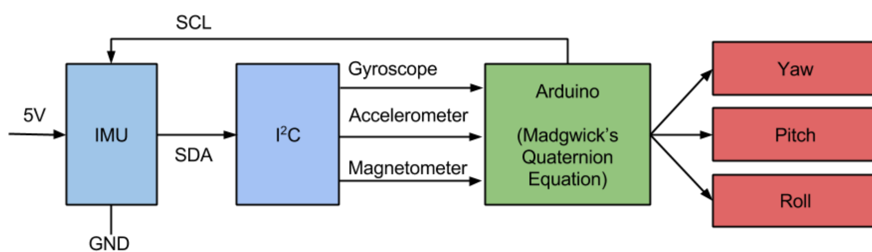


Figure 4.3: IMU Block Diagram

To interpret the IMU sensor output, the team implemented an AHRS on the Arduino. The team used open-source beerware-licensed code provided by Sparkfun, the IMU manufacturer[Spa]. The code sets up a polling system such that the calculations were only carried

out if $mpu.getIntDataReadyStatus()$ was set to 1. The raw acceleration, rotation and magnometer values are calculated according to equations 4.1 to 4.3. The values $q[0]$ to $q[3]$ utilized in equation 4.4 are calculated using Madgwick quaternion mathematics, as seen in appendix 6.2, in the function $MadgwickQuaternionUpdate()$. The interpreted sensor output yields acceleration (mg_n), rotation (deg/s) and magnetometer readings (mG) for each axis, as well as quaternion values. The team only utilized the roll of the IMU, as rotation is the easiest wrist movement. This integer is then sent over Bluetooth as serial data according to section 4.1.3, where they can be interpreted in Max as seen in section 4.2.

$$A = a_{in} * 2.0f/32768.0f \tag{4.1}$$

Accelerometer Scaling

$$G = g_{in} * 250.0f/32768.0f \tag{4.2}$$

Gyroscope Scaling

$$M = m_{in} * 10.0f * 1229.0f/4096.0f + calibrationOffset \tag{4.3}$$

Magnometer Scaling

$$roll = atan2(2.0f * (q[0] * q[1] + q[2] * q[3]), q[0] * q[0] - q[1] * q[1] - q[2] * q[2] + q[3] * q[3]) \tag{4.4}$$

Roll Interpretation

4.1.3 Bluetooth Serial Communication

The team used the BlueSMiRF module for the Arduino in the final prototype, because it was easy to implement, widely used and well documented. The module allows for short distance wireless communication with a computer or other device. The team found the range to be nearly $20m$, and could communicate through interior building walls. In future iterations, if a higher range is desired, it would be simple to implement another module, since all Bluetooth modules conform to a standard input/output (I/O) setup. Some modern computers have a Bluetooth device built-in, otherwise a small USB dongle is required for communication. The block diagram showing the connection of the Bluetooth module is seen in figure 4.4.

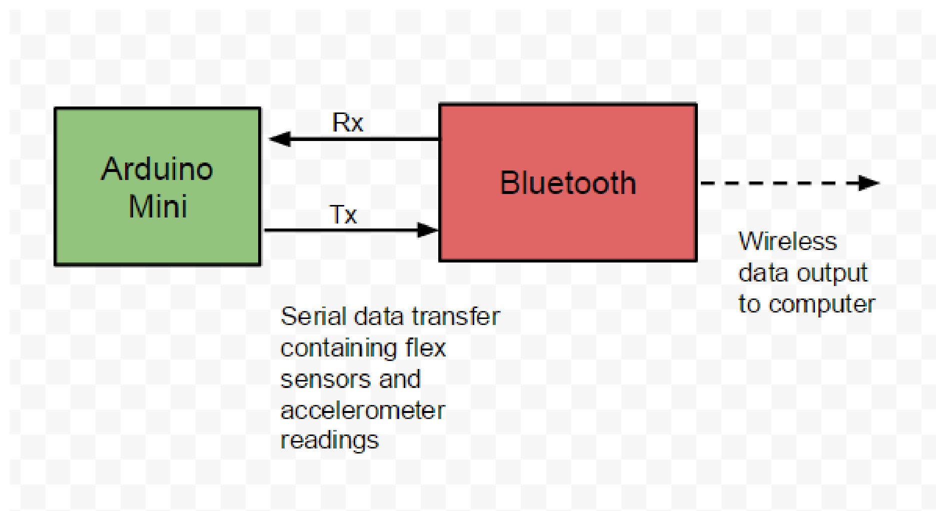


Figure 4.4: Bluetooth Block Diagram

The Bluetooth module has a number of I/O pins: $5V$, GND, RX, and TX. The team connected these to the Arduino V_{CC}, GND, 3, and 4, respectively. The Bluetooth module can then pass the data to a computer wirelessly. The computer Bluetooth module receives the transmission and converts it back into serial data. The software discussed in section 4.2 can then interpret the data.

The team used the software serial library, which has the ability to create another serial port. The team created the *bluetooth.print*() function by defining a new serial port using

softwareserial to assign the digital pins of the Bluetooth module to the Tx and Rx functions. The team sent three $ symbols at the default baud rate of 115200 to initialize the Bluetooth module. Then, the team redefined the baud rate to 9600, allowing for standard communication between the Arduino and Max.

At the end of the main loop, the team sent the sensor data gathered according to sections 4.1.1 and 4.1.2 in specific message formats. The flex sensor integers were sent in the format $finger\ X\ value$, where X is the finger number as seen in table 4.1 and $value$ is the integer voltage reading. The roll data was formatted $roll : value$ where $value$ is the floating point roll value from the AHRS output.

4.1.4 3D-Printed Case

The team created a case which attaches to the users forearm to enclose the electrical components not housed on the glove, namely the Arduino mini, Bluetooth module, the battery and the voltage regulator. The case attaches to the forearm with Velcro bands. The case increases the durability of the prototype by protecting the electrical components and gives the prototype a desirable compact look.

The team used Solidworks in order to design the box. The box lid was designed like a friction fit slide in style because it was the most forgiving way in terms of dealing with the ± 0.3mm error of the 3D printer. The case and components are seen in figure 4.5. The case is comfortable and lightweight, and keeps the components from interfering with wrist and finger movements, thereby complying with the design specifications.

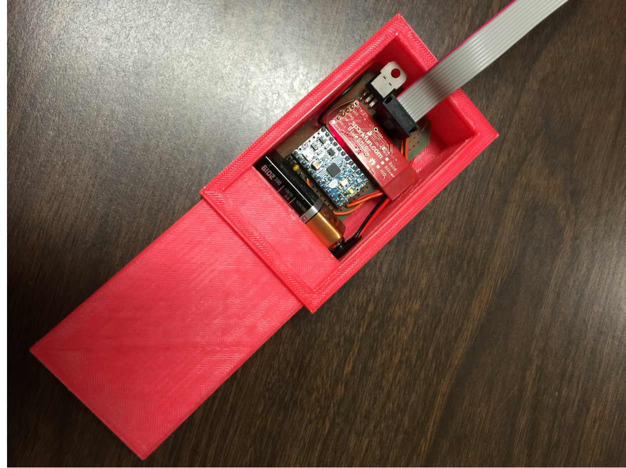


Figure 4.5: Forearm Component Case

4.2 Max Software

The data retrieved from the Arduino through Bluetooth is interpreted by Max as serial data. The serial data from the flex sensors and accelerometer can be read by Max using the *serial* object, as seen in figure 4.6. The *serial* object takes in a number of inputs, most importantly the serial port information. The team set the serial baud rate to match the Bluetooth baud rate of 9600. The port information changes depending on the Bluetooth connectivity, and requires the user of the program to select the proper port on start up. Since the serial object in Max outputs the information it receives as integers representing ASCII characters, these integers needed to be converted to the actual numerical output using the *itoa* object. The integer values are then interpreted into MIDI values in the *datain* patch, as seen in figure 6.3.

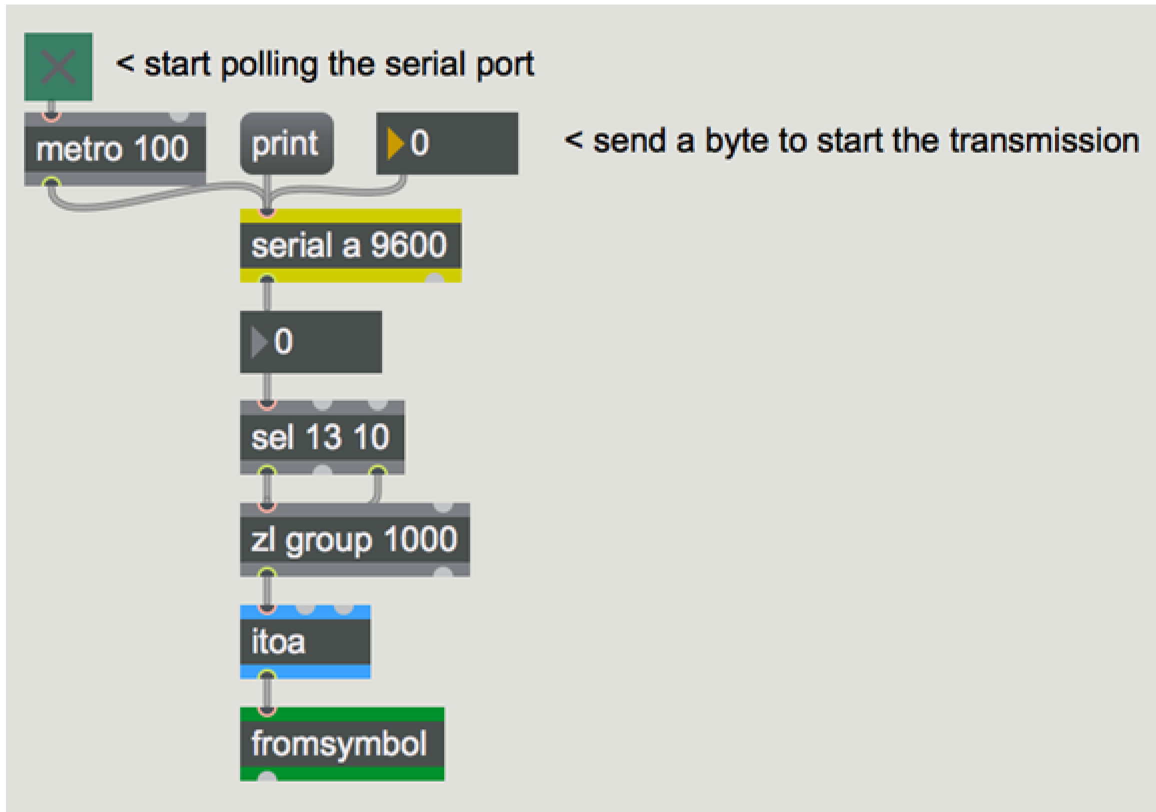


Figure 4.6: Max Serial Object Implementation

4.2.1 Flex Sensor Interpretation and Gesture Recognition

The team routed the flex sensor readings, in the form *finger X value* to the appropriate section of code using the *route* object, with the argument *finger*. This object passes all messages that begin with *finger* to the left output, and all other messages to the right output. This allows *roll* : messages to pass to the right output unhindered. Then the message is then split into finger number and finger value using the *unpack* object. The finger number is used to select an output of the *gate* object that will route the finger value to the proper finger interpretation module. Each finger is interpreted in the same manner.

The team created a calibration system for the flex sensors by scaling the sensor values to MIDI values (0-127) and using the *peak* and *trough* objects in conjunction with the *scale* object. Upon start up, the patch sets arbitrary values for the minimum and maximum of the

scale object, then continually changes these values based upon the smallest and largest input values seen, using the *peak* and *trough* objects. This allows the software to continuously calibrate finger ranges to the user movements.

To interpret these MIDI values into gestures, a second patch was created which recognizes gestures based on each finger being in a specific range. A flex sensor value of 127 corresponds to a finger fully bent, and 0 corresponds to a finger fully open. If all fingers were in a range near 127, the patch would recognize that as a fist. If all fingers were in a range near 0, that would be interpreted as an open hand. Once basic gestures like this were set up, adding new gestures only required finding the range of flex sensor values for each finger when making that gesture. The full gesture recognition code is seen in figure 6.4. This necessitates finding reasonably orthogonal gestures, primarily treating the individual fingers as binary values. Having two large ranges of detection spread over the whole range of motion of the sensor allows for simple detection of the different gestures by creating a very small range where an interpretation error could occur. Once a gesture is detected, any values associated with it can be modified by motion data from the accelerometer.

The team created a state diagram, seen in figure 4.7, for the possible states that the glove could be in, which correspond to the states of waiting for the gestures in figure 4.8. The states are "in between" the levels of the gesture menu, which allows for a simplification of the menu design. The initial state, $S0$, simply waits for the user to make the activation gesture. The activation gesture allows the user to move freely in most situations, while not requiring a hardware switch to control gesture recognition. Once that has been made, $S1$ waits for the user to select a track, sending the glove to state $S2$ or $S3$, or modify all tracks, upon which the glove cycles back to state $S1$. $S2$ and $S3$ correspond to track one and two, respectively. These two states are waiting for the same set of gestures, but will modify only one track. Once one of the modification gestures is made, the glove moves to the appropriate state $S4$ or $S5$. $S4$ waits for any of the final gestures, and will play the sample that is mapped to that gesture through Ableton. Finally, $S5$ waits for orientation input to modify the value

that the prior gesture corresponds to.

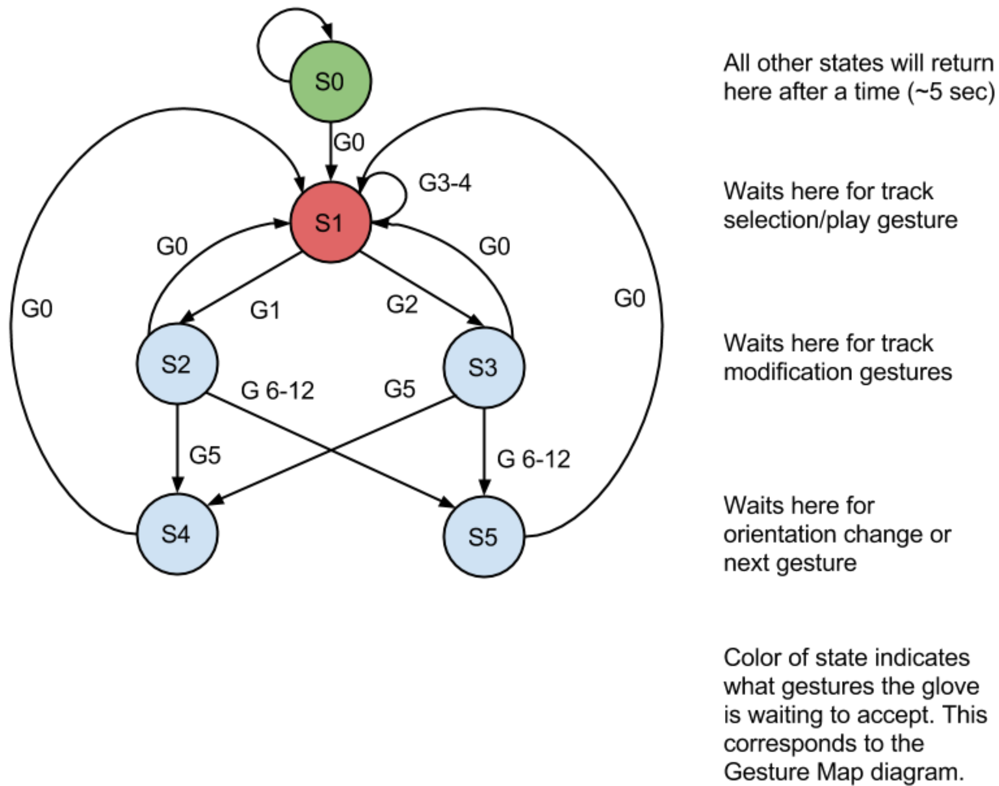


Figure 4.7: Gesture Recognition State Diagram

This design creates a menu system, allowing for a small pool of gestures to have many effects. A system of this sort is easily memorable, and quickly navigated. The team believes that it is an effective manner of modulating music through the use of a dataglove.

The gestures the team chose to recognize were mapped to the different outputs as seen in figure 4.8.

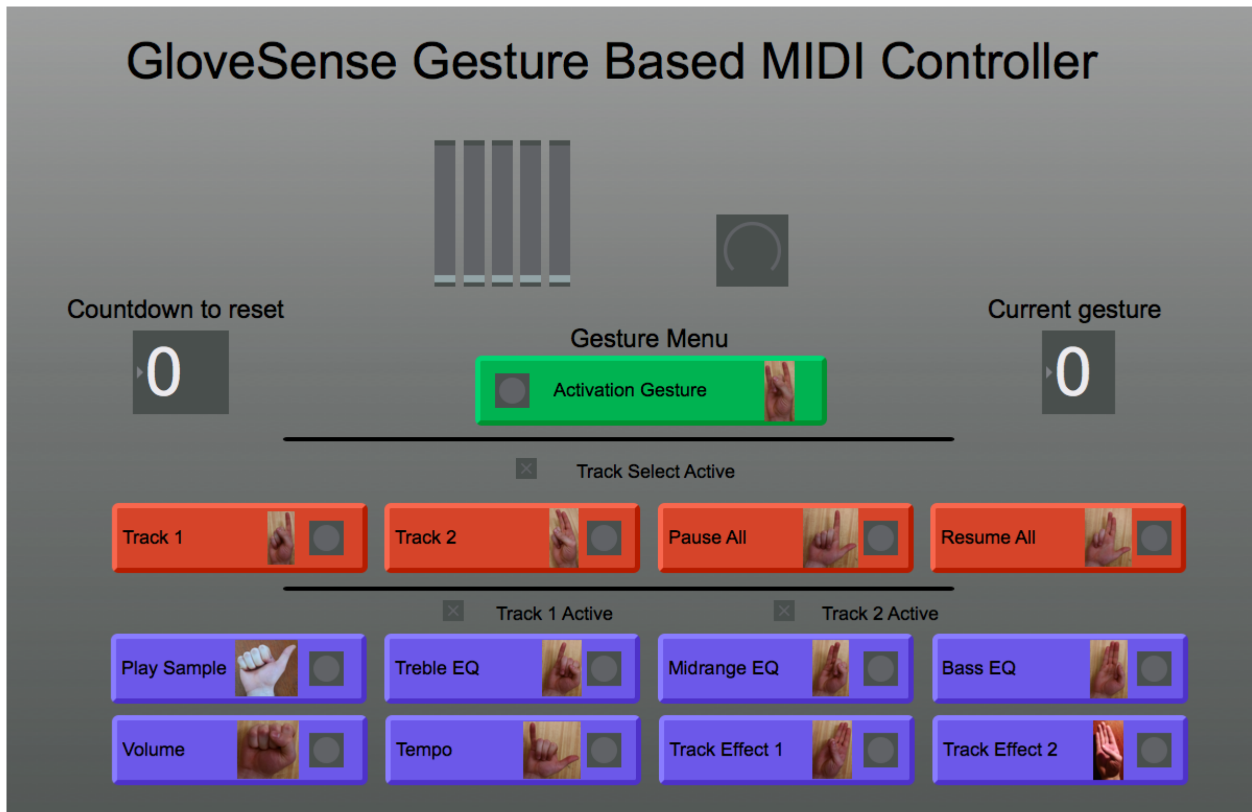


Figure 4.8: GUI with Gesture Map

4.2.2 AHRS Interpretation

The team routed the roll value from the AHRS, to the proper section of code in a similar manner to the flex sensor readings. The team routed the roll data, in the format $roll : value$, using the route object. The change in this reading was used to linearly modify a MIDI value. For example, a clockwise rotation of 45 degrees would increase the MIDI value by 45. However, any changes that were less than five degrees per sample were filtered out as they could be unintentional, and changes greater than 100 degrees per sample were also filtered out to prevent drastic instantaneous changes. The team decided to use the change in roll values rather than gyroscope data as it produced smoother changes in MIDI value. The MIDI value obtained this way could then be mapped directly to MIDI controls.

4.2.3 Ableton Live Interface

Once the team created the patches described in section 4.2.1, the MIDI control outputs for each gesture were chosen and mapped to various parameters in Ableton live. The *ctlout* object was used to send MIDI data to Ableton. This object takes in a MIDI value, a controller number, and a channel. The value of these messages ranges from 0 to 127, and there are 128 controllers available on each MIDI channel. Each gesture was then mapped to a specific parameter in Ableton.

A user can map a physical knob or button to controls within Ableton, and the process is very similar for mapping control messages from Max. Upon entering the MIDI map mode in Ableton, as seen in figure 4.9, the user selects the parameter they wish to map to and sends a control message with the desired controller number and channel. The EQ, Effect, and Volume/Fade gestures were mapped to knobs, which can be turned with rotation of the wrist to adjust values. The Pause, Resume, and Play Sample gestures were mapped to buttons, which are activated upon making the corresponding gesture. All mapped MIDI controllers are seen in figure 4.10.

Figure 4.9: MIDI Map Mode in Ableton Live

MIDI Mappings

C...	Note/Control	Path	Name	Min	Max
1	CC 3	Transport	Start		
1	CC 4	Transport	Stop		
1	CC 6	1-Audio | EQ Three	GainHi	-12.0 dB	6.00 dB
2	CC 6	2-Audio | EQ Three	GainHi	-12.0 dB	6.00 dB
1	CC 7	1-Audio | EQ Three	GainMid	-12.0 dB	6.00 dB
2	CC 7	2-Audio | EQ Three	GainMid	-12.0 dB	6.00 dB
1	CC 8	1-Audio | EQ Three	GainLo	-12.0 dB	6.00 dB
2	CC 8	2-Audio | EQ Three	GainLo	-12.0 dB	6.00 dB
1	CC 9	1-Audio | Mixer	Track Volume	-6.0 dB	6.0 dB
2	CC 9	2-Audio | Mixer	Track Volume	-6.0 dB	6.0 dB
1	CC 10	Master | Mixer	Song Tempo	40.00	240.00
2	CC 10	Master	Song Tempo(fine)		
1	CC 11	1-Audio | Old Sam...	Sample Hard	1	40
1	CC 11	2-Audio | Old Sam...	Bit Depth	3	7
1	CC 12	1-Audio | Hiss	Frequency	300 Hz	18.0 kHz

Figure 4.10: All MIDI Map in Ableton Live

4.3 Design Summary

The team used the different modules presented in this Chapter to create a dataglove capable of detecting motions and gestures. The hardware and software modules were designed concurrently, through the use of a rapid prototype made using the Arduino Uno. The final design fully satisfied the initial design specifications, but the team discusses potential improvements to the design in Chapter 5.

Chapter 5

Results

This Chapter presents the results the team gathered during the project. Since GloveSense was an experiment in product design, there are not many quantitative results. The team will primarily discuss the limitations of the final design and potential design improvements in section 5.1, Hardware Results, and section 5.2, Software Results. The primary quantitative result of this project is the gesture recognition accuracy, discussed in section 5.3.

5.1 Hardware Limitations and Improvements

The final hardware design utilized in this project fully completed the initial goals of the project. However, there were areas that could be improved.

5.1.1 Mi.Mu Glove

The Mi.Mu glove is a dataglove designed specifically for musical applications. This glove was not discussed in section 3.1 because the Mi.Mu glove was produced concurrently with this project, and was publicized towards the end of GloveSense development. The Mi.Mu glove is seen in figure 5.1. The battery is housed opposite the PCB, below the wrist on the inner forearm [Perner-Wilson].

Figure 5.1: Mi.Mu Dataglove [Perner-Wilson]

The Mi.Mu glove utilizes similar gesture detection to GloveSense, using flex sensors and an IMU. The Mi.Mu glove has a number of major advantages over GloveSense, namely haptic feedback motors and open fingertips and palm. These allow the user to better interact with other instruments while using the glove, as the haptic feedback allows for user feedback without the need for a computer screen, and the open fingertips and palm allow for better tactile feedback when interacting with other objects. Additionally, the Mi.Mu glove has all the components directly on the glove, not requiring the use of a forearm case. All these aspects should be considered when creating an improved version of GloveSense.

5.1.2 Accelerometer

The team only utilized one gyroscope on the 9-axis IMU. If the software design was to remain unchanged, a higher accuracy single gyroscope could be employed to increase the accuracy of MIDI value modification in the gesture menu. If changes to software were to be made, as described in section 5.2.1, a higher accuracy 9-axis IMU could be employed to fully utilize the software changes.

5.1.3 Printed Circuit Board

The team used a PCB prototyping board to implement the final design, as it was inexpensive and simple to implement, with no additional software design necessary. An obvious improvement to this is to create a custom PCB that could house all the aspects of the glove. Currently, the IMU, Bluetooth and Arduino are each on their own PCBs. The design could be improved by putting all these components on one PCB, alongside the voltage divider circuit and other necessary circuitry. This will drastically decrease the footprint of the circuitry, eliminating the need for a forearm case.

5.2 Software Limitations and Improvements

The software was fully able to carry out the intended application, but some limitations were still present. The software limitations are harder to quantify that the hardware ones, as the software was designed with a specific application in mind.

5.2.1 Accelerometer Interpretation

The team found that the greatest impediment to accurate performance was the coarse adjustment that the hand rotation gave. In the current design, as described in section 4.2.2, one degree of rotation corresponds to one unit of MIDI value change. If a single high-accuracy gyroscope was employed, as described in section 5.1.2, this could be fixed in software by allowing the user to select the scaling factor for each parameter; from coarse, less than one degree per unit MIDI, to fine, greater than one degree per unit MIDI. This would allow for higher precision during performance.

Alternately, if a high-accuracy 9-axis IMU was employed, an improved AHRS and position measurement system could be developed. This would allow software to use the position of the glove in three-space to make different gestures available, enabling many more possibilities for final application. One such application is a drum simulator, where glove location on an X-Y plane would change the instrument timbre, and Z-axis movement would set the note velocity. This would likely necessitate the use of one glove for each hand.

5.3 Gesture Recognition Accuracy

The team tested gesture accuracy by performing the same gesture 50 times, starting from multiple different gestures. This allowed the team to test the many different gesture permutations that would be used in a performance. The results for the individual gestures and the overall accuracy is seen in table 5.1.

Gesture	Accuracy (%)
0	100
1	96
2	98
3	94
4	98
5	100
6	90
7	100
8	98
9	98
10	98
11	100
12	92
Total	97.1

Table 5.1: Gesture Accuracy

The accuracy of the GloveSense hardware is high enough that it is usable in a performance situation, but there is still room for improvement. Some gestures are recognized as low as 90% of the time, which is nearing unacceptable. This could be improved by utilizing a faster microprocessor, as some of the gesture inaccuracy occurred because of delays between flex sensor measurement on the Arduino and communication with Max. Additionally, the flex sensors tended to shift in the glove after long periods of use, which made certain fingers, namely the little finger and thumb, more difficult to get an accurate reading from.

5.4 Conclusion

The turntable mixing board use case proved the potential of a MIDI-capable dataglove. The limitations discovered during testing could be remedied with a better design, or through a different use case. The team believes that with further development, datagloves such as GloveSense and the Mi.Mu glove will become commonplace in musical applications. The team believes that another potential market for the dataglove will be virtual reality. Overall, the GloveSense project was a success in proving the usefulness of a dataglove in musical application.

Chapter 6

Appendices

6.1 Hardware Schematic

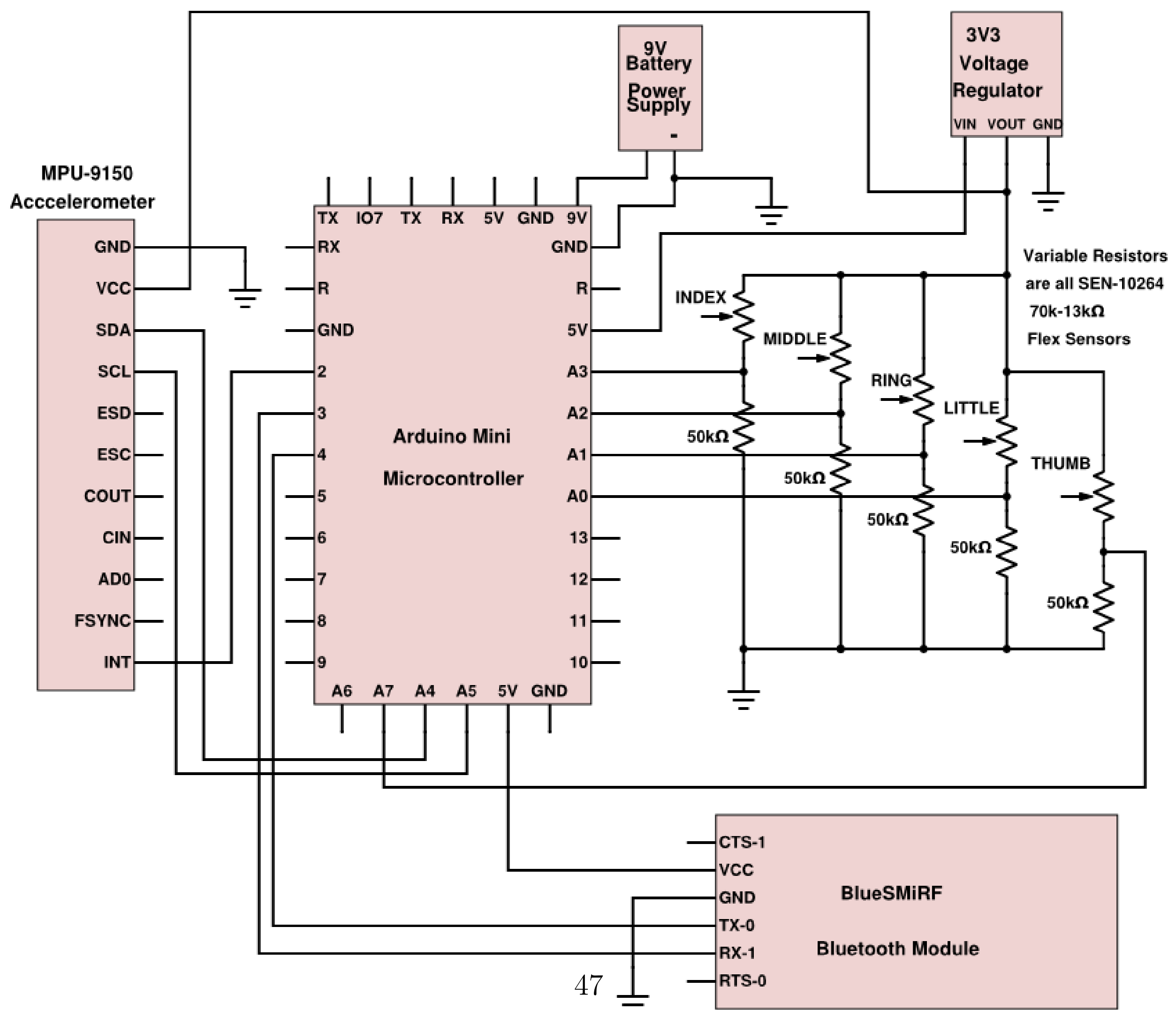


Figure 6.1: Hardware Schematic

6.2 Arduino Code

```
/****************************************************************
MPU9150_AHRS_directdata.ino
SFE_MPU9150 Library AHRS Data Fusion Example Code
Kris Winer for Sparkfun Electronics
Original Creation Date: April 8, 2014
https://github.com/sparkfun/MPU9150_Breakout

The MPU9150 is a versatile 9DOF sensor. It has a built-in
accelerometer, gyroscope, and magnetometer that
functions over I2C. It is very similar to the 6 DoF MPU6050 for which an extensive
library has already been built. Most of the function of the MPU9150 can utilize the MPU6050 library.

This Arduino sketch utilizes Jeff Rowberg's MPU6050 library to generate the basic sensor data
for use in two sensor fusion algorithms becoming increasingly popular with
DIY quadcopter and robotics engineers. I have added and slightly modified Jeff's library here.

This simple sketch will demo the following:
* How to create a MPU6050 object, using a constructor (global variables section).
* How to use the initialize() function of the MPU6050 class.
* How to read the gyroscope, accelerometer, and magnetometer
  using the readAcceleration(), readRotation(), and readMag() functions and the
  gx, gy, gz, ax, ay, az, mx, my, and mz variables.
* How to calculate actual acceleration, rotation speed, magnetic
  field strength using the  specified ranges as described in the data sheet:
  http://dlnmh9ip6v2uc.cloudfront.net/datasheets/Sensors/IMU/PS-MPU-9150A.pdf
  and
  http://dlnmh9ip6v2uc.cloudfront.net/datasheets/Sensors/IMU/RM-MPU-9150A-00.pdf.

In addition, the sketch will demo:
* How to check for data updates using the data ready status register
* How to display output at a rate different from the sensor data update and fusion filter update rates
* How to specify the accelerometer and gyro sampling and bandwidth rates
* How to use the data from the MPU9150 to fuse the sensor data into a quaternion
  representation of the sensor frame orientation relative to a fixed Earth frame
  providing absolute orientation information for subsequent use.
* An example of how to use the quaternion data to generate standard aircraft orientation
  data in the form of Tait-Bryan angles representing the sensor yaw, pitch, and roll angles
  suitable for any vehicle stabilization control application.

Hardware setup: This library supports communicating with the
MPU9150 over I2C. These are the only connections that need to be made:
        MPU9150 --------- Arduino
         SCL ---------- SCL (A5 on older 'Duinos')
         SDA ---------- SDA (A4 on older 'Duinos')
         VDD ------------- 3.3V
         GND ------------- GND

The LSM9DS0 has a maximum voltage of 3.5V. Make sure you power it
off the 3.3V rail! And either use level shifters between SCL
and SDA or just use a 3.3V Arduino Pro.

In addition, this sketch uses a Nokia 5110 48 x 84 pixel display which requires
digital pins 5 - 9 described below. If using SPI you might need to press one of the A0 - A3 pins
```

```
into service as a digital input instead.

Development environment specifics:
        IDE: Arduino 1.0.5
        Hardware Platform: Arduino Pro 3.3V/8MHz
        MPU9150 Breakout Version: 1.0

This code is beerware. If you see me (or any other SparkFun
employee) at the local, and you've found our code helpful, please
buy us a round!

Distributed as-is; no warranty is given.
*****************************************************************/

#include <Wire.h>
#include "I2Cdev.h"
#include "MPU6050_9Axis_MotionApps41.h"
#include <SoftwareSerial.h>

// Declare device MPU6050 class
MPU6050 mpu;

// global constants for 9 DoF fusion and AHRS (Attitude and Heading Reference System)
#define GyroMeasError PI * (40.0f / 180.0f)
// gyroscope measurement error in rads/s (shown as 3 deg/s)
#define GyroMeasDrift PI * (0.0f / 180.0f)
// gyroscope measurement drift in rad/s/s (shown as 0.0 deg/s/s)


// There is a tradeoff in the beta parameter between accuracy and response speed.
// In the original Madgwick study, beta of 0.041
// (corresponding to GyroMeasError of 2.7 degrees/s) was found to give optimal accuracy.
// However, with this value, the LSM9SD0 response time is about 10 seconds to a stable initial quaternion.
// Subsequent changes also require a longish lag time to a stable output, not fast
// enough for a quadcopter or robot car!
// By increasing beta (GyroMeasError) by about a factor of fifteen, the response time
// constant is reduced to ~2 sec
// I haven't noticed any reduction in solution accuracy. This is essentially the
// I coefficient in a PID control sense;
// the bigger the feedback coefficient, the faster the solution converges,
// usually at the expense of accuracy.
// In any case, this is the free parameter in the Madgwick filtering and fusion scheme.
#define beta sqrt(3.0f / 4.0f) * GyroMeasError   // compute beta
#define zeta sqrt(3.0f / 4.0f) * GyroMeasDrift   // compute zeta, the other free parameter
// in the Madgwick scheme usually set to a small or zero value
#define Kp 2.0f * 5.0f // these are the free parameters in the Mahony filter and fusion scheme,
// Kp for proportional feedback, Ki for integral
#define Ki 0.0f

int16_t a1, a2, a3, g1, g2, g3, m1, m2, m3;     // raw data arrays reading
uint16_t count = 0;  // used to control display output rate
uint16_t delt_t = 0; // used to control display output rate
uint16_t mcount = 0; // used to control display output rate
uint8_t MagRate;     // read rate for magnetometer data

float pitch, yaw, roll;
```

```
float deltat = 0.0f;        // integration interval for both filter schemes
uint16_t lastUpdate = 0; // used to calculate integration interval
uint16_t now = 0;        // used to calculate integration interval

float ax, ay, az, gx, gy, gz, mx, my, mz; // variables to hold latest sensor data values
float q[4] = {1.0f, 0.0f, 0.0f, 0.0f};    // vector to hold quaternion
float eInt[3] = {0.0f, 0.0f, 0.0f};       // vector to hold integral error for Mahony method

int flexSensorPin0 = A7; //analog pin 7
int flexSensorPin1 = A3; //analog pin 3
int flexSensorPin2 = A2; //analog pin 2
int flexSensorPin3 = A1; //analog pin 1
int flexSensorPin4 = A0; //analog pin 0

int bluetoothTx = 4;  // TX-O pin of bluetooth mate, Arduino D3
int bluetoothRx = 3;  // RX-I pin of bluetooth mate, Arduino D4

SoftwareSerial bluetooth(bluetoothTx, bluetoothRx);

void setup()
{
    //Serial.begin(9600);

    bluetooth.begin(115200);  // The Bluetooth Mate defaults to 115200bps
    bluetooth.print("$");  // Print three times individually
    bluetooth.print("$");
    bluetooth.print("$");  // Enter command mode
    delay(100);  // Short delay, wait for the Mate to send back CMD
    bluetooth.println("U,9600,N");  // Temporarily Change the baudrate to 9600, no parity
    // 115200 can be too fast at times for NewSoftSerial to relay the data reliably

    // join I2C bus (I2Cdev library doesn't do this automatically)
    Wire.begin();
    //bluetooth.begin(38400); // Start serial at 38400 bps

    // initialize MPU6050 device
    bluetooth.println(F("Initializing I2C devices..."));
    mpu.initialize();

    // verify connection
    bluetooth.println(F("Testing device connections..."));
    bluetooth.println(mpu.testConnection() ? F("MPU9150 connection successful") : F("MPU9150 connection failed"));

    bluetooth.begin(9600);  // doesn't seem to affect baud rate for bt, only 38400 works

// Set up the accelerometer, gyro, and magnetometer for data output

   mpu.setRate(7); // set gyro rate to 8 kHz/(1 * rate) shows 1 kHz, accelerometer ODR is fixed at 1 KHz

   MagRate = 10; // set magnetometer read rate in Hz; 10 to 100 (max) Hz are reasonable values

// Digital low pass filter configuration.
// It also determines the internal sampling rate used by the device as shown in the table below.
// The accelerometer output rate is fixed at 1kHz. This means that for a Sample
// Rate greater than 1kHz, the same accelerometer sample may be output to the
// FIFO, DMP, and sensor registers more than once.
```

```
/*
 *           |   ACCELEROMETER    |           GYROSCOPE
 * DLPF_CFG | Bandwidth | Delay  | Bandwidth | Delay  | Sample Rate
 * ---------+-----------+--------+-----------+--------+-------------
 * 0        | 260Hz     | 0ms    | 256Hz     | 0.98ms | 8kHz
 * 1        | 184Hz     | 2.0ms  | 188Hz     | 1.9ms  | 1kHz
 * 2        | 94Hz      | 3.0ms  | 98Hz      | 2.8ms  | 1kHz
 * 3        | 44Hz      | 4.9ms  | 42Hz      | 4.8ms  | 1kHz
 * 4        | 21Hz      | 8.5ms  | 20Hz      | 8.3ms  | 1kHz
 * 5        | 10Hz      | 13.8ms | 10Hz      | 13.4ms | 1kHz
 * 6        | 5Hz       | 19.0ms | 5Hz       | 18.6ms | 1kHz
 */
  mpu.setDLPFMode(4); // set bandwidth of both gyro and accelerometer to ~20 Hz

// Full-scale range of the gyro sensors:
// 0 = +/- 250 degrees/sec, 1 = +/- 500 degrees/sec, 2 = +/- 1000 degrees/sec, 3 = +/- 2000 degrees/sec
  mpu.setFullScaleGyroRange(0); // set gyro range to 250 degrees/sec

// Full-scale accelerometer range.
// The full-scale range of the accelerometer: 0 = +/- 2g, 1 = +/- 4g, 2 = +/- 8g, 3 = +/- 16g
  mpu.setFullScaleAccelRange(0); // set accelerometer to 2 g range

  mpu.setIntDataReadyEnabled(true); // enable data ready interrupt
}

void loop()
{
  int timerBegin;
  int timerEnd;
  int flexSensorReading0 = analogRead(flexSensorPin0);
  int flexSensorReading1 = analogRead(flexSensorPin1);
  int flexSensorReading2 = analogRead(flexSensorPin2);
  int flexSensorReading3 = analogRead(flexSensorPin3);
  int flexSensorReading4 = analogRead(flexSensorPin4);

        if(mpu.getIntDataReadyStatus() == 1) { // wait for data ready status register to
        //  update all data registers
           mcount++;
          // read the raw sensor data
           mpu.getAcceleration  ( &a1, &a2, &a3  );
           ax = a1*2.0f/32768.0f; // 2 g full range for accelerometer
           ay = a2*2.0f/32768.0f;
           az = a3*2.0f/32768.0f;

           mpu.getRotation  ( &g1, &g2, &g3  );
           gx = g1*250.0f/32768.0f; // 250 deg/s full range for gyroscope
           gy = g2*250.0f/32768.0f;
           gz = g3*250.0f/32768.0f;
//  The gyros and accelerometers can in principle be calibrated in addition to any
//  factory calibration but they are generally
//  pretty accurate. You can check the accelerometer by making sure the reading
//  is +1 g in the positive direction for each axis.
//  The gyro should read zero for each axis when the sensor is at rest. Small or
//  zero adjustment should be needed for these sensors.
//  The magnetometer is a different thing. Most magnetometers will be sensitive to circuit currents, computers, and
//  other both man-made and natural sources of magnetic field. The rough way to
```

```
//  calibrate the magnetometer is to record
//  the maximum and minimum readings (generally achieved at the North magnetic direction).
//  The average of the sum divided by two
//  should provide a pretty good calibration offset. Don't forget that for the MPU9150,
//  the magnetometer x- and y-axes are switched
//  compared to the gyro and accelerometer!
            if (mcount > 1000/MagRate) \{  // this is a poor man's way of setting the magnetometer read rate (see below)
            mpu.getMag  ( &m1, &m2, &m3 );
            mx = m1*10.0f*1229.0f/4096.0f + 18.0f;
            // milliGauss (1229 microTesla per 2^12 bits, 10 mG per microTesla)

            my = m2*10.0f*1229.0f/4096.0f + 70.0f;
            // apply calibration offsets in mG that correspond to your environment and magnetometer

            mz = m3*10.0f*1229.0f/4096.0f + 270.0f;
            mcount = 0;
            }
          }

  now = micros();
  deltat = ((now - lastUpdate)/1000000.0f); // set integration time by time elapsed since last filter update
  lastUpdate = now;
  // Sensors x (y)-axis of the accelerometer is aligned with the y (x)-axis of the magnetometer;
  // the magnetometer z-axis (+ down) is opposite to z-axis (+ up) of accelerometer and gyro!
  // We have to make some allowance for this orientationmismatch in feeding the output to the quaternion filter.
  // For the MPU-9150, we have chosen a magnetic rotation that keeps the sensor forward along the x-axis just like
  // in the LSM9DS0 sensor. This rotation can be modified to allow any convenient orientation convention.
  // This is ok by aircraft orientation standards!
  // Pass gyro rate as rad/s
   MadgwickQuaternionUpdate(ax, ay, az, gx*PI/180.0f, gy*PI/180.0f, gz*PI/180.0f,  my,  mx, mz);
// MahonyQuaternionUpdate(ax, ay, az, gx*PI/180.0f, gy*PI/180.0f, gz*PI/180.0f, my, mx, mz);

    // Serial print and/or display at 0.5 s rate independent of data rates
    delt_t = millis() - count;
    if (delt_t > 500) { // update LCD once per half-second independent of read rate

  // Define output variables from updated quaternion---these are Tait-Bryan angles,
  // commonly used in aircraft orientation.
  // In this coordinate system, the positive z-axis is down toward Earth.
  // Yaw is the angle between Sensor x-axis and Earth magnetic North
  // (or true North if corrected for local declination, looking down on the sensor
  // positive yaw is counterclockwise.
  // Pitch is angle between sensor x-axis and Earth ground plane, toward the Earth is
  // positive, up toward the sky is negative.
  // Roll is angle between sensor y-axis and Earth ground plane, y-axis up is positive roll.
  // These arise from the definition of the homogeneous rotation matrix constructed from quaternions.
  // Tait-Bryan angles as well as Euler angles are non-commutative;
  // that is, the get the correct orientation the rotations must be
  // applied in the correct order which for this configuration is yaw, pitch, and then roll.
  // For more see http://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles
  // which has additional links.
    yaw   = atan2(2.0f * (q[1] * q[2] + q[0] * q[3]), q[0] * q[0] + q[1] * q[1] - q[2] * q[2] - q[3] * q[3]);
    pitch = -asin(2.0f * (q[1] * q[3] - q[0] * q[2]));
    roll  = atan2(2.0f * (q[0] * q[1] + q[2] * q[3]), q[0] * q[0] - q[1] * q[1] - q[2] * q[2] + q[3] * q[3]);
    pitch *= 180.0f / PI;
    yaw   *= 180.0f / PI - 13.8; // Declination at Danville, California is 13 degrees
```

```
    // 48 minutes and 47 seconds on 2014-04-04
    roll  *= 180.0f / PI;

  bluetooth.print("roll: "); bluetooth.println(roll, 2); //the 2 means two decimal places

  bluetooth.print("finger 0 ");
  bluetooth.println(flexSensorReading0);
  bluetooth.print("finger 1 ");
  bluetooth.println(flexSensorReading1);
  bluetooth.print("finger 2 ");
  bluetooth.println(flexSensorReading2);
  bluetooth.print("finger 3 ");
  bluetooth.println(flexSensorReading3);
  bluetooth.print("finger 4 ");
  bluetooth.println(flexSensorReading4);
  //timerEnd = millis();
  //bluetooth.print("time: "); bluetooth.println(timerEnd - timerBegin);
  //timerBegin = millis();
    // With these settings the filter is updating at a ~145 Hz rate using the Madgwick scheme and
    // >200 Hz using the Mahony scheme even though the display refreshes at only 2 Hz.
    // The filter update rate is determined mostly by the mathematical steps in the respective algorithms,
    // the processor speed (8 MHz for the 3.3V Pro Mini), and the magnetometer ODR:
    // an ODR of 10 Hz for the magnetometer produce the above rates, maximum magnetometer ODR of 100 Hz produces
    // filter update rates of 36 - 145 and ~38 Hz for the Madgwick and Mahony schemes, respectively.
    // This is presumably because the magnetometer read takes longer than the gyro or accelerometer reads.
    // This filter update rate should be fast enough to maintain accurate platform orientation for
    // stabilization control of a fast-moving robot or quadcopter. Compare to the update rate of 200 Hz
    // produced by the on-board Digital Motion Processor of Invensense's MPU6050 6 DoF and MPU9150 9DoF sensors.
    // The 3.3 V 8 MHz Pro Mini is doing pretty well!

    count = millis();
    }
}


// Implementation of Sebastian Madgwick's "...efficient orientation filter for... inertial/magnetic sensor arrays"
// (see http://www.x-io.co.uk/category/open-source/ for examples and more details)
// which fuses acceleration, rotation rate, and magnetic moments to produce a quaternion-based estimate of absolute
// device orientation -- which can be converted to yaw, pitch, and roll. Useful for stabilizing quadcopters, etc.
// The performance of the orientation filter is at least as good as conventional Kalman-based filtering algorithms
// but is much less computationally intensive---it can be performed on a 3.3 V Pro Mini operating at 8 MHz!
        void MadgwickQuaternionUpdate(float ax, float ay, float az, float gx, float gy,
                        float gz, float mx, float my, float mz)
        {
            float q1 = q[0], q2 = q[1], q3 = q[2], q4 = q[3];   // short name local variable for readability
            float norm;
            float hx, hy, _2bx, _2bz;
            float s1, s2, s3, s4;
            float qDot1, qDot2, qDot3, qDot4;

            // Auxiliary variables to avoid repeated arithmetic
            float _2q1mx;
            float _2q1my;
            float _2q1mz;
            float _2q2mx;
            float _4bx;
```

```
float _4bz;
float _2q1 = 2.0f * q1;
float _2q2 = 2.0f * q2;
float _2q3 = 2.0f * q3;
float _2q4 = 2.0f * q4;
float _2q1q3 = 2.0f * q1 * q3;
float _2q3q4 = 2.0f * q3 * q4;
float q1q1 = q1 * q1;
float q1q2 = q1 * q2;
float q1q3 = q1 * q3;
float q1q4 = q1 * q4;
float q2q2 = q2 * q2;
float q2q3 = q2 * q3;
float q2q4 = q2 * q4;
float q3q3 = q3 * q3;
float q3q4 = q3 * q4;
float q4q4 = q4 * q4;

// Normalise accelerometer measurement
norm = sqrt(ax * ax + ay * ay + az * az);
if (norm == 0.0f) return; // handle NaN
norm = 1.0f/norm;
ax *= norm;
ay *= norm;
az *= norm;

// Normalise magnetometer measurement
norm = sqrt(mx * mx + my * my + mz * mz);
if (norm == 0.0f) return; // handle NaN
norm = 1.0f/norm;
mx *= norm;
my *= norm;
mz *= norm;

// Reference direction of Earth's magnetic field
_2q1mx = 2.0f * q1 * mx;
_2q1my = 2.0f * q1 * my;
_2q1mz = 2.0f * q1 * mz;
_2q2mx = 2.0f * q2 * mx;
hx = mx * q1q1 - _2q1my * q4 + _2q1mz * q3 + mx * q2q2 + _2q2 * my *
                                q3 + _2q2 * mz * q4 - mx * q3q3 - mx * q4q4;
hy = _2q1mx * q4 + my * q1q1 - _2q1mz * q2 + _2q2mx * q3 - my * q2q2
                                + my * q3q3 + _2q3 * mz * q4 - my * q4q4;
_2bx = sqrt(hx * hx + hy * hy);
_2bz = -_2q1mx * q3 + _2q1my * q2 + mz * q1q1 + _2q2mx * q4 - mz *
                                q2q2 + _2q3 * my * q4 - mz * q3q3 + mz * q4q4;
_4bx = 2.0f * _2bx;
_4bz = 2.0f * _2bz;

// Gradient decent algorithm corrective step
s1 = -_2q3 * (2.0f * q2q4 - _2q1q3 - ax) + _2q2 * (2.0f * q1q2 + _2q3q4 - ay)
    - _2bz * q3 * (_2bx * (0.5f - q3q3 - q4q4) + _2bz * (q2q4 - q1q3) - mx) +
    (-_2bx * q4 + _2bz * q2) * (_2bx * (q2q3 - q1q4) + _2bz * (q1q2 + q3q4) - my)
    + _2bx * q3 * (_2bx * (q1q3 + q2q4) + _2bz * (0.5f - q2q2 - q3q3) - mz);

s2 = _2q4 * (2.0f * q2q4 - _2q1q3 - ax) + _2q1 * (2.0f * q1q2 + _2q3q4 - ay) -
```

```
        4.0f * q2 * (1.0f - 2.0f * q2q2 - 2.0f * q3q3 - az) + _2bz * q4 * (_2bx *
        (0.5f - q3q3 - q4q4) + _2bz * (q2q4 - q1q3) - mx) + (_2bx * q3 + _2bz * q1)
        * (_2bx * (q2q3 - q1q4) + _2bz * (q1q2 + q3q4) - my) + (_2bx * q4 - _4bz * q2)
        * (_2bx * (q1q3 + q2q4) + _2bz * (0.5f - q2q2 - q3q3) - mz);

    s3 = -_2q1 * (2.0f * q2q4 - _2q1q3 - ax) + _2q4 * (2.0f * q1q2 + _2q3q4 - ay) -
        4.0f * q3 * (1.0f - 2.0f * q2q2 - 2.0f * q3q3 - az) + (-_4bx * q3 - _2bz * q1)
        * (_2bx * (0.5f - q3q3 - q4q4) + _2bz * (q2q4 - q1q3) - mx) + (_2bx * q2 +
        _2bz * q4) * (_2bx * (q2q3 - q1q4) + _2bz * (q1q2 + q3q4) - my) + (_2bx * q1
        - _4bz * q3) * (_2bx * (q1q3 + q2q4) + _2bz * (0.5f - q2q2 - q3q3) - mz);

    s4 = _2q2 * (2.0f * q2q4 - _2q1q3 - ax) + _2q3 * (2.0f * q1q2 + _2q3q4 - ay)
        + (-_4bx * q4 + _2bz * q2) * (_2bx * (0.5f - q3q3 - q4q4) + _2bz * (q2q4
        - q1q3) - mx) + (-_2bx * q1 + _2bz * q3) * (_2bx * (q2q3 - q1q4) + _2bz *
        (q1q2 + q3q4) - my) + _2bx * q2 * (_2bx * (q1q3 + q2q4) + _2bz *
        (0.5f - q2q2 - q3q3) - mz);

    norm = sqrt(s1 * s1 + s2 * s2 + s3 * s3 + s4 * s4);    // normalise step magnitude
    norm = 1.0f/norm;
    s1 *= norm;
    s2 *= norm;
    s3 *= norm;
    s4 *= norm;

    // Compute rate of change of quaternion
    qDot1 = 0.5f * (-q2 * gx - q3 * gy - q4 * gz) - beta * s1;
    qDot2 = 0.5f * (q1 * gx + q3 * gz - q4 * gy) - beta * s2;
    qDot3 = 0.5f * (q1 * gy - q2 * gz + q4 * gx) - beta * s3;
    qDot4 = 0.5f * (q1 * gz + q2 * gy - q3 * gx) - beta * s4;

    // Integrate to yield quaternion
    q1 += qDot1 * deltat;
    q2 += qDot2 * deltat;
    q3 += qDot3 * deltat;
    q4 += qDot4 * deltat;
    norm = sqrt(q1 * q1 + q2 * q2 + q3 * q3 + q4 * q4);    // normalise quaternion
    norm = 1.0f/norm;
    q[0] = q1 * norm;
    q[1] = q2 * norm;
    q[2] = q3 * norm;
    q[3] = q4 * norm;

}
```

6.3 Max Code

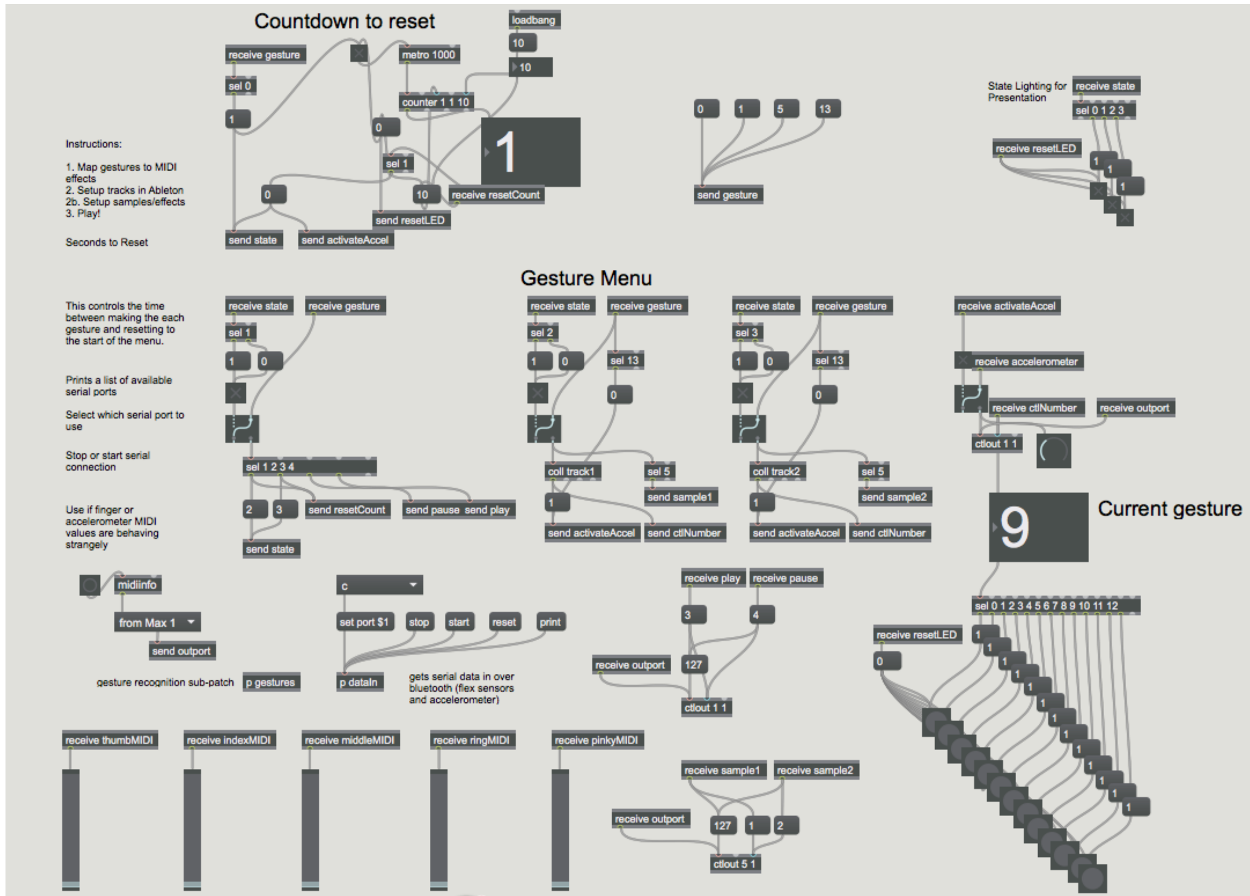


Figure 6.2: Main Max Code Modules

Figure 6.3: *Datain* Max Code Module

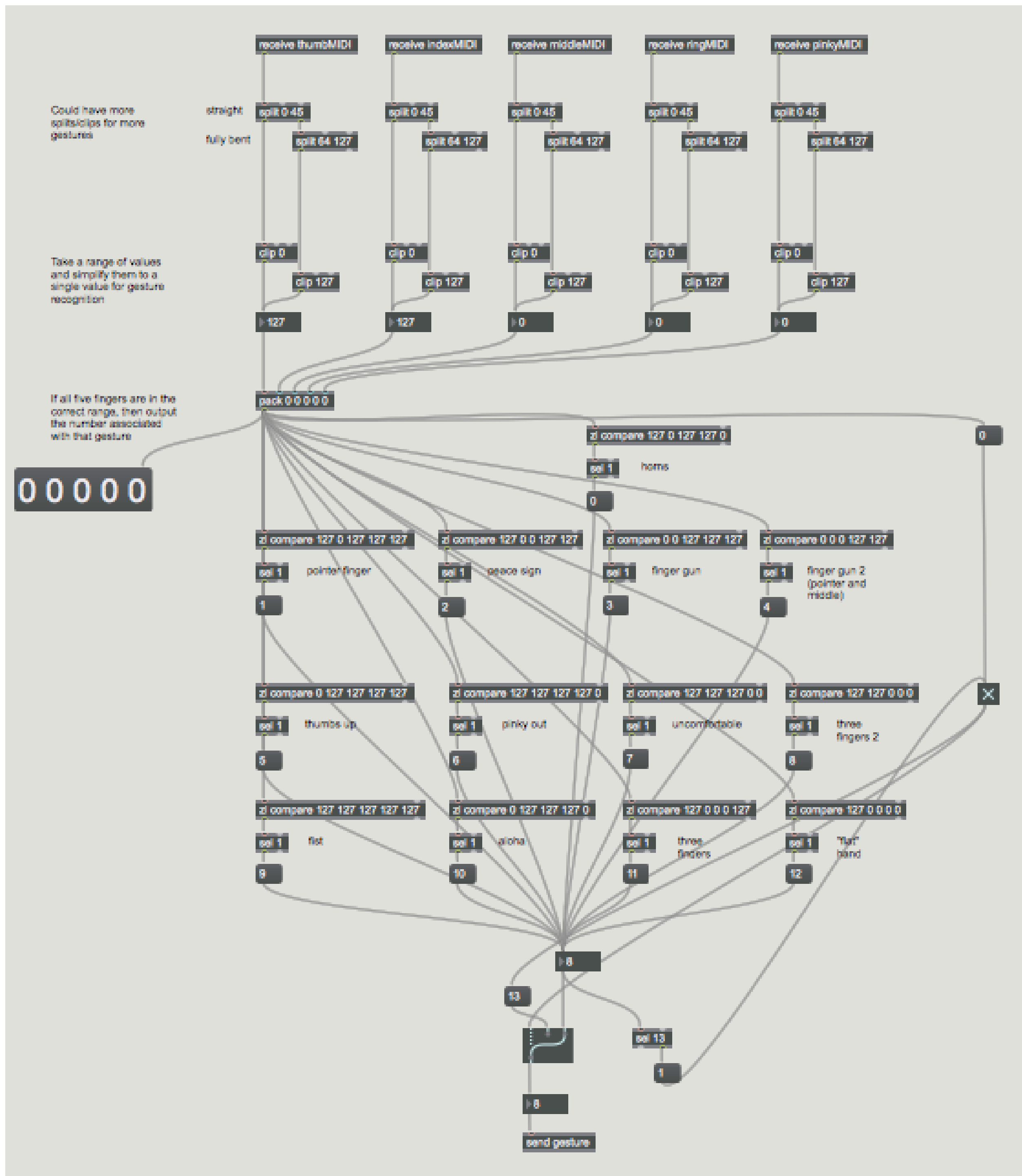


Figure 6.4: Gesture Recognition Max Code Module

6.4 Survey Results

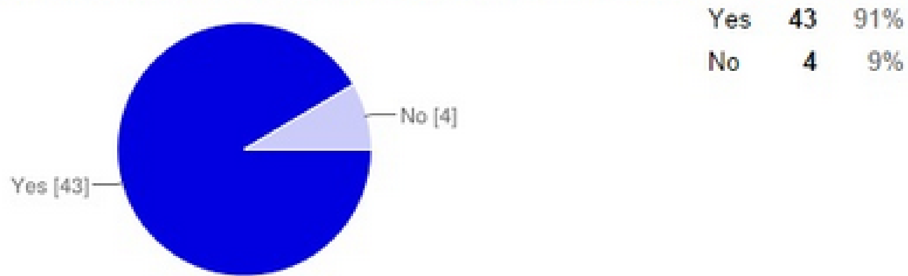


Figure 6.5: Survey Results, Glove Interest

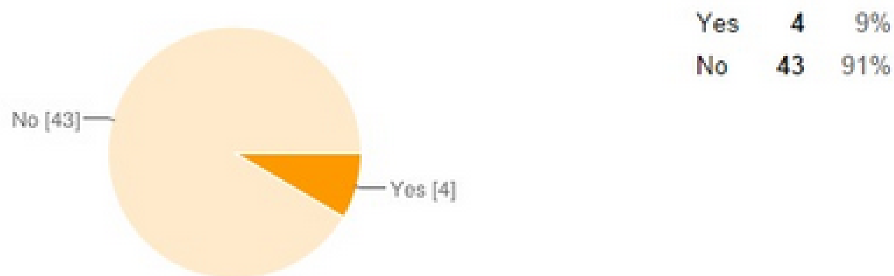


Figure 6.6: Survey Results, Prior Use

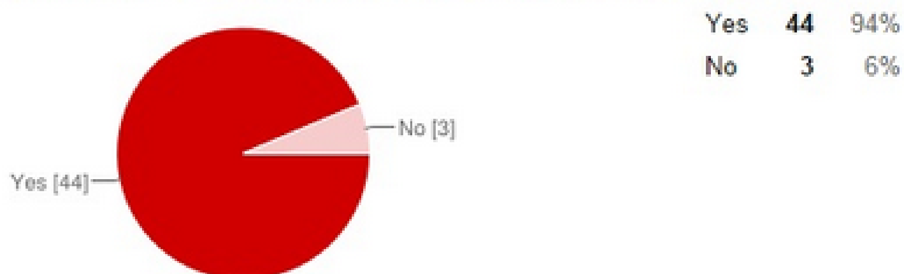


Figure 6.7: Survey Results, Music Application

Figure 6.8: Survey Response, Concert Application

Figure 6.9: Survey Response, Gathering Application

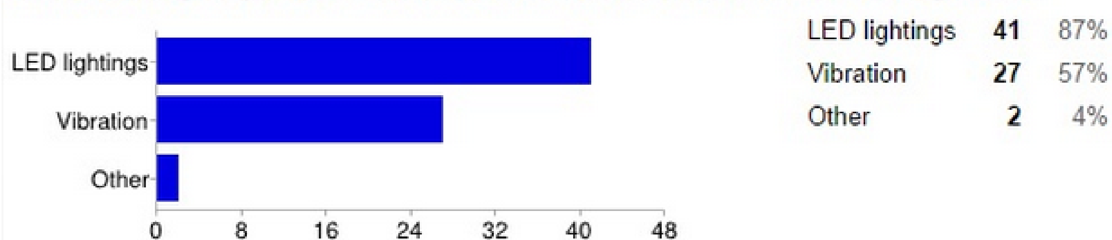
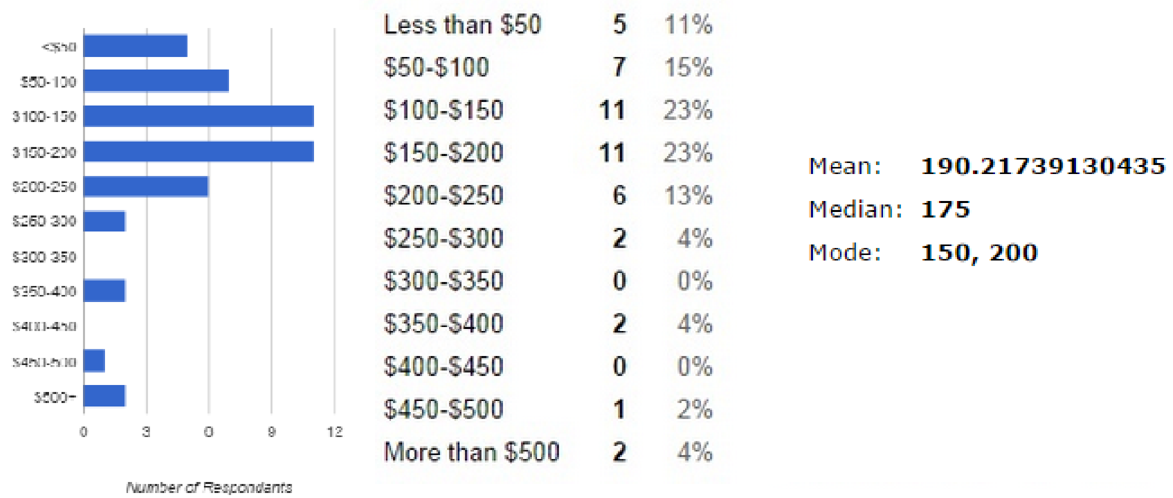


Figure 6.10: Survey Results, Feedback Device



How much would you be willing to pay for an "interactive audio effect controller glove"?

Less than $50	5	11%
$50-$100	7	15%
$100-$150	11	23%
$150-$200	11	23%
$200-$250	6	13%
$250-$300	2	4%
$300-$350	0	0%
$350-$400	2	4%
$400-$450	0	0%
$450-$500	1	2%
More than $500	2	4%

Figure 6.11: Survey Results, Cost

Bibliography

What is live? URL https://www.ableton.com/en/live/. Accessed: 2014.10.05.

Max. URL http://cycling74.com/products/max/. Accessed: 2014.10.12.

Enabletalk. URL http://www.enabletalk.com. Accessed: 2014.10.6.

9 degrees of freedom breakout - mpu-9150. URL https://www.sparkfun.com/products/11486. Accessed: 2014.10.14.

Sean Chen and Evan Levine. Mister gloves - a wireless usb gesture input system, 2010.

Rummenigge Dantas, Andre Pantoja, Antonio Pereira, and Lucas Silva. Development of a low cost dataglove based on arduino for virtual reality applications. *IEEE*, 2013.

William Rowan Hamilton. On quaternions; or on a new system of imaginaries in algebra (letter to john t. graves).

David Miles Huber. *The MIDI Manual*. SAMS, Carmel, Indiana, 1991.

Hannah Perner-Wilson. Mi.mu gloves. URL http://mimu.org.uk/. Accessed: 2015.3.9.

Andrew Swift. A brief introduction to MIDI, May 1997.

Various. Openpilot documentation. URL http://wiki.openpilot.org/display/ Doc/OpenPilot+ Documentation#OpenPilot. Accessed: 2014.11.25.