# University of Central Florida

College of Engineering and Computer Science

*EEL 4914*
*Senior Design 2*

*Programmable Trackpad*

*Group 18 Members:*
Taylor Barnes - CpE
Jonah Halili - CpE
Brian Modica - CpE
Bradley Vanderzalm - CpE

*Project Coordinators:*
Dr. Samuel Richie
Dr. Lei Wei

**1 Introduction** **11**
1.1 Motivation 11
1.2 Problem Statement 11
**2 Project Description** **2**
2.1 Goals and Objectives 2
2.2 Function 2
2.2.1 Device Layout 3
2.2.2 Orientation Switching 5
2.3 Requirements 5
2.4 Constraints and Standards 7
2.5 House of Quality 7
**3 Technology Investigation** **8**
3.1 Market Analysis 9
3.2 Hardware 10
3.2.1 Power 10
3.2.1.1 USB Power Standard 11
3.2.1.2 Power Calculations 11
3.2.1.3 Battery Technology 12
3.2.1.4 Battery Management Boards 13
3.2.1.5 Voltage Converter 14
3.2.1.6 Power Switch 16
3.2.2 Touchpad 16
3.2.2.1 AliExpress RGB TFT Touch LCD Display Module 17
3.2.2.2 DFRobot Capacitive Touch Kit for Arduino 18
3.2.2.3 Adafruit Resistive Touch Screen 19
3.2.2.4 Touchpad Comparison 20
3.2.3 Touchpad Controllers 21
3.2.3.1 AR1100 Chip 21
3.2.3.2 STMPE610 22
3.2.3.3 Touchpad Direct Connection 23
3.2.4 Input Units 24
3.2.4.1 Stretching The Objective 27
3.2.5 USB 28
3.2.6 Microcontroller 29
3.2.6.1 Microcontroller Technologies 29
3.2.6.2 Market Microcontrollers 30
3.2.7 Bluetooth 32

3.2.7.1 Bluetooth Standards 32
3.2.7.2 Bluetooth Modules 32
3.2.8 PCB 35
3.2.8.1 PCB Plate 35
3.2.8.2 PCB CAD Software 36
3.3 Firmware 37
3.3.1 Keyboard Profiles 37
3.3.1.1 Human Interface Device Specification 38
3.3.1.2 Quantum Mechanical Keyboard (and Derivatives) 38
3.3.1.3 Circuit Python for Bluetooth Connection 39
3.3.1.4 Arduino HID Library 40
3.3.2 Serial Wire Debug 40
3.4 Application Software 43
3.4.1 Coding Language Consideration 43
3.4.1.1 Python 44
3.4.1.2 Java 45
3.4.2 Programmable Macros/Changing Key Press Functionality 45
3.4.2.1 Microsoft PowerToys 46
3.4.2.2 AutoHotkey 46
3.4.3 Graphical User Interface (GUI) 47
3.4.4 Macro Preset Storage Considerations 48
3.4.4.1 Text File Storage 49
3.4.4.2 SQLite Database 50
3.5 Chassis and 3D printing 50
**4 Design Details 52**
4.1 Preliminary Designs 52
4.2 Hardware 54
4.2.1 Power System 54
4.2.1.1 Battery Management System Electrical Schematic 55
4.2.1.2 Voltage Converter Electrical Schematic 57
4.2.2 Microcontroller 58
4.2.2.1 Bluetooth Module 59
4.2.3 USB 63
4.2.4 Touchpad Control 64
4.2.5 Miscellaneous Input Units 64
4.3 Firmware 67
4.3.1 Establishing Connection 67
4.3.2 Hardware Control 67

4.3.3 Firmware Uploading 68
4.4 Application Software 68
4.4. AutoHotkey 69
4.4.2 User Interface 70
4.4.3 Macro Generation and Saving 73
4.5 Chassis 76
4.5.1 3D-Printed Enclosure 76
4.5.2 Mouse Buttons 77
**5 Prototypes and Testing 77**
5.1 Hardware 77
5.1.1 Battery Charging 78
5.1.1.1 Procedure 78
5.1.1.2 Results 78
5.1.2 Simultaneous Transmission and Charging 81
5.1.2.1 Procedure 81
5.1.2.2 Results 81
5.1.3 Configuring the Touchpad Controller 85
5.1.4 Microcontroller Selection 87
5.1.4.1 Procedure 87
5.1.4.2 Results 88
5.1.5 Input Unit Testing 88
5.1.5.1 Mechanical Switches 89
5.1.5.2 Rotary Encoders 89
5.1.6 Reading Touchpad Inputs 91
5.1.6.1 Prototype Environment 91
5.1.6.2 MCU Environment 94
5.2 Firmware 95
5.2.1 Macro, Rotary Encoder, and Mouse Functional Development 95
5.2.1.1 Backup Firmware 97
5.2.2 Bluetooth Connection 97
5.2.3 Procedure 97
5.2.4 Results 98
5.2.4.1 Prototype v1.0 99
5.2.4.2 Prototype v2.0 101
5.2.4.3 Prototype v3.0 (Final PCB) 102
5.3 Application Software 103
5.3.1 Running AutoHotkey Script Inside a Program 103
5.3.1.1 Procedure 104

5.3.1.2 Results 105
5.3.2 Running AHK Inside Python Prototype V2 107
5.3.2.1 Procedure 107
5.3.2.2 Result 108
**6 Administrative Materials 112**
6.1 Budget and Costs 112
6.2 Milestones 114
**7 Conclusion 119**
**8 Bibliography 121**

Figure 1: System Concept 1
Figure 2: Project Goals and Objectives 2
Figure 3: Project Functions 3
Figure 4: Device Layout for Right-handed Users 4
Figure 5: Device Layout for Left-handed Users 4
Figure 6: Project Requirements 6
Figure 7: Project Constraints and Standards 7
Figure 8: House of Quality 8
Figure 9: Market Programmable Mice 9
Figure 10: Market Trackpads 10
Figure 11: Discharge Time Equation 11
Figure 12: Charge Time Equation 11
Figure 13: Battery Technology Comparison 12
Figure 14: Discharge Profile of LiPo Batteries 13
Figure 15: Battery Management System Prototype Boards 14
Figure 16: Voltage Converter Comparison 15
Figure 17: 5 Volt Converter Comparison 15
Figure 18: Generic Power Switch 16
Figure 19: AliExpress RGB TFT Touch LCD Display Module 18
Figure 20: DFRobot Capacitive Touch Kit 19
Figure 21: Adafruit Resistive Touch Screen 20
Figure 22: Touchpad Comparison 20
Figure 23: AR1100 Chip 22
Figure 24: AR1100 Schematic 22
Figure 25: STMPE610 Chip Pictured on a Breakout Board 23
Figure 26: STMPE610 Schematic 23
Figure 27: Kailh Hot-Swappable Socket 24
Figure 28: Mechanical Switch Comparison 25
Figure 29: Generic Rotary Encoder and Tactile Switch Button 26
Figure 30: Mouse Button Implementation Sketch 26
Figure 31: Orientation Switch 27

Figure 32: Stretch Goal References for Input Devices 27
Figure 33: USB Pinout Diagram 28
Figure 34: USB Technology Comparison 29
Figure 35: Microcontroller Pin Requirements 30
Figure 36: Microcontroller Comparison 36
Figure 37: Bluetooth Module Comparison 33
Figure 38: Seeed Studio and MDBT42Q Side by Side 33
Figure 39: MDBT50Q Module PCB Footprint 34
Figure 40: Ergogen Output 35
Figure 41: PCB CAD Software Comparison 37
Figure 42: KMK Flowchart 39
Figure 43: SWD Programmers Comparison 41
Figure 44: Third-Party J-Link Debugger 42
Figure 45: Third-Party J-Link Debugger Diagram 42
Figure 46: Python vs Java Comparison 44
Figure 47: Flowchart of AHK Implemented in App Software 47
Figure 48: GUI Library Comparison 48
Figure 49: 3D Printing Material Comparison 51
Figure 50: Chassis Sketches 51
Figure 51: Chassis Design Considerations Table 52
Figure 52: Initial Concept Sketches 53
Figure 53: Initial Concept CAD Model 54
Figure 54: Generalized Power System Flow Diagram 55
Figure 55: MCP73831 Circuit Diagram from Datasheet 56
Figure 56: MCP73831 Current Regulation Equation 56
Figure 57: MCP73831 Circuit Schematic 57
Figure 58: LM3671 Circuit Diagram from Datasheet 57
Figure 59: LM3671 Circuit Schematic 58
Figure 60: nRF52840 Connection Diagram 59
Figure 61: MDBT50Q-1MV2 Technical Specifications 60
Figure 62: MDBT50Q Circuit Diagram from Documentation 60

Figure 63: MDBT50Q Circuit Schematic 61
Figure 64: MDBT50Q Pin Description Table 62
Figure 65: USB Breakout Diagram 63
Figure 66: USB Block Diagram 64
Figure 67: Circuit Schematic for Macro Keys 68
Figure 68: Circuit Schematic for Mouse Buttons 68
Figure 69: Circuit Schematic for Rotary Encoders 66
Figure 70: Circuit Schematic for Orientation Switch 66
Figure 71: Bluetooth Pairing Process Flowchart 67
Figure 72: Hardware Control Flow Diagram 68
Figure 73: Application Layout Mockup 69
Figure 74: AutoHotkey Block Diagram 70
Figure 75: User Interface Block Diagram 71
Figure 76: Creating New Macro Preset Menu Design 72
Figure 77: Editing Existing Macro Preset Menu Design 73
Figure 78: Macro Generation Flowchart 74
Figure 79: Search Results Window showing all macros the user created 75
Figure 80: Search Results Window with search query 76
Figure 81: 3D Rendering of Chassis 76
Figure 82: Mouse Button Construction 77
Figure 83: TP4056 Charging Characteristics 79
Figure 84: MCP73831 Charging Characteristics 80
Figure 85: Wiring Diagram of Charging Prototype With Generic Computer Mouse 82
Figure 86: Table of Voltages During Failed Simultaneous Transmission and Charging 83
Figure 87: Wiring Diagram of Charging Prototype With Arduino Uno 84
Figure 88: AR110 Breakout Board 85
Figure 89: AR110 Breakout Board PCB Layout 86
Figure 90: AR110 Breakout Board Config Settings 86

Figure 91: ItsyBitsy Prototyping Board 87
Figure 92: NeoKey Socket Breakout Board 89
Figure 93: PEC11 Rotary Encoder 90
Figure 94: Rotary Encoder Prototype State Diagram 90
Figure 95: Arduino Serial Output 92
Figure 96: Touchpad Prototype using Arduino Uno 93
Figure 97: Additional Prototype Close-ups 93
Figure 98 Touchpad Prototype Schematic 94
Figure 99: Initial Macro Prototyping 96
Figure 100: KMK Pinout Table 97
Figure 101: Prototype v1.0 99
Figure 102: Digital I/O Inputs on Prototype v1.0 100
Figure 103: Pin Connection on Prototype v1.0 100
Figure 104: Prototype PCB with PCA10056 102
Figure 105: Final Assembled Prototype 103
Figure 106: Overall Procedure Layout for Prototype 103
Figure 107: Software Versions Used During Testing and Prototyping 104
Figure 108: Procedure for Testing AutoHotkey 105
Figure 109: Running Initial Hello World AutoHotkey Script by Double Clicking 105
Figure 110: Prototype AutoHotkey Script 106
Figure 111: Error Results attempting to run AutoHotKey.exe in AHK Program Files 107
Figure 112: Runtime Goals Regarding Installation of Python and AutoHotKey 108
Figure 113: Test Results converting files into executables 109
Figure 114: Advantages Found Converting Python Code into Executable File 110
Figure 115: Configuration Settings to Convert Python Code into Executable - “Auto Py to Exe” 111
Figure 116: Generated Files - Converting Python Program to Executable 112

Figure 117: Project Development Budget 113
Figure 118: Bill of Materials 114
Figure 119: Project Timeline 115

# 1 Introduction

## 1.1 Motivation

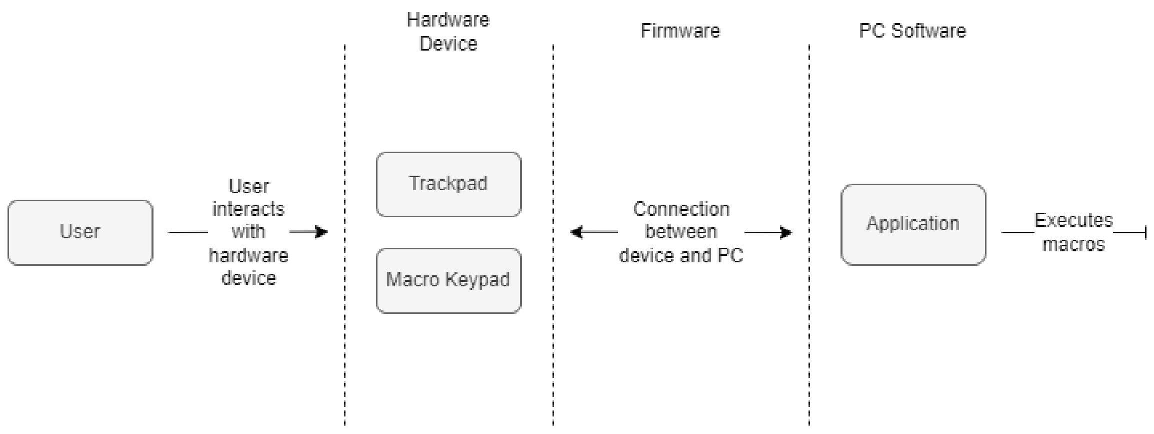
On the market today, there exists a subset of a common computer peripheral meant to boost a basic end-user's productivity. This product is the programmable mouse. The concept is simple; it is a computer mouse that contains several extra buttons that the user can map to any shortcut or command which he/she chooses. The added convenience of these extra buttons means that a user can save valuable time when performing common, repetitive tasks.

Currently, this type of device only exists for the mouse, but not for the trackpad which has millions of users every day. This is where our project comes in. Our purpose in creating the Programmable Trackpad is to bring this type of technology to the trackpad, for users who prefer to use a trackpad over a mouse. Our trackpad will contain a suite of macro keys and rotary encoders, all of which end users can program themselves. With these, trackpad users will be granted the same convenience and functionality as programmable mouse users in a compact and ergonomic package. This device is intended to completely replace the default trackpad on a traditional laptop.

## 1.2 Problem Statement

The guiding principle of our development process is this: create a system which, when connected to a PC, can act as a trackpad and a macro keypad. In order to accomplish this, we will create a hardware device with input systems (trackpad and keypad) and a software application which the PC will use to interpret the hardware's output. Additionally, the device's firmware will manage communication between the device and the PC. The requirements for these systems, as well as the technology used to accomplish these tasks, will be expanded upon in further sections. At a basic conceptual level, however, the system can be described using the diagram below.

*Figure 1: System Concept*



# 2 Project Description

## 2.1 Goals and Objectives

The major goal of this project is to take a task from people's daily use of their PCs and attempt to create an overall convenience and improved efficiency in their work. Whether they use their computer for personal or work use, this project strives to reduce the steps taken in repetitive and common tasks done on the computer. The project aims to create an external trackpad device that exists outside of the computer that is compact and portable, yet also purposeful and meaningful in its functionality.

To reach the goals of this project, there are certain objectives that need to be met based on specific design choices in hardware and software. The following table lays out the general goals that guided our team's design process, as well as the specific objectives that we accomplished as a means to reach each goal.

*Figure 2: Project Goals and Objectives*

| **Goal** | **Objective** (how we achieved said goal) |
|---|---|
| Reduce common and repetitive tasks | Add buttons with macro key capabilities that are programmable. |
| Convenient and Ergonomic | Manages and runs all Hotkey macros within a Graphical User Interface and has them operational even after termination of the application. |
| Ergonomic | Supports ambidextrous users. |
| Low Learning Curve | - Application with a user-friendly interface to program macro keys.<br>- Only have to run one executable file, the user doesn't need Python or AutoHotKey installed. |
| Customizable for user | **Hardware** - Ability to easily remove keys to the user's liking.<br>**Software** - Application is be able to create and store to run on the device |

## 2.2 Function

In order to accomplish the goals of the project, our system will include many interrelated functions. On a high level, these functions can be summarized with a list of the interfaces with which the user will interact. While the inner workings of the device will be expanded upon in further sections, the following table is a general look at the device from a user's perspective.

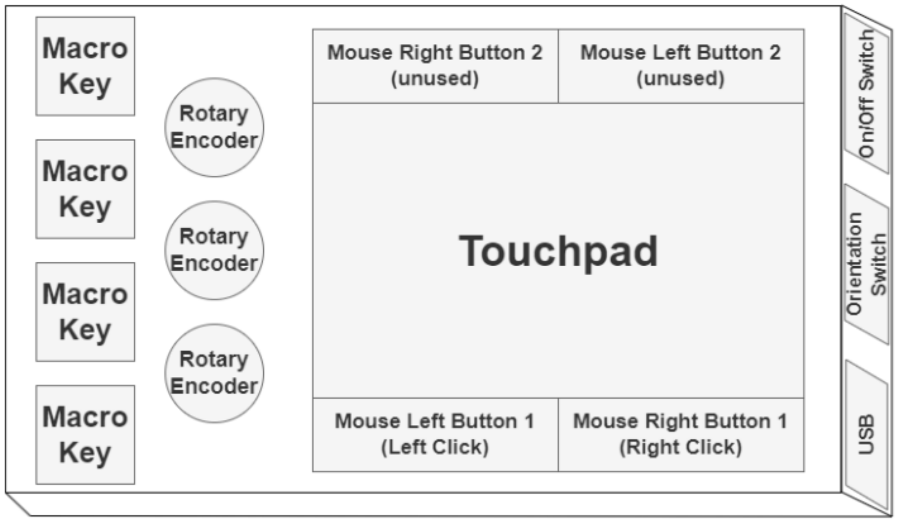
*Figure 3: Project Functions*

| Function | Description |
|---|---|
| 4 Mechanical Keys | Capable of macro and keybind function. |
| 3 Rotary Encoders | Capable of audio mixer, adjusting windows, etc. functions, (per-application functionality). |
| USB Connection | For charging the battery or having a wired connection. |
| Bluetooth Connection | Main connection for using the device. |
| Touchpad/Trackpad | Mouse replacement offering ergonomics. |
| 4 Mouse Buttons | Availability changes based on dominant hand usage. |
| Power Switch | Turn the device on or off. |
| Application User Interface | Main ability to program and customize hardware keys with macros |

### 2.2.1 Device Layout

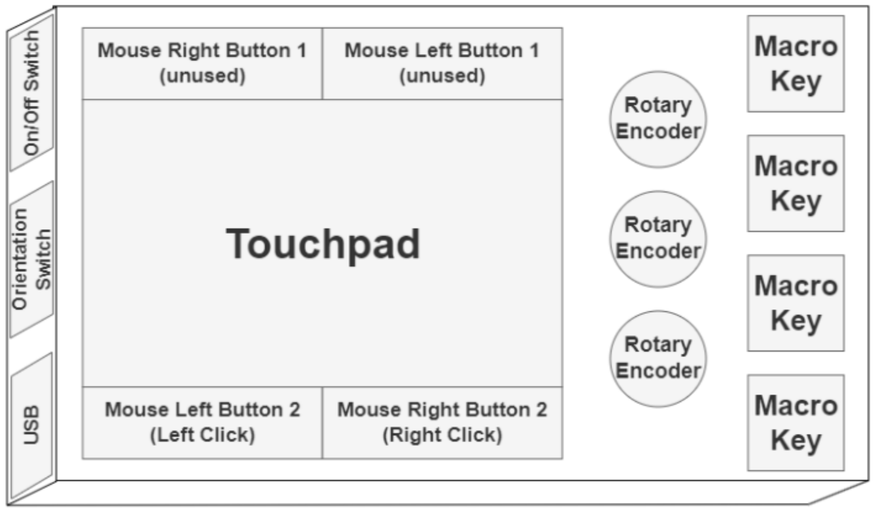
The intended way to use the Programmable Trackpad is for the user to place the device on his/her desk next to the PC’s keyboard. For right-handed users, the Programmable Trackpad will be placed on the right of the keyboard, while left-handed users will place the device to the left of the keyboard. The following diagram shows how the device’s components are arranged for right-handed users.

*Figure 4: Device Layout for Right-handed Users*


Macro Key
Macro Key
Macro Key
Macro Key
Rotary Encoder
Rotary Encoder
Rotary Encoder
Mouse Right Button 2 (unused)
Mouse Left Button 2 (unused)
Touchpad
Mouse Left Button 1 (Left Click)
Mouse Right Button 1 (Right Click)
On/Off Switch
Orientation Switch
USB

For left-handed users, the same device can be used slightly differently. Rotating the Programmable Trackpad 180 degrees, the device's layout becomes as shown in the diagram below. Note that the mouse click buttons that were unused in the previous diagram become the primary mouse click buttons in this layout.

*Figure 5: Device Layout for Left-handed Users*


On/Off Switch
Orientation Switch
USB
Mouse Right Button 1 (unused)
Mouse Left Button 1 (unused)
Touchpad
Mouse Left Button 2 (Left Click)
Mouse Right Button 2 (Right Click)
Rotary Encoder
Rotary Encoder
Rotary Encoder
Macro Key
Macro Key
Macro Key
Macro Key

Note that the USB connector is on the side of the device. Standard PC peripherals will often position the USB connector or wire on the back of the device so that the wire will lead directly to the PC, and the user will have minimal contact with it. In the case of this device, positioning the USB connector on the back would be impossible because the back side of the device becomes the front side of the device when the orientation is switched.

### 2.2.2 Orientation Switching

The previous section alluded to the ability of the device to switch between modes for left-handed and right-handed users. For the sake of the device's layout, all the user needs to do is rotate the device 180 degrees to change the orientation. If this is done, however, the trackpad inputs will be upside-down, and it will be impossible to use the trackpad regularly. In order to solve this problem, there is a physical sliding switch on the side of the Programmable Trackpad that toggles between left-handed mode and right-handed mode. When switched from one position to another, the trackpad will automatically adjust itself through firmware, and the PC will not read the inputs upside-down.

The orientation switch is positioned on the side of the Programmable Trackpad so that it is not likely to be in the user's way at any time. Note that the power switch and orientation switch are specifically positioned on the far side of the device from the touchpad (adjacent to the USB connector). This is because the user's wrist tends to rest on the near side of the touchpad. This setup ensures that the user will not be bothered by the switches, and also that the user will not accidentally toggle the switches during ordinary use.

## 2.3 Requirements

The requirements for this project should highlight the system's technical needs, which will determine our estimated budget for the overall design. The specifications we are aiming to achieve support our aforementioned goals of convenience, flexibility, and programmability.

*Figure 6: Project Requirements*

| Requirement | Justification |
|---|---|
| Device dimensions 5” x 5” x 2” | Device should be portable and have a small form factor accentuating ergonomic qualities. |
| Device weight should be ≤ 1 lb | Device should be light and portable. |
| Device trackpad latency ≤ 48 ms | Device should have a low latency for an accurate and precise experience for the user. |
| Hot-swappable switches | User customization. |
| 4 mechanical switch inserts | Optimal amount of macro keys to provide efficiency. |
| 3 rotary encoders | Users should be able to control audio and customize sliders based on programmability. |
| USB connection | Users should have a usable connection when Bluetooth doesn’t work. |
| Bluetooth connection | Users should have wireless connection for ease of use and less cable clutter. |
| Battery lifetime should be ≥ 10 hours of average usage | Users should be able to use the device during a full day of wireless utilization without having to plug it in or charging. |
| Battery charging time should be < 3 hours | Charging does not need to be fast because it can be used while charging. However, it is reasonable to expect it to fully charge within a certain time frame. |

## 2.4 Constraints and Standards

The constraints and standards highlight the implied limitations caused by budget, environment, or market standards.

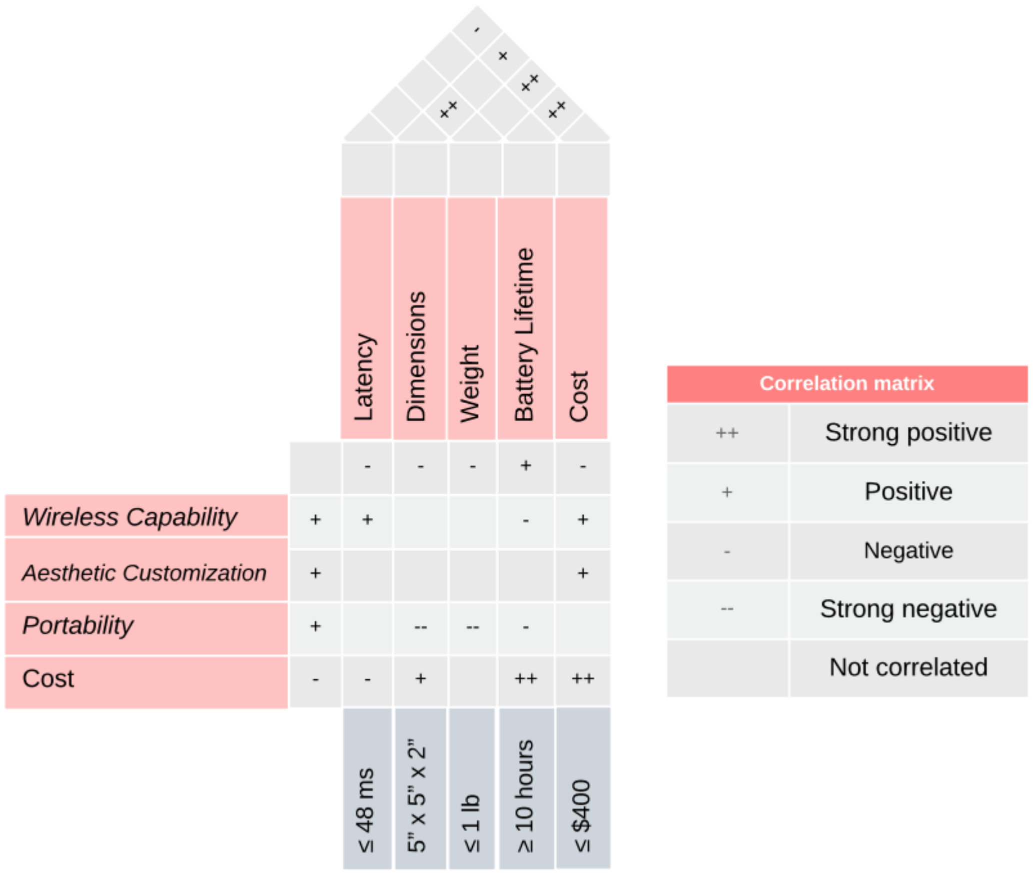
*Figure 7: Project Constraints and Standards*

| Constraint/Standard | Reasoning |
|---|---|
| Micro USB Type-B | Standard in consumer technology for wired connections due to its fast transfer speeds and power efficiency. |
| Bluetooth 5.0-5.3 | Low bandwidth, reliable and fast speeds over air. |
| Human Interface Device (HID) | Standard in modern operating systems for mouse/keyboard communication. |
| JTAG | The standard for uploading firmware to ARM devices. |
| Serial Wire Debug (SWD) | A specification within the JTAG standard that is particularly compatible with our project's hardware. |
| … ≤ 400 for development budget | Due to limited budgeting of the group, we must keep the development budget low for the whole project. |
| Total budget for the project should not exceed $1000 | We wanted the device to be budget friendly while also having the development of the project be efficient and cost effective based on other market items. |

## 2.5 House of Quality

The following diagram is a house of quality, a graphic representation of the various requirements of our design and how each one interacts with each other. Plus signs represent positive correlation between quantities; minus signs represent negative correlation. Each column represents a requirement of the design process; each row represents a consumer requirement.

*Figure 8: House of Quality*



| | | Latency | Dimensions | Weight | Battery Lifetime | Cost |
|---|---|---|---|---|---|---|
| | | - | - | - | + | - |
| *Wireless Capability* | + | + | | | - | + |
| *Aesthetic Customization* | + | | | | | + |
| *Portability* | + | | -- | -- | - | |
| Cost | - | - | + | | ++ | ++ |
| | | ≤ 48 ms | 5" x 5" x 2" | ≤ 1 lb | ≥ 10 hours | ≤ $400 |

| Correlation matrix | |
|---|---|
| ++ | Strong positive |
| + | Positive |
| - | Negative |
| -- | Strong negative |
| | Not correlated |

This graphic, though not a set of immutable rules to follow, can serve as a guideline for our design process. The most noteworthy tradeoffs to examine in the design process are those that affect the device's portability and development costs. As the device grows in size, it becomes less portable, which is a strong negative for the product. As the battery lifetime increases, the cost of the device greatly increases, which is another negative for the product. Ideally, we would find a balance between high-quality electronics and low-cost development, and we would use the numerical data suggested above as the baseline for this decision-making process.

# 3 Technology Investigation

The first step in the development process for our project was researching the existing technology relevant to the product we are creating. This section will detail our findings from this research process.

Initial research was focused on existing consumer products comparable to our vision. This research is important because it provides context for the niche that our product fills. We were able to draw inspiration from what existing products on the market succeed at doing, as well as what they fail to do.

Further research was focused on the hardware and software technology that our project can leverage in order to fulfill its purpose. This section contains a review of hardware devices, computer chips, firmware programming systems, and software packages that may be useful in the development of our system.

## 3.1 Market Analysis

Millions of PC users use trackpads every day, largely due to their standard presence in laptop computer design. Many consumers who use laptop computers at stationary desks, however, still prefer to use their trackpads over the traditional mouse. Mac users, in particular, commonly prefer trackpads to mice. This trend is due to the prevalence of Apple's Magic Trackpad, a relatively high-end external trackpad. It is safe to conclude that there is a market for external trackpads among Mac users, and there would likely be a similar market among Windows users if more convenient options were commonly available.

As mentioned previously in this document, the programmable mouse is the market standard solution to the goals prescribed in our project. The following table shows a few popular options currently available. Each of these mice interfaces with a PC application to program the various buttons on the device.

*Figure 9: Market Programmable Mice*

| Name | Image | Price |
|---|---|---|
| Logitech MX Master[1] | | $99.99 |
| Razer Deathadder v2[2] | | $69.99 |
| Microsoft Surface Precision Mouse[3] | | $99.99 |

A broad description of the market niche we intended to fulfill was the market for any peripheral trackpad device that interfaces with PC software to facilitate custom button inputs. The following is a list of the devices currently on the market that are most comparable to our own.

*Figure 10: Market Trackpads*

| **Name** | **Image** | **Price** | **Similarities to our design** | **Differences from our design** |
|---|---|---|---|---|
| Apple Magic Trackpad[4] | | $129.99 | Fits conveniently on a desk. | No physical customizable buttons. |
| Mousetrapper Advance 2.0[5] | | $200-$300 (Not currently available for purchase from manufacturer) | Fully programmable physical buttons, ergonomic. | Too large to be comparable to a mouse, wired only. |
| Keymecher MANO-703 UB[6] | | $39.99 | Includes macro buttons, fits conveniently on a desk. | Macro buttons are hard-coded for specific purposes (not customizable). |

While there exist many comparable products, there is no single device that meets all of the goals for our project currently on the market.

## 3.2 Hardware

The following section is an overview of the various hardware technologies that were incorporated in the design of the Programmable Trackpad.

### 3.2.1 Power

The basic power requirements of the system are simple. It is designed to be wireless, therefore it must be powered by a battery. Because it is expected to be used over a period of years, the battery must be rechargeable. Since the device can communicate with a PC over wired USB, it is also reasonable to design it so that the battery charges over the same USB connection as the data transmission. Thus, it must remain functional while charging. Finally, charging should be automatic. As soon as the device is plugged into a power source, the battery should begin charging up to its charge limit.

#### 3.2.1.1 USB Power Standard

Different USB standards vary slightly in the amount of power supplied over the power wires, but they all supply 5 volts. Typical USB ports in a computer supply a maximum of 500 mA. This figure can be used as a maximum for our purposes because in typical use, the trackpad will be plugged into a standard computer USB port.

The current supplied over USB is irrelevant to the functionality of the device because the USB cable will only be supplying power to the battery, not to the computer components of the device. Therefore, the battery's maximum current needs only be sufficient to supply the necessary current to all components. The current supplied over USB instead is used to determine how quickly the battery can be charged.

Most components used in this device are rated for lower than 5 volts, so it is reasonable to use a battery that supplies lower than 5 volts. Since USB power is fixed at 5 volts, the charging system must account for this and lower the voltage at which it charges the battery.

#### 3.2.1.2 Power Calculations

The following equation can be used to determine the amount of time a battery will last starting from a full charge, going to a full discharge.

*Figure 11: Discharge Time Equation*

$$T = \frac{Q}{\Sigma\, i_n}$$

$$T = Amount\ of\ time\ for\ the\ battery\ to\ fully\ discharge\ (h)$$
$$Q = Capacity\ of\ battery\ (Ah)$$
$$\Sigma\, i_n = The\ sum\ of\ typical\ currents\ drawn\ by\ each\ component\ n\ in\ the\ device\ (A)$$

Conversely, the following equation can be used to determine the amount of time it will take for a battery to charge fully from empty assuming that it is being constantly supplied 500 mA.

*Figure 12: Charge Time Equation*

$$T = \frac{Q}{0.5\, A}$$

Note that in practice, "full charge" and "full discharge" will not be exactly as rated by the manufacturer. The age of the battery and conditions of operation will affect these numbers. These equations should only be used to determine a baseline for the charge and discharge time.

### 3.2.1.3 Battery Technology

The key attributes that we looked for in a battery are that it is rechargeable, it has built-in circuit protection, and that it is low-profile. High capacity and high current output are not priorities because the device consumes relatively little power, and it is reasonable to assume that it will be charged regularly. The following table compares various technologies of rechargeable batteries.

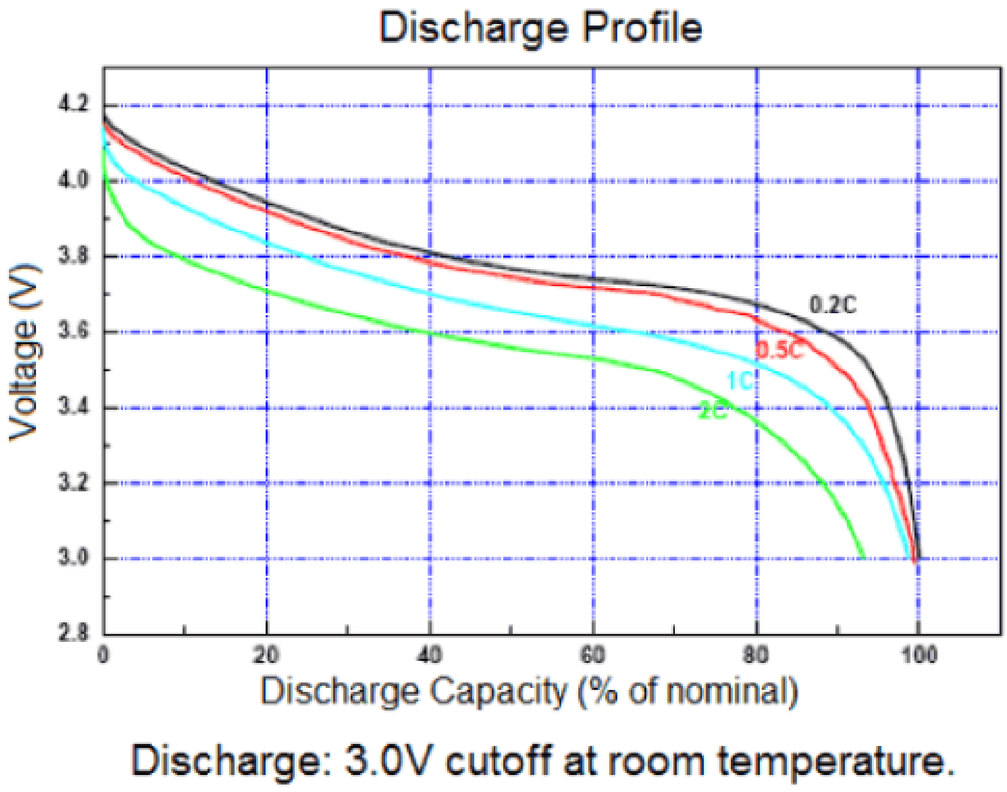
*Figure 13: Battery Technology Comparison*

| **Technology** | **Size** | **Capacity** | **Efficiency** | **Cost** | **Protection** |
|---|---|---|---|---|---|
| Lead-acid | Largest | Medium | Medium | Lowest | None |
| Lithium Ion | Medium | Medium | Best | Highest | None |
| Lithium Polymer | Smallest | Medium | Best | Highest | Built-in |
| Nickel Metal Hydride | Medium | Best | Medium | Medium | None |

Based on the nature of our device, it is imperative that the battery is as low-profile as possible. The only commercially available batteries that suit our purposes are lithium polymer (LiPo) batteries.

Consumer LiPo batteries have a maximum voltage of 4.2 volts and an average working voltage of 3.7 volts. It is possible to discharge them beyond 3.7 volts, but the built-in protection cuts off current flow at 3.0 volts. The following table shows the relationship between discharge and voltage in LiPo batteries.

*Figure 14: Discharge Profile of LiPo Batteries[7]*



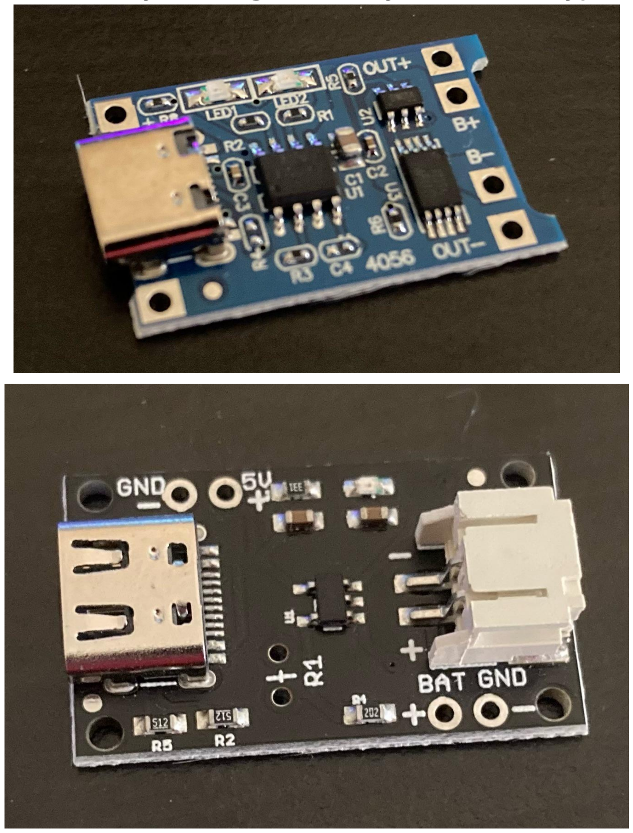
### 3.2.1.4 Battery Management Boards

For prototyping, we procured two USB charging boards intended to interface with lithium batteries. Both purport to offer circuit protection as well as a USB Type-C female connector. Neither board includes a breakout option for the data pins, which will be necessary for the trackpad to transmit data to the PC.

The first battery management option uses the TP4056 chip, which provides constant current (up to 1 amp) and constant voltage at 4.2 volts[8]. This voltage value is the typical upper limit for LiPo batteries, so it should protect our battery. The constant current value can be adjusted by adjusting the resistance on pin 2 of the chip. In the pre-built board that we used for testing, this current value was set to the maximum value of 1 amp.

The second battery management board uses the MCP7383X-2 chip, which provides constant current (up to 500 mA) and constant voltage of 4.2 volts[9]. Like the previous chip, this one's constant current value is adjusted by adjusting the resistance on pin 5, and the pre-built board uses the maximum current.

Figure 15: Battery Management System Prototype Boards



While the TP4056 does allow for higher current (and consequently faster charging), the MCP7383X-2 is simpler and has better documentation. We will be using the latter in our design.

### 3.2.1.5 Voltage Converter

As has been noted in previous sections, USB operates at 5 volts, and LiPo batteries operate between 3.7 and 4.2 volts. In order to compensate for this difference, a battery management system is necessary to convert USB voltage to battery voltage. The onboard electronic components of the Programmable Trackpad are rated for 3.3 volts, which is significantly lower than the voltage of the battery. Therefore, it is also necessary to convert the battery's voltage to 3.3 volts before powering the device's electronics. A simple voltage converter can be connected to the device to accomplish this.

Buck converters with variable output voltage are very common on the market. These devices accept an input voltage on one pin and output a voltage on another pin. The output pin must be connected through resistors to achieve the desired voltage. This would be a valid solution for our device to achieve 3.3 volts, but since 3.3 volts is a common voltage level anyway, it would be simpler to use a buck converter with fixed output voltage of 3.3 volts. The following table provides an overview of some simple buck converters that could suit the needs of this project.

*Figure 16: Voltage Converter Comparison*

| Part number | Input Voltage | Current Output | Number of extra parts (inductors and capacitors) needed per documentation |
|---|---|---|---|
| TPS62203[10] | 2.5-6 volts | 300 mA | 3 |
| AP63203[11] | 3.8-32 volts | 2000 mA | 4 |
| LM3671[12] | 2.7-5.5 volts | 600 mA | 3 |

All three of these converters are fixed-voltage (3.3 volts) versions of generic buck converters. Each one has thorough documentation including example circuits demonstrating the proper way to set up the converter with capacitors and inductors. While the AP63203 has the best current output at 2000 mA, its input voltage is rated for a minimum 3.8 volts, which is greater than the minimum voltage of the device's battery (3.7 volts). The LM3671, on the other hand, is specifically designed to work with lithium batteries, so its input voltage aligns with the voltage of the device's battery. For this reason, our team has opted to use the LM3671 in our design.

For prototyping purposes, our team acquired a simple LM3671 breakout board produced by Adafruit[13]. This board's electrical schematic is available online and can be used to inform the design of our own device.

In addition to the 3.3 volt buck converter that will be necessary to convert the battery voltage for use by the electronics, another consideration during development was the possibility that the device might require a 5 volt boost converter to ensure a steady voltage from the PC to the USB devices. The need for this device is elaborated upon in the prototyping section. Some popular options for 5 volt converters are the TPS61023 and the ME2108. The following table compares these two options.

*Figure 17: 5 Volt Converter Comparison*

| Chip | Input Voltage | Number of external components required |
|---|---|---|
| TPS61023 | 0.5-5.5 volts | 3 |
| ME2108 | 0.9-6.5 volts | 4 |

Because the input voltage for this chip would be coming directly from the PC's USB port, it is expected that these voltage ranges would be acceptable. The PC's USB ports are intended to support 5 volts at all times, so they should never dip far below this level. The complexity of implementing these chips is comparable, as well. Therefore, there are no significant differences between the chips, and either one would be acceptable for prototyping a solution in our device.

In practice, it was experimentally determined that the 5 volt boost converter is not necessary to make our design function. As such, neither the TPS61023 nor the ME2108 was used.

#### 3.2.1.6 Power Switch

Because the device can operate wirelessly, it must have a method for the user to turn it on or off. This will be solved with a simple on/off switch that will connect the battery to the rest of the electronics in the device. Such a switch can be seen below.

*Figure 18: Generic Power Switch*

Because the voltage converter represents a constant draw on the battery, it is important that the converter is fully disconnected from the battery when the device is off. Therefore, the power switch connects the battery's positive terminal to the voltage converter when closed (in the ON position). The battery should not, however, be disconnected from the battery management system when the device is turned off. The battery should always be connected to the battery management system so that the battery can be charged while the device is off. In this case, because the battery is connected to the battery management system, it charges, but because it is not connected to the voltage converter, the device's computer electronics will remain off.

### 3.2.2 Touchpad

While conducting research, we came across 3 possible options for touch pads. Each of these three options had unique functionalities and interfaces so determining *exactly* what we wanted from a touchpad was critical. The 3 possibilities were Adafruit's Resistive Touch Screen, AliExpress's RGB TFT Touch LCD Display module, and DFRobot's Capacitive Touch Kit for Arduino.

The initial search criteria was any resistive or capacitive touchpad that could be integrated into our own printed circuit board design. That is to say, we wanted a touchpad that was modular.

In narrowing our options, our first consideration was the dimensions of the touchpad. One of the key features of our project is that it should be compact and easy to fit on a users desk. Therefore, the touchpad should not be excessively large, but still big enough to be used comfortably by an average sized adult. In our initial design render, we decided that an optimal size for the touchpad itself would be around 3.7” diagonally, so that is the metric we used for evaluation.

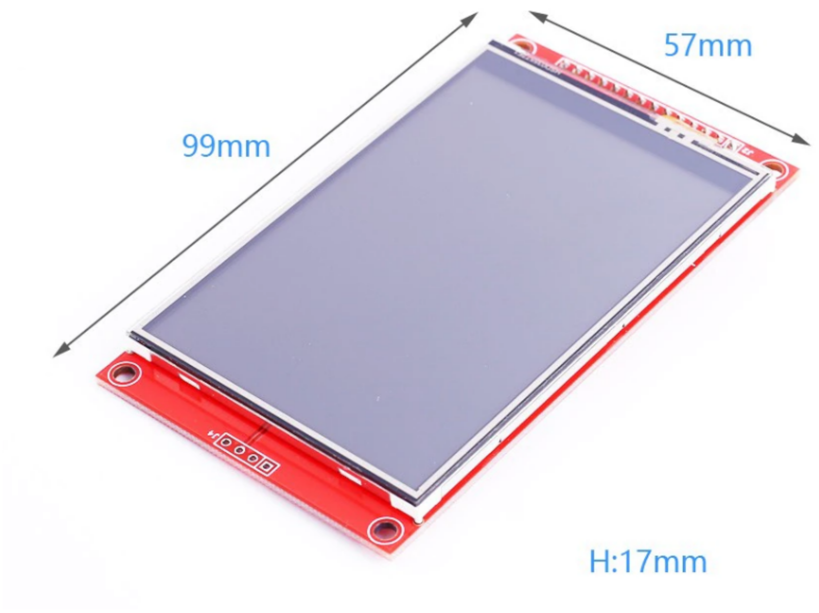
Our second consideration was power drain. In its completed state, the Programmable Trackpad should have a wireless operation mode. With this, it's extremely important to take into account the power drain of each individual component. If we choose a touchpad that draws a large amount of power, then the battery life of the device will be shortened.

Our third consideration was ease of integration into our printed circuit board design. Different touchpads interface in a variety of different ways. The number and position of pins on the touchpad influence how we design not only the printed circuit board, but the entire housing of the device as well. Pins layouts could lead to drastic design changes.

#### 3.2.2.1 AliExpress RGB TFT Touch LCD Display Module

Our first candidate was the AliExpress RGB TFT Touch LCD Display Module[14]. This screen is unique in that it is not merely a touchpad, but an LCD touchscreen. The screen is 16-bit and has a resolution of 480x320. It interfaces via 9 pins on its board, communicates via SPI, and uses 3.3V for its logic inputs. This option allows us to graphically display information to the user on the screen, in addition to acting as a touchpad. However, it is important to note that including this screen means that this unit will draw more power and deplete the battery faster. While we did not originally intend for the Programmable Trackpad to have an integrated screen, we can envision the many uses for it. As for the touchpad capabilities, the usable area will be approximately 4.5 inches across diagonally. This is a little larger than we would like, and would potentially require 3D printing a larger housing for the device than we originally intended. This would harm our efforts to make the device compact and desk-friendly. In addition, this touchpad would be medium in terms of ease of integration. All we would have to do is account for 9 through-holes on our PCB design so that we can solder the header pins onto it. We would also have to ensure that through-holes are in a central position relative to the whole device so that the touchpad will be positioned such that it can be used properly. This option would also be on the pricier side. It comes out to approximately $15. This touchpad is pictured directly below.

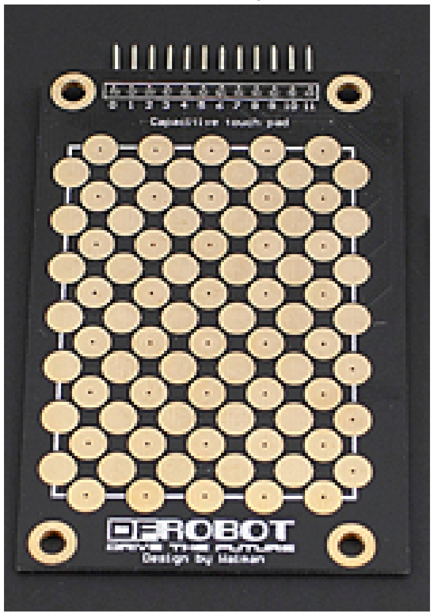
*Figure 19: AliExpress RGB TFT Touch LCD Display Module*



### 3.2.2.2 DFRobot Capacitive Touch Kit for Arduino

The next candidate that was considered was the DFRobot Capacitive Touch Kit for Arduino[15]. This touchpad comes as part of a larger kit, and is strictly a capacitive touchpad. Since it comes as part of a larger kit, it is the most expensive option that we have reviewed thus far. It comes out to approximately $20. In the kit are other neat components, but none of them are relevant to this project. This touchpad has a functional area of 3.2 inches diagonally. This is slightly below our target of 3.7 inches diagonally. We believe that any size below this will lead to difficulties and inconveniences when a user goes to operate the touchpad, so we would prefer a bigger size than that. We also do not foresee this touchpad to be a large power drainer. This touchpad only has to poll for finger touches. This touchpad is also medium difficulty in terms of PCB integration. It has a total of 12 pins that need to be soldered onto the board. Similar to the AliExpress option, we would have to incorporate through holes on our PCB design, then solder the touchpad onto our PCB afterwards. This touchpad is pictured below.

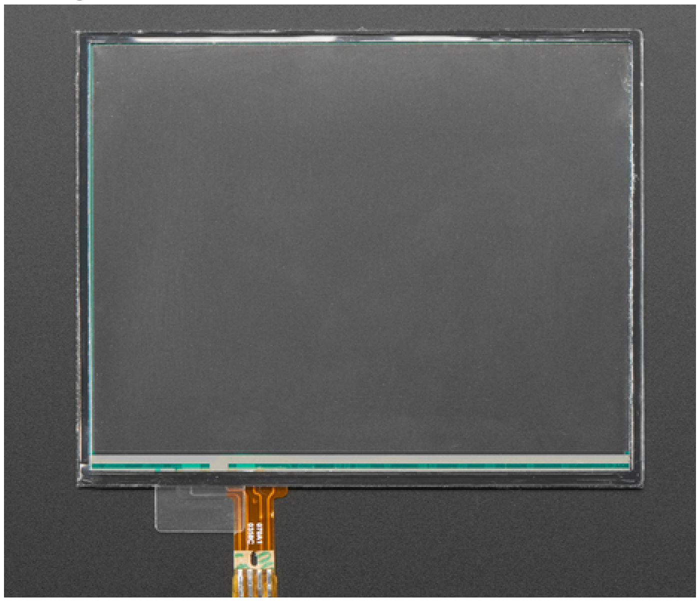
*Figure 20: DFRobot Capacitive Touch Kit*



### 3.2.2.3 Adafruit Resistive Touch Screen

Another touchpad that was considered was the Adafruit Resistive Touch Screen[16]. This touchpad is by far the cheapest option that we have encountered. This part is approximately $6. This option is a resistive touchpad, as the name implies. This touchpad has a functional area of 3.7 inches diagonal, which matches our initial design renders exactly. In terms of power drain, this touchpad will drain the least amount of power on a battery operated system. This candidate interfaces via an FFC (flat flex cable) with 4 pins. To integrate this module into our PCB, we would have to solder on an FFC socket onto it. Given the length of the cable, placement should be no issue. Therefore, this touchpad would be very easy to integrate. This touchpad is pictured below.

Figure 21: Adafruit Resistive Touch Screen



### 3.2.2.4 Touchpad Comparison

Below is a table that compares and contrasts each of the three touchpads that were considered during our technology research

Figure 22: Touchpad Comparison

| **Touchpad** | **Size** | **Price** | **Integration** | **LCD Display** | **Power Drain** |
|---|---|---|---|---|---|
| AliExpress RGB TFT Touch LCD Display Module | 4.5” Diagonal | $15 | 9 Through-Hole Pins | Yes | High |
| DFRobot Capacitive Touch Kit for Arduino | 3.2” Diagonal | $20 | 12 Through-Hole Pins | No | Medium-Low |
| Adafruit Resistive Touch Screen | 3.7” Diagonal | $6 | 4 Pin Flat Flex Cable | No | Low |

Upon review of these choices, we ended up choosing the Adafruit Resistive Touch Screen. Of the three, this touchpad was exactly the size that we were looking for, and came at the cheapest price point. Given its barebones nature, we also anticipate this touchpad to have the lowest power drain of the group which lends itself well to wireless devices.

## 3.2.3 Touchpad Controllers

A resistive touchpad controller chip is a possible way of handling the inputs that will come in from the touchpad. These chips provide a way of accepting inputs from the user and delivering them to the microcontroller unit in a monitored way. Using one of these chips is also a good way to save on microcontroller pins. Using one of these chips requires two pins, while hooking up the touchpad directly to the microcontroller will require 4 pins.

### 3.2.3.1 AR1100 Chip

The AR1100 chip is a universal resistive touchpad controller, and it is an option that we had considered to process the touch inputs from our touchpad. This chip stood out to us because it is a very common and cheap way to interface with a touchpad. Not only this, there also exists a large amount of documentation and detailed circuit schematics for this board which will help with its integration into our printed circuit board. This chip also allows us to have the most flexibility in how we want to utilize the incoming data. The chip can be configured by downloadable software to be more or less sensitive, sample faster or slower, enter a sleep mode etc. After doing so, the AR1100 chip will store these configured settings in non-volatile memory so that it will not have to be re-configured every time it loses power, which is highly convenient for the user. This means that this chip will be configured once by us in the development process, and the end-user will not have to worry about these settings.

After conducting several experiments we decided that there was no added benefit in using the AR1100 in our design. All input processing is done via the firmware running on our MCU, so having a hardware element serves no purpose. Therefore, we eliminated this chip very early in the prototyping stage.

*Figure 23: AR1100 Chip[17]*

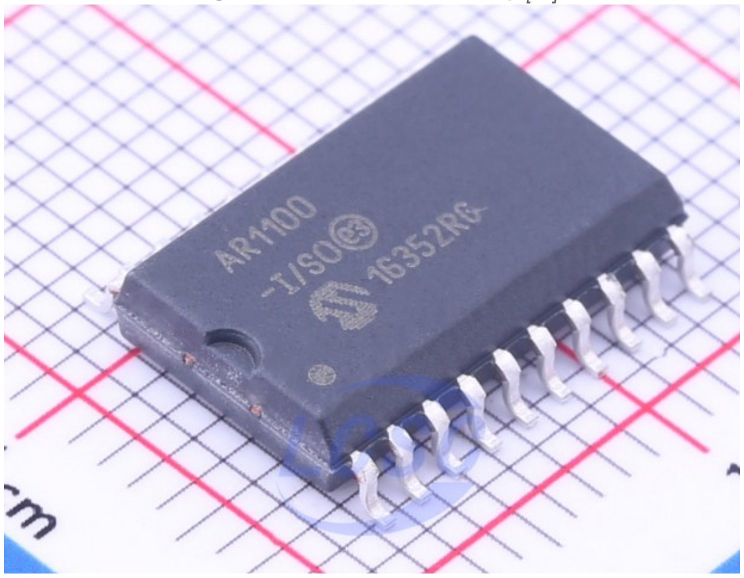
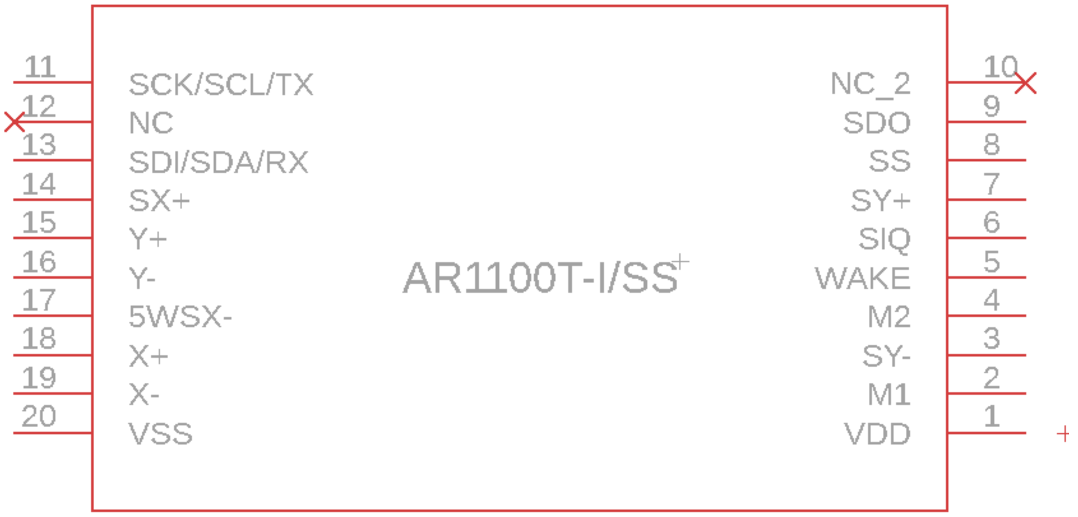


*Figure 24: AR1100 Schematic*



### 3.2.3.2 STMPE610

The STMPE610 chip is another option that we considered. The STMPE610 works in a similar manner to the AR1100 in that it will accept user inputs from the touchpad, process them, then route them elsewhere. For communication, this chip can use either I2C or SPI protocols, and operates on 1.8 - 3.3Vs. With this chip, there exists an option to purchase it on a breakout board first which would be used in the prototyping stage. If we decided that we wanted to move forward with this chip, we would then have to buy it individually and mount it to our PCB.

*Figure 25: STMPE610 Chip Pictured on a Breakout Board*

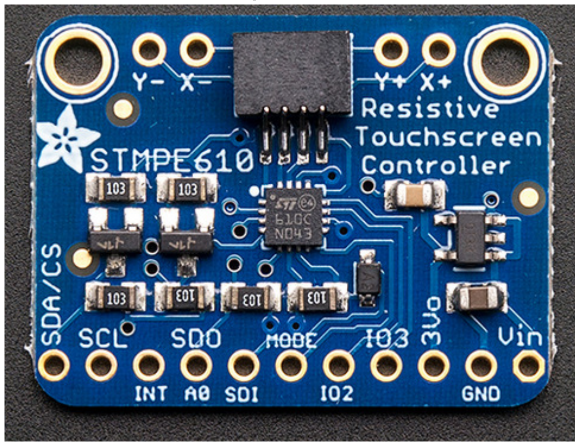
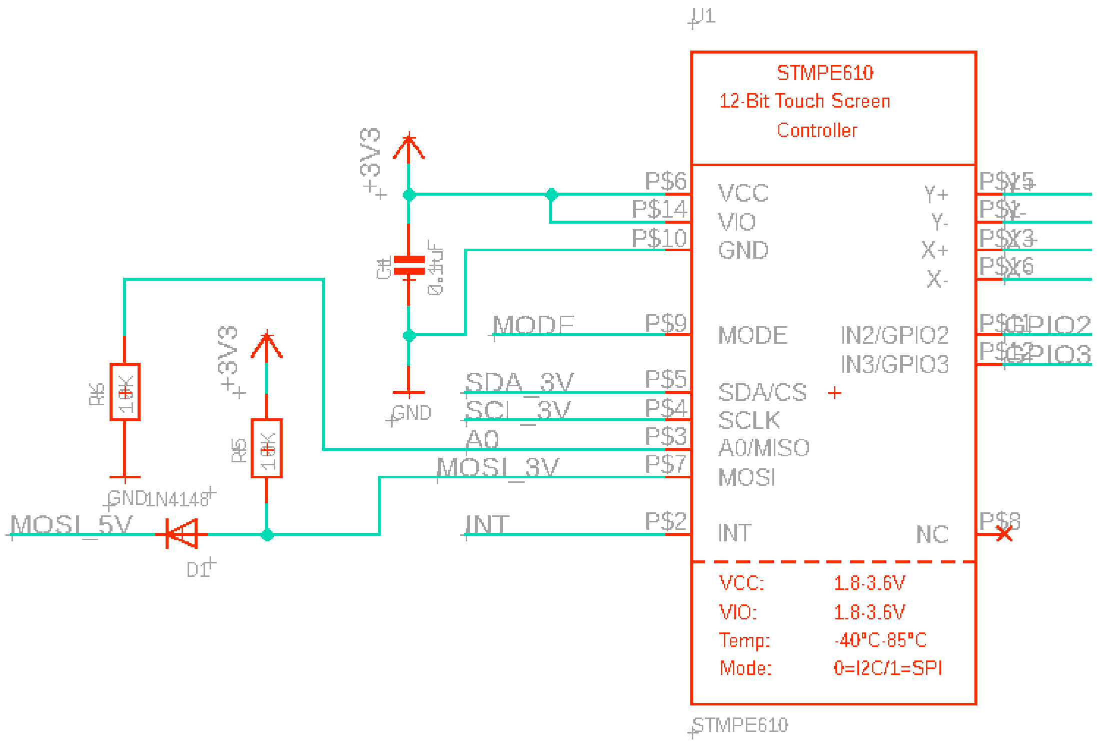


*Figure 26: STMPE610 Schematic*[18]



### 3.2.3.3 Touchpad Direct Connection

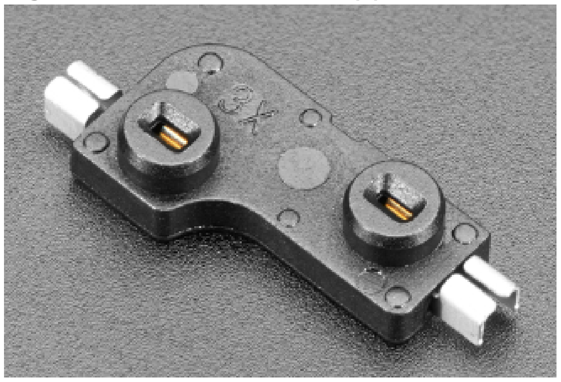
In lieu of the AR1100, we decided to connect the touchpad directly to the MCU. This was fairly straightforward, as the touchpad only required 4 connections, 2 analog and 2 analog or digital. The Touchscreen.h library for Arduino handled all of the heavy lifting when it came to translating finger presses on the touchpad to usable data. Upon a finger press, a data structure is instantiated that contains the X and Y coordinates of the press, as well as the pressure of the press. Using these values, it was simple to send commands to the PC to move the mouse to those X and Y values using the TinyUSB Mouse and Keyboard library.

## 3.2.4 Input Units

An important concern of the project is to provide for the consumer with options to change aesthetics and convenience. There is a market out there for custom keycaps, rotary knobs, switches, etc. and we wanted to reach out to those niche communities. One thing to do this was to implement hot-swappable sockets. This allows the consumer to switch up the mechanical switches with ease, eliminating the pain of soldering each switch. Another aspect that comes with the implementation of mechanical switches is being able to switch up the keycaps.

The common part used as a mechanical switch for keys on PCBs is known as the Kailh hot-swappable socket. Since this is standard, we have used it in our device.

*Figure 27: Kailh Hot-Swappable Socket*



For the default switches we opted for Boba U4Ts which offers the user an actuation force of 62 grams with a quiet gentle sound level when pressed. Of course, the user could change the switch depending on their preference. We went for these because for the general public, it would be the perfect balance between tactile feedback and quiet experience while also being a cheaper option in production. Some other cheaper options would include the more clicky switches, gateron blues, or the more quiet and linear feeling switches, the gateron yellows.

*Figure 28: Mechanical Switch Comparison*

| **Mechanical Switch** | **Actuation Force** | **Travel Distance** | **Behavior** | **Sound Level** | **Price (pack of 10)** |
|---|---|---|---|---|---|
| Gateron Red | 45 grams | 4 mm | linear | Quiet | $8.50 |
| Gateron Brown | 55 grams | 4 mm | tactile | Slightly audible | $8.50 |
| Gateron Blue | 60 grams | 4 mm | clicky | Very audible | $8.50 |
| Boba U4T | 62 and 68 grams options | 4 mm | tactile | Slightly audible | $11.99 |
| Gateron Black Ink v2 | 60 grams | 4 mm | linear | Slightly audible | $7.50 |
| Kailh Box Jade | 65 grams | 3.6 mm | clicky | Very audible | $10.99 |
| Durock Poms | 48 grams | 4 mm | tactile | Slightly Audible | $6.99 |
| Glorious Panda | 67 grams | 4 mm | tactile | Slightly more audible | $16.99 |
| Boba U4 Silents | 62-68 grams options | 4 mm | tactile | Silent | $23.99 |

As for the rotary encoders, there weren't that many to choose from since they all serve the same purpose and don't have many features to improve upon. One we came across was a 24-pulse encoder with detents and a very nice haptic feedback. This would allow us to know the current position of the encoder through the microcontroller using how many clicks are left from its current position. If we were to determine its rotational position, then a potentiometer would be a better option in this case, which could be a potential route we might encounter when prototyping, but, since the rotary encoder could be mapped to many different thing the user might choose, it would most likely be the better option. The main difference between the two is that the rotary encoders have a fully continuous rotation in either direction using digital signals whereas a potentiometer has a set direction in a clockwise or counter-clockwise direction using analog signals.

For the touchpad buttons we wanted them to be integrated within the parameters of the touchpad we had selected. Initially a route we had considered was using mechanical switches that fit the flat design of the touchpad, those would be using low profile mechanical switches, though we came to the conclusion we wanted the user to have a feeling of using a traditional touchpad. To do this, we decided on using some tactile switch buttons where they would be placed below some sort of metal plate similar to a traditional touchpad and it has some flex to press the button but still have the structure to hold itself. These tactile switch buttons range in different styles and tactile feedback and we needed one that would reach up to the elevated touchpad while also not having the possibility of being accidentally pressed.

*Figure 29: Generic Rotary Encoder and Tactile Switch Button*

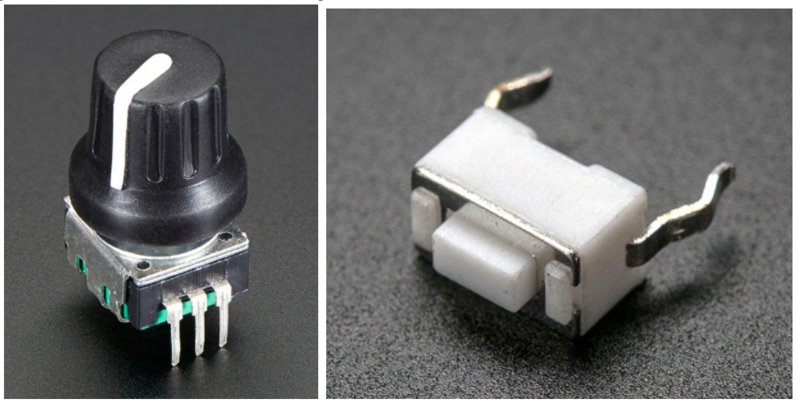
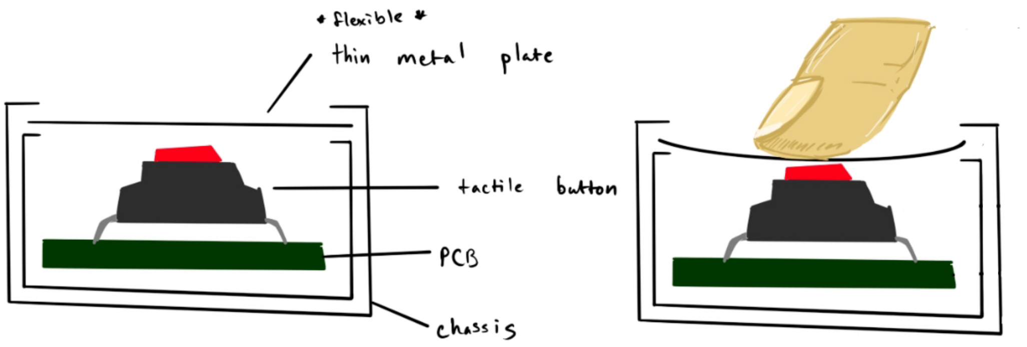
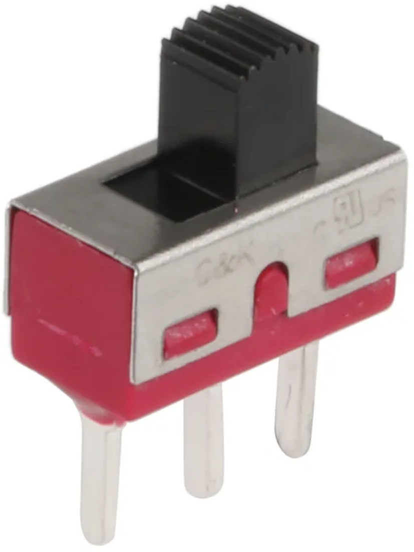


*Figure 30: Mouse Button Implementation Sketch*



The last input device included on the Programmable Trackpad is the orientation switch. This is a physical switch that the user can toggle between right-handed-mode and left-handed mode. Any simple toggle switch can be used for this purpose. It must be located on the outside of the hardware so that the user can access it.

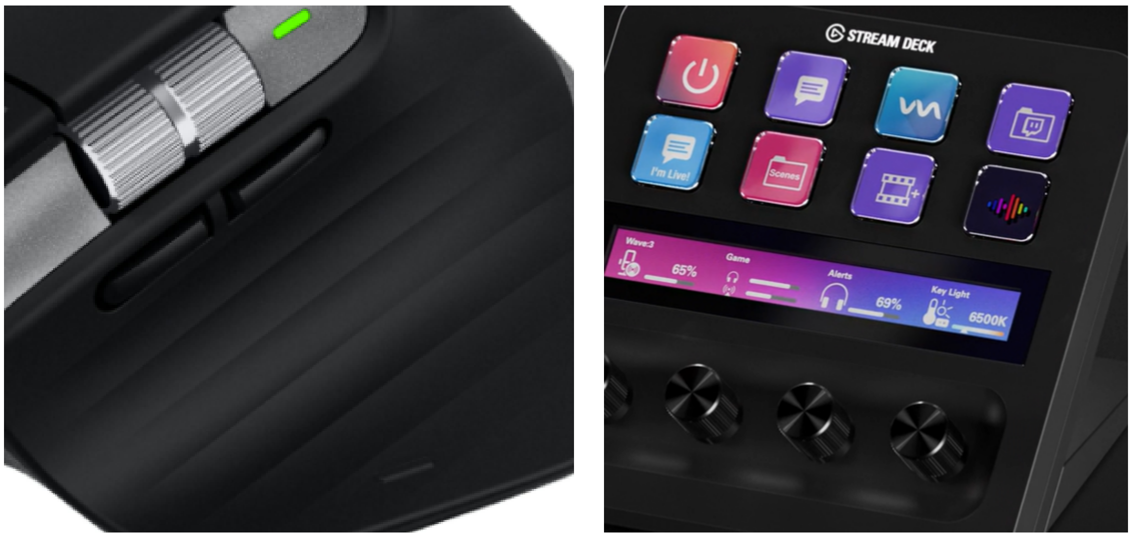
*Figure 31: Orientation Switch*



### 3.2.4.1 Stretching The Objective

The macropad industry is quite abundant in the market and it’s very obtainable to the average consumer. We eventually wanted to offer a different type of experience in efficiency and convenience for the users. On paper there’s not much you can change with the inputs but we have seen things like Logitech or Elgato that integrate some type of convenience for the user. These examples can be seen below with Logitech’s MX Master 3 and the hidden and convenient button on the thumb rest, and with the newly released stream deck with rotary encoders and a customizable LCD screen and haptic LCD screen buttons.

*Figure 32: Stretch Goal References for Input Devices*



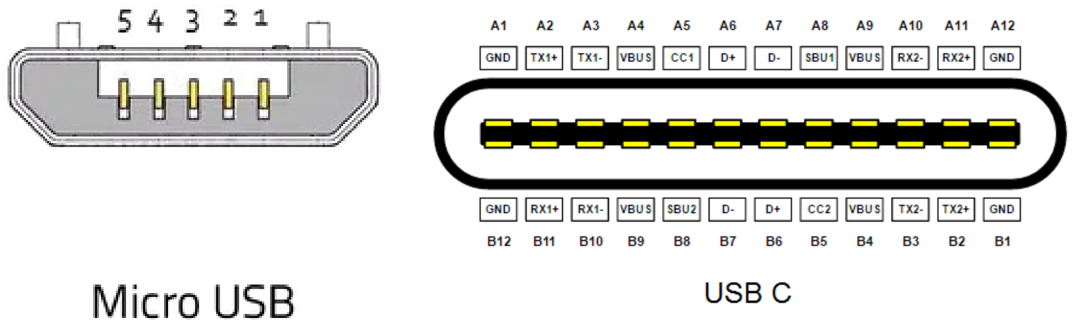
Both of these are perfect examples of what benefits and increases productivity for the user, and that is what we wanted to achieve. While we aimed to do just that, one thing we wanted to eventually improve on was the aesthetic of the input devices without sacrificing functionality, convenience, and the haptic feedback.

### 3.2.5 USB

USB is the most common market standard for wired PC peripherals (including mice/trackpads), as well as for charging electronic devices. Therefore, it is the obvious choice for our device to use USB for wired PC connection.

Previously in this document, the USB power standards were reviewed. In this section, data transmission over USB is examined. USB uses four wires, two power wires and two data wires. However, various USB connectors include more than 4 pins, including the two most common USB connectors for PC peripherals, Micro-USB and USB Type-C.

*Figure 33: USB Pinout Diagram[19]*


5 4 3 2 1
Micro USB
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12
GND TX1+ TX1- VBUS CC1 D+ D- SBU1 VBUS RX2- RX2+ GND
GND RX1+ RX1- VBUS SBU2 D- D+ CC2 VBUS TX2- TX2+ GND
B12 B11 B10 B9 B8 B7 B6 B5 B4 B3 B2 B1
USB C

As can be seen in the pinout diagram above, both of the common connector standards have more pins than simply 5v, GND, Data+, and Data-. However, those four pins do exist on both connectors, meaning that they can be broken out onto a PCB. Since the Programmable Trackpad's connection needs are very simple, we can do exactly that. The GND pin can be routed to the device's battery's GND. The 5v pin can be routed to the battery management system to charge the device. The Data+ and Data- pins can be routed to our device's control system.

Since both standard connectors are viable as solutions for our device's wired interface, we should use whichever one suits developer and consumer needs the best.

*Figure 34: USB Technology Comparison*

| Technology | Cost | Can be used for mouse/keyboard | Market state |
|---|---|---|---|
| Micro-USB | Cheap | Yes | Currently being phased out |
| USB Type-C | Cheap | Yes | Current standard |

Considering the lightweight requirements of our device, both Micro-USB and USB Type-C are powerful enough to meet the device's requirements. Since consumer technology is currently in the process of phasing out Micro-USB in favor of USB Type-C, our initial designs and prototypes used USB Type-C. However, a review of the technology used by USB Type-C revealed that it would add more design concerns to our circuit board than the Micro-USB connector. Therefore, we chose to use Micro-USB in the final product.

### 3.2.6 Microcontroller

The device's main functionality is as an input/output device. It takes in input from the touchpad, encoders, and buttons. It produces output in the form of Windows commands that simulate mouse movement or key presses. There are only a few other miscellaneous computing functions that the device needs to handle: it needs to be able to route its output to either USB or Bluetooth, and it needs to be able to translate electrical inputs to meaningful signals that can be read by Windows. To accomplish these computing tasks, the device will use a programmable microcontroller.

The MCU that we use in the device must be able to accommodate all of the functions described above. It must be capable of accepting input from four macro keys, four click buttons, one touchpad (four pins), and two rotary encoders. Additionally, it must be capable of sending output to two distinct channels (USB and Bluetooth). Finally, it must be programmable so that, in the manufacturing process, it can be configured to control the device according to specifications. Any MCU that does not meet all of these requirements will be insufficient for this project.

#### 3.2.6.1 Microcontroller Technologies

The method through which an MCU typically receives input from hardware devices is known as General Purpose In-Out (GPIO). This simple technology can measure if a digital input device is in one of two states. This technology is sufficient for the Programmable Trackpad's macro keys, which will either be in the state of pressed or not pressed.

The most basic form of serial communication, UART, is another protocol of which typical microcontrollers are capable. When a microcontroller is outputting information serially, this is the technology it will most likely use.

USB communication is a different protocol than UART; therefore, simple MCUs require a peripheral component to convert UART data to USB before it can be interpreted by USB devices, such as a PC. Some MCUs have built-in UART-USB translation. These MCUs are typically larger-scale System-on-chip (SoC) devices that include several other functionalities in addition to USB compatibility.

In total, the microcontroller that we use must support at minimum enough GPIO pins to cover each input component on the device, a UART output to send data serially to a UART-USB converter, some type of compatible input to receive data from a touchpad controller chip, and enough other pins to support a Bluetooth connection and a programming interface. The following table lists the devices that will require microcontroller pins.

*Figure 35: Microcontroller Pin Requirements*

| Device | MCU Technology | Number of pins |
|---|---|---|
| Macro Keys | GPIO | 4 |
| Rotary Encoders | GPIO | 6 |
| Mouse Buttons | GPIO | 2 |
| Touchpad | ADC | 2 |
|  | GPIO | 2 |
| Orientation Switch | GPIO | 1 |
| USB Data | USB | 2 |
| Programmer | SWD | 2 |

### 3.2.6.2 Market Microcontrollers

When exploring the MCUs that we could potentially implement in our project, our chief concerns were that the MCU fulfill all of the minimum requirements listed in the previous section, that the MCU be easily compatible with the other technologies that we intend to use (Bluetooth, USB), and also that the MCU have a significant base of documentation and existing open-source libraries that we could employ. These criteria led us to explore the following MCU options listed in the table below.

Figure 36: Microcontroller Comparison

| MCU | GPIO Pins | Bluetooth | USB | Programming Method |
|---|---|---|---|---|
| ATMega32[20] | 32 | None | None | Arduino IDE, Microchip Studio |
| nRF52840[21] | 48 | On-chip | On-chip | nRF5 SDK, Arduino IDE, CircuitPython |
| ESP32[22] | 34 | On-chip | On-chip | Arduino IDE |

All of these options fit the minimum specifications for our project. The ATMega32 was considered because of its ease of use and thorough documentation publicly available. However, it is significantly less powerful than the other options. Among the keyboard community, it is one of the most widespread microcontrollers that is used because of its qualities featuring an AVR RISC-based processor, on-board full USB module, etc.; however it does not come with Bluetooth or Wi-Fi capability. Seeing this immediately had us uninterested in this microcontroller

The ESP32 is known for its design for portable devices including mobile phones and such. It also comes with the integration of WiFi and Bluetooth 4, so it also became a strong contender for our microcontroller selection. The included peripheral interfaces included 34 GPIOs, 12-bit SAR ADC, 10 touch sensors, UART, SPI, I2S, and I2C capabilities, and many more. Programming the chip also came with beginner friendly options including Arduino and CircuitPython. Seeing the capabilities of this microcontroller was very promising, however some of the qualities of the wireless connectivity were not as up to date as we wanted as per our standards.

Another popular option used for the niche layout of split keyboards is the nRF52840, and is found on many popular market keyboards due to its specifications. It features a 64 Mhz ARM Cortex-M4 FPU, bluetooth 5.3 capability, 2.4 GHz Wi-Fi, flexible power management with a 1.7V to 5.5V supply voltage range and a 1.8V to 3.3V regulated supply for peripherals. On the Nordic website, it also gave a suggestion of applications of the usage of the micotroller including but not limited to computer peripherals (mouse, keyboard, multi-touch trackpad), electronic wearables (health watches and wireless payment devices), IoT (smart home electronics), and entertainment devices. With all this information in mind the nRF52840 became the strongest contender for our testing and final build.

The nRF52840 and ESP32 are both powerful SoC devices that could streamline the Programmable Trackpad’s hardware design by implementing several functions in one chip. Because our research into Bluetooth had already led us to the nRF52840, and because it is the more powerful chip, that is the microcontroller we have chosen to pursue.

### 3.2.7 Bluetooth

Initially, the Programmable Trackpad's primary connection method was intended to be wireless via Bluetooth. We had brainstormed this because Bluetooth is the most common standard for wireless connections between end-user devices. There are also several different standards for Bluetooth. Typical Bluetooth devices may support several of these standards. However, due to unforeseen difficulties with our programming environment, we ended up having to cut Bluetooth functionality from the final product. The final device only works via a wired connection. The following sections detail the investigations into Bluetooth that we did in Senior Design I.

#### 3.2.7.1 Bluetooth Standards

The main concerns we needed to consider were a microcontroller that supported some type of Bluetooth microchip and a Bluetooth module that fit our requirements. Due to our requirements and technological constraints, we needed to get the best support for these modules. The most recent iteration of Bluetooth was a must, especially the low energy counterpart. This was for making sure we had desirable power consumption and since we weren't really transferring large amounts of important data, Bluetooth low energy does the job perfectly. With the release of Bluetooth 5 in 2016 Bluetooth LE provided up to 2 Mbps of transfer and introduced an extended advertising mode further allowing more data bytes to be put in a single advertising packet. It wasn't until the release of 5.1 where there was Angle of Arrival of a received packet allowing better connectivity and identifying where the communication is coming from. This is important in our implementation in an environment of multiple Bluetooth connections. In the release of 5.3, there were updates to the extended advertising process being able to filter out messages in the controller stack without needing the host stack and it allowed peripheral devices, such as ours, to provide the list of preferred channels to a central device. This leads to improved throughput and reliability. Because of this we decided to use the most reliable and recent iteration of BLE.

#### 3.2.7.2 Bluetooth Modules

The nRF52840 chip has a built-in Bluetooth module. Using this, it is possible to connect to devices wirelessly. However, the manufacturer recommends using a separate Bluetooth antenna to bolster the wireless connection. Third-party manufacturers produce modules with the MCU and antenna together in one piece that can be soldered directly onto circuit boards.

Looking for a Bluetooth module wasn't very difficult to find. By referencing some Bluetooth keyboard PCBs we had in hand or just a simple Google search, we were able to find one cheap and effective. All of the Bluetooth modules considered are extensions of the nRF chip with a wireless antenna connected. The following table shows the differences between each of the Bluetooth modules considered.

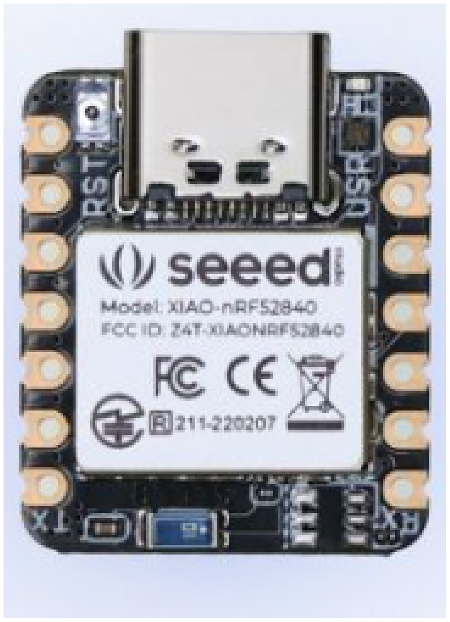
Figure 37: Bluetooth Module Comparison

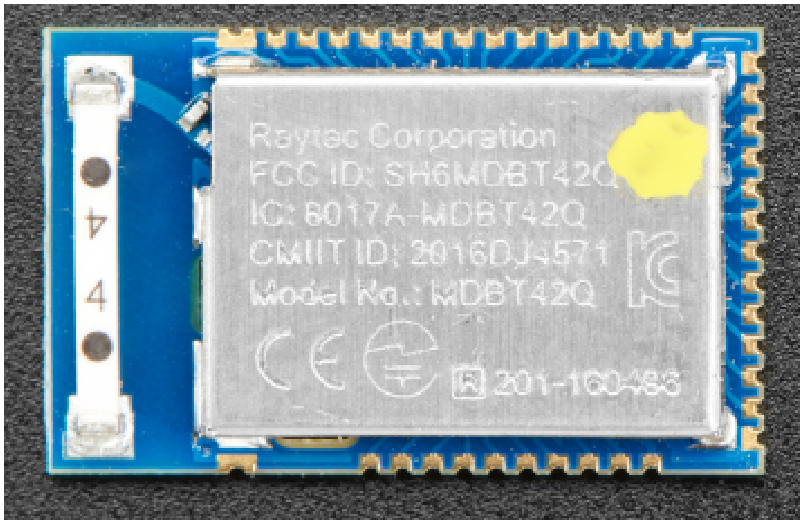
| **Bluetooth Module** | **Bluetooth Type Support** | **Voltage Supply** | **Storage** |
|---|---|---|---|
| MDBT50Q-1MV2[23] | Bluetooth 5.2 | 1.7V-3.3V | 1MB flash and 256 KB SRAM |
| MDBT40-256RV3[24] | Bluetooth 4.2 | 1.8V-3.6V | 256KB flash memory |
| MDBT42Q-512KV2[25] | Bluetooth 5.2, 5.1, 5, 4.2 | 1.7V-3.6V | 512KB flash memory |
| Seeed Studio XIAO[26] | Bluetooth 5.0 | 1.7V-3.3V | 1 MB flash and 256 kB RAM |

Of these modules, the MDBT50Q and Seeed Studio XIAO are the only ones that use the nRF52840. The other modules use earlier editions of the nRF chip. Because we chose to use the nRF52840 for our microcontroller, the other modules cannot be used for Bluetooth.

The Seeed Studio XIAO is intended to be the most user-friendly of the Bluetooth modules. It includes through-holes for easy soldering and a USB Type-C port for programming. While this makes the module an attractive option for hobbyists, it is not appropriate for our purposes.

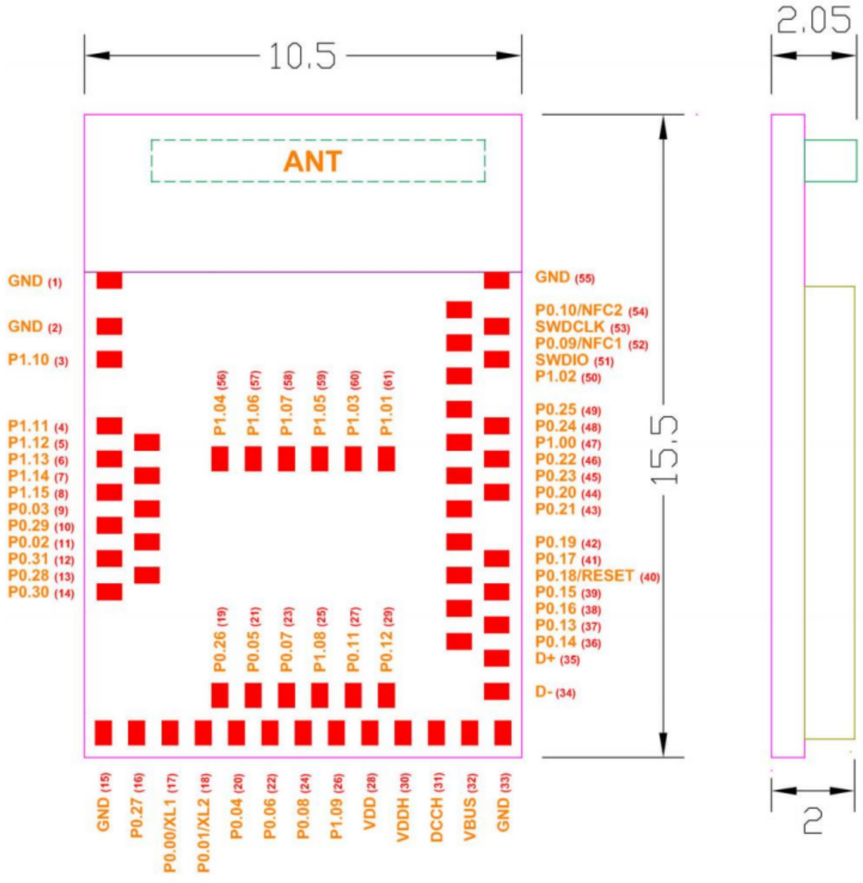
Figure 38: Seeed Studio and MDBT42Q Side by Side





Because of the reasons stated above, the MDBT50Q is the module we will be using for our device. The following diagram shows the layout of solder pads used to connect this module to a PCB.

*Figure 39: MDBT50Q Module PCB Footprint*



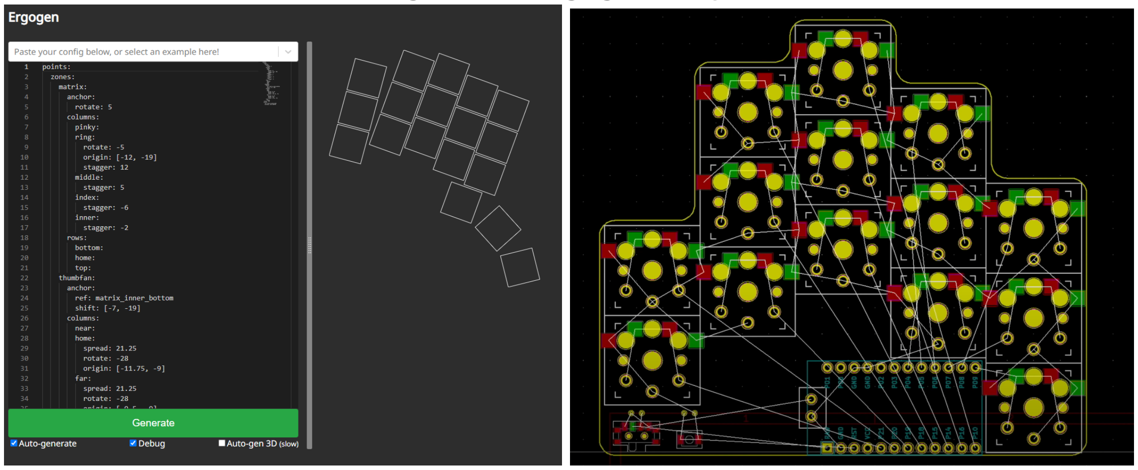
## 3.2.8 PCB

The various electrical components of the Programmable Trackpad will be routed to each other using a printed circuit board (PCB). Design challenges associated with the PCB include selecting an appropriate PCB manufacturer and arranging components on the PCB using CAD software.

### 3.2.8.1 PCB Plate

Making the PCB and finding an affordable manufacturing option was also not as difficult. An open source online PCB maker called Ergogen generates unrouted PCBs based on the user's desires. The software also emphasizes the ergonomics in a keyboard and allows the user to map out their own layout based on their own hand shape, of course within the limitations of the PCB plausible. This allowed us to work off of a baseline where we can expand on including our other requirements including rotary encoders and the touchpad while also having the option of playing around with the design. By laying out a design with a schema file, we are able to get all the .dxf files that are needed for the PCB as well as a section where the microcontroller breakout board would be placed, shown in Figure 40. Although not one of the most necessary parts of designing the PCB but does make the beginning of creating one more streamlined.

*Figure 40: Ergogen Output*



```
points:
  zones:
    matrix:
      anchor:
        rotate: 5
      columns:
        pinky:
        ring:
          rotate: -5
          origin: [-12, -19]
          stagger: 12
        middle:
          stagger: 5
        index:
          stagger: -6
        inner:
          stagger: -2
      rows:
        bottom:
        home:
        top:
    thumbfan:
      anchor:
        ref: matrix_inner_bottom
        shift: [-7, -19]
      columns:
        near:
        home:
          spread: 21.25
          rotate: -28
          origin: [-11.75, -9]
        far:
          spread: 21.25
          rotate: -28
```

There are many things that contribute to having a functional and effective PCB, and we need to consider some things when developing it. First is the actual material of the PCB. Typically polytetrafluoroethylene and advanced polyimide substrates are some of the best materials used as they offer flexibility and are what is used in the industry for phones and the medical field. As for the cladding, we should be using copper that meets the tolerance standards under the IPC. This ensures better control on the dielectric layer thickness, and with that being said, it in turn increases performance. Hole wall thickness should also be considered for resisting expansion, it typically should be about 25 microns thick. One last thing among others is the quality of the solder resist layer. An appropriate thickness of solder resistance should tolerate enough to support electrical insulation. This can reduce the risk of peeling and any other mechanical disasters. The IPC recommends UL approved solder resist material for better insulation to avoid things like corrosion.

#### 3.2.8.2 PCB CAD Software

To design our final printed circuit board, we plan on using a circuit CAD software to generate a schematic, which we will then have manufactured. There are a number of circuit CAD programs on the market today, and we performed research on a few of the most popular ones to see which one would be best suited to our needs.

The first CAD program we looked at was EAGLE. This program was initially our go-to because all members of the team have used EAGLE previously in the Junior Design class. EAGLE is a product offered by AutoDesk and has both a free version and a premium version. The free version is scaled back in terms of features and functionality, but still allows the user to create complex circuit schematics albeit limited by board area. To gain access to the premium version, users have the pricing options of $70/month, $545/year, and $1,555/3 years. There also exists an Education version of the software which is granted to students/professors if they can prove they belong to a pre-approved University. The Education version of EAGLE gives users access to all of the premium features for free. If we elect to use this software, we will acquire the Education version so we will not have to pay out of pocket. EAGLE offers a lot of convenient features such as auto-routing, which will automatically create routes on a board based on the nets in its schematic. Features like that save time and make it a great option.

The second CAD software we considered was KiCAD. This is another fairly common program that is widely available. It works in a very similar way to EAGLE, you start with a schematic, begin adding components and nets, then at the push of a button you can turn that schematic into a board layout. A unique feature of KiCAD is that you can generate a 3D model of the board once it is created. This would be very helpful when it comes to visualizing the complete chassis of our device. Having a model of not only the plastic chassis, but the completed PCB as well would give us the most complete render of how our device will look in its final and completed state. This would allow us to make tweaks and revisions to the chassis ensuring that all of our components will fit together cleanly. KiCAD is completely free to download and use.

The third CAD software we looked into was OrCAD. It works in a very similar manner to the software mentioned previously. You start with a schematic, add components, then generate a board layout from that schematic. A feature unique to OrCAD, however, is the ability to run signal simulations. Once a user has created a valid schematic and board design, they can then apply simulated signal inputs to the board. From here, the user can “probe” the board view output waveforms at various different points. While this is an interesting feature, we do not think it will be of too much use to us. OrCAD comes with the heaviest price tag of them all at $2,300. It seems like this software is geared more towards commercial use, and as such is way out of our budget.

*Figure 41: PCB CAD Software Comparison*

| | EAGLE | KiCAD | OrCAD |
|---|---|---|---|
| **Cost** | Free (With Education License) | Free | $2,300 |
| **Auto-Routing** | Yes | No | Yes |
| **3-D PCB Render** | No | Yes | No |
| **PCB Area** | Unlimited | 4 m x 4 m | 40 in x 30 in |
| **Supports Library Expansions** | Yes | Yes | Yes |
| **Auto Design Synchronization** | Yes | No | No |
| **Max Number of Layers** | 16 | 16 | 6 |
| **Simulation** | No | No | Yes |

After writing this section of the paper in Senior Design I, we had to pivot to an entirely new software called Fusion 360. This is because of licensing issues we ran into when attempting to use EAGLE (Our first choice). AutoDesk had phased out EAGLE to make way for Fusion 360. Fusion 360 is another powerful software that easily filled the role of EAGLE.

## 3.3 Firmware

The following section is an overview of the various technologies bridging the gap between hardware and software that inform the design of the Programmable Trackpad.

### 3.3.1 Keyboard Profiles

To make the macros work, the device requires a way to map the keys for the application software to interpret. This is where the selected firmware comes in. There must be a method to flash the hardware with firmware that specifies a mapping of macros to the keys and rotary encoders.

By mapping traditionally unused keys (such as the extended function keys F13-F24) to the inputs of the device, the consumer is able to use the macro keys without having to interfere with their regular keyboard. There are a few options to do that, but each comes with guidelines and limitations revolving around the hardware.

#### 3.3.1.1 Human Interface Device Specification

All modern PCs use the Human Interface Device (HID) specification for mouse and keyboard control. HID exists for both USB and Bluetooth, the two methods of communication supported by our device. In order to use the Programmable Trackpad as a mouse device without the need for special drivers, it must be configured as an HID device.

The HID Usage Tables define the standard by which HID devices communicate[27]. In these tables, there is a definition for every key on a keyboard, as well as the buttons and axes on a mouse. These definitions explain how a hardware device can signal to a host PC that a key is pressed or a mouse is moving. The Programmable Trackpad's firmware will be programmed to send the signals specified in these tables to the PC. At a base level, the device will send these exact same signals regardless of how this is implemented in firmware. In the most extreme case, we would program the device manually to generate each particular signal based on user input. However, there exist many streamlined methods of programming mouse/keyboard profiles. The following sections elaborate on possible implementations.

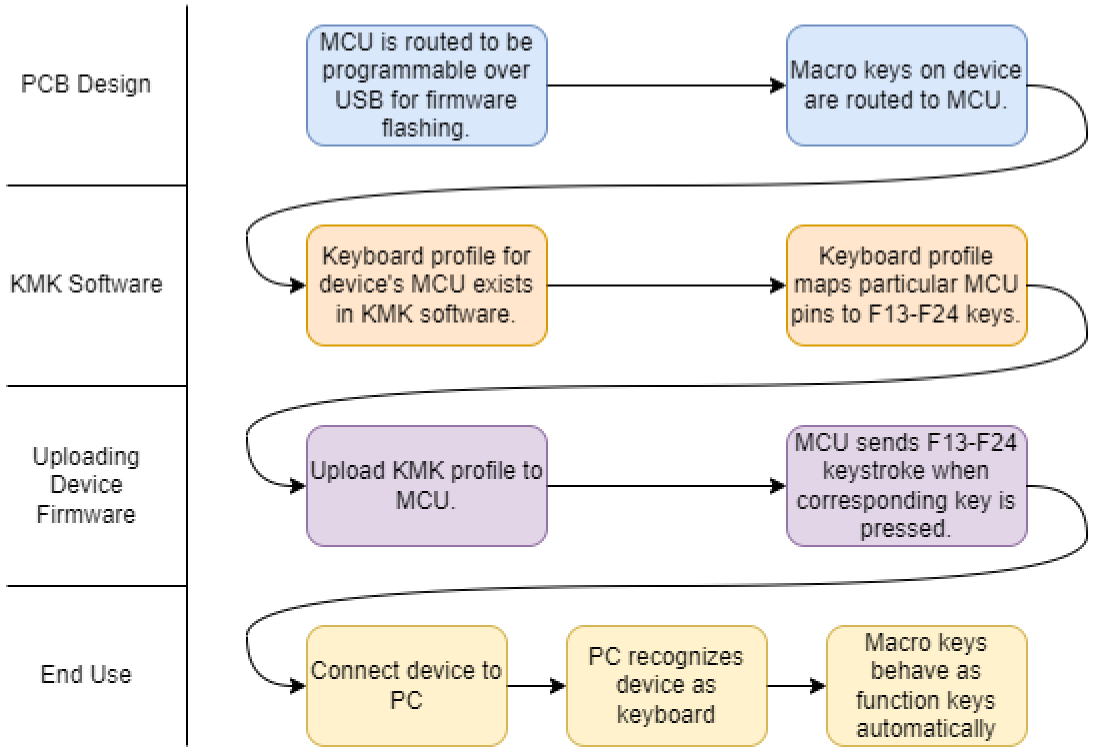
#### 3.3.1.2 Quantum Mechanical Keyboard (and Derivatives)

A particular application called Quantum Mechanical Keyboard (QMK) is used to flash market PCBs with keyboard profiles.  It is an open source community that supports computer input devices including keyboards, mice, and MIDI devices. We sought this route because it was familiar and because of the documentation, there was plenty to go off of. However when it came to implementing Bluetooth, we found that QMK was not the way to go. This affected the compatible AVR microcontrollers we had in mind for prototyping. QMK was mostly used for wired connections, and although it was kind of Bluetooth compatible, the latency was undesirable. This was due to the technical limitations since QMK was built on hardware abstraction layers for LUFA (8-bit ATMEGA), ChibiOS (ARM), and V-USB (ATMEGA), which were chips that did not support Bluetooth. If it were to happen, we would need a Bluetooth chip communicating over SPI which makes it bad for latency and power consumption.

There is another keyboard profile flashing program based on QMK called ZMK. It is another open-source program, and it supports many other features that QMK could not achieve. Since this firmware was built on Zephyr RTOS, it included Bluetooth support with low latency and low power usage; however, we were to take a small detour in selecting some of our PCB parts due to ZMK supporting mostly ARM chips. Since this isn't your ordinary macropad, we also needed to take into account how to integrate the touchpad and the mouse keys with it. The route we were planning to go with this was either use the innate Windows API to get the touchpad and mouse keys working or rather just include some of its functionality within the firmware and eventually be able to edit the keys in the software. With a little bit of scouring on the internet, ZMK tends to have a hard time with implementing mouse keys into the firmware, this is where another keyboard firmware comes in called KMK.

KMK is also an open source firmware flasher that emphasizes its user-friendliness. The firmware is built around CircuitPython and since it has many similarities with ZMK, we would not have to derail off of choosing a different microcontroller. The advantage with this firmware flasher was also its compatibility with mapping mouse keys which is what the other ones lacked. The KMK software allows us to configure the macro keys and rotary encoders as function keys without writing our own code. This requires a KMK-compatible MCU that is connected to a PC with the KMK software over USB. The following flowchart is a step-by-step outline of how the KMK firmware is programmed to the MCU. Note that this process is done in the assembly step of development, after the PCB is soldered, but before the device is consumer-ready.

*Figure 42: KMK Flowchart*


PCB Design
KMK Software
Uploading Device Firmware
End Use
MCU is routed to be programmable over USB for firmware flashing.
Macro keys on device are routed to MCU.
Keyboard profile for device's MCU exists in KMK software.
Keyboard profile maps particular MCU pins to F13-F24 keys.
Upload KMK profile to MCU.
MCU sends F13-F24 keystroke when corresponding key is pressed.
Connect device to PC
PC recognizes device as keyboard
Macro keys behave as function keys automatically

While KMK does seem to be a better option than QMK or ZMK, it still does not fulfill all of our needs, so we will need a different firmware flashing program that supports mouse functionality as well as key presses. However, it could offer an option of giving the user a more accessible way of uploading their own firmware.

#### 3.3.1.3 Circuit Python for Bluetooth Connection

Circuit Python is a programming language built off of Python made as a beginner friendly option to program microcontrollers. With the most recent version, it offers support for the nRF52840, the microcontroller at the center of the Programmable Trackpad. Using the Adafruit Bluetooth LE libraries, we are able to use the provided libraries and code up a connectivity where we can transfer data. Using the microcontroller board to test functionality, we can connect it to an app on the phone and test the capabilities of the chip and connection. In our case we can use a premade Circuit Python code for the HID keyboard and map out the pins to the key that is being pressed. We can also use the HID mouse example and take the serial information from that for our trackpad use.

#### 3.3.1.4 Arduino HID Library

The well-documented Arduino firmware for AVR microcontrollers includes an HID library that allows for very developer-friendly implementation of mouse and keyboard technology in a microcontroller. When the HID library is used, it begins by establishing the device's USB interface as an HID device, and from there, any HID commands can be sent over the same USB connection.

The simplicity and ease of use associated with this firmware is appealing; however, it has certain drawbacks. Much of the Arduino code base has been translated for use with other microcontrollers, including our microcontroller, thus allowing us to use it as part of the device's development.

### 3.3.2 Serial Wire Debug

The microcontroller that we selected for this project, the nRF52840, is based on the ARM architecture. In order to configure the microcontroller with the firmware specifications of our product, it is necessary for our development team to use a programming interface compatible with the ARM device. The most common standard for programming and debugging ARM-based chips is JTAG. JTAG specifies particular pins to connect to a microcontroller for the purpose of debugging, but it is also often used for programming.

The JTAG standard has a derivative standard known as Serial Wire Debug (SWD), which is often a more attractive option due to the low number of pins necessary. SWD only specifies 2 mandatory pins, SWCLK and SWDIO (one clock and one bi-directional data). The nRF52840 was designed with this standard in mind. There are two pins on the chip specially designated as SWCLK and SWDIO. Because these pins exist and are the standard for programming, our team will use SWD to program the system firmware. The table below shows a list of some considered debuggers/programmers.

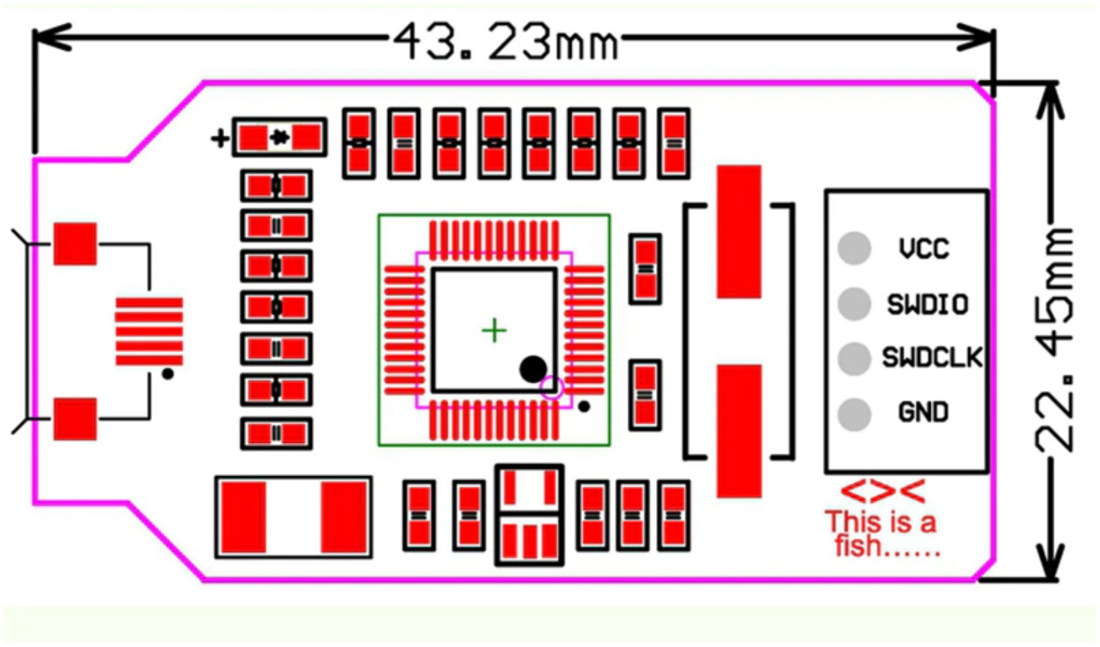
*Figure 43: SWD Programmers Comparison*

| Programmer/Debugger | Price |
|---|---|
| SEGGER J-Link EDU - JTAG/SWD Debugger | $69.95 |
| SEGGER J-Link BASE - JTAG/SWD Debugger | $449.95 |
| SEGGER J-Link EDU Mini - JTAG/SWD Debugger | $19.95 |
| nrf52840 DK | $57.18 |

These debuggers were recommended on the Adafruit website, but because of trading, shipping, and availability issues, they are currently out of stock. Especially with the prices of these devices, we had to look for options that were within our budget. The main thing we really needed was the J-Link functionality as it allows us to enable the use of common IDEs. In our case for the nRF52840 we need to install the bootloader for CircuitPython using the Segger Embedded Studio or the Arduino IDE, that way the Adafruit libraries can be implemented. For alternatives to the sold-out devices, we sought to look for a third party device that was J-Link compatible. The J-Link software will allow us to configure our microcontroller and activate the bluetooth capabilities. To start testing we ended up purchasing the third party device shown in the figure below. We purchased this specific one through Aliexpress and they had many options to choose from when it came to third party debuggers/programmers. This was one of many that was capable of doing what we needed as it was capable of being compatible with the installation of SDK, Arduino, CircuitPython software for our microcontroller. This was our best option due to not being able to justify purchasing $500+ SEGGER debuggers for the sole purpose of the microcontroller. This one includes the ARM OB motherboard, a microUSB cable, and a 4 pin SWD download cable. The only thing lacking compared to the original OB debugger, is the JTAG interface and only retains the SWD interface for debugging. All microcontrollers with an SWD interface are supported. Some other features of this third-party device include compatibility with traditional V8 emulators, support for 3.3V output where maximum output current is up to 300 MA making it very convenient for users to debug and download the target board, self-recovery fuse provides short-circuit prevention and makes debugging safer, ESD protection device, and of course its very portable size.

*Figure 44: Third-Party J-Link Debugger*

*Figure 45: Third-Party J-Link Debugger Diagram*



Through further research and testing we found that the 3rd party programmer was unable to do what we wanted it to do and we had to look to an official programmer, this led us to using the PCA10056.

## 3.4 Application Software

In addition to communicating with the computer sending various key presses, mouse movements, etc.; There needs to be an application interface with which the user of this product would interact. The interface should be running as a program on the user's computer where it will receive and send data to the device as well as provide the user with an interface to customize and program the macro key functionality. The responsibility of the firmware is to map the keys for the computer to interpret, whereas the software application should be able to reprogram function keys to different macros. The user should be able to control and customize what macros they desire. The software should present some default macros, making it quick and easy for the user if he/she wants to use a common macro. This information needs to be stored within the software and can be re-generated at runtime; however, the user should also have the ability to create their own custom macros. The only catch for the application software itself is for this customized input from the user, it must be stored where they should be able to have access to these custom macros even after termination of the software.

We are limited to the number of physical keys on the device itself, hence the software should be able to store as many custom macros as the user wants where they should be interchangeable.

### 3.4.1 Coding Language Consideration

There are many different coding languages out there to consider when it comes to application design. The programming language that will best fit this project and efficiently achieve objectives is one that is very compatible with other aspects of this project such as the firmware. As well as a number of built-in libraries and/or open source libraries that are maintained by larger developers.

Our application will allow users to select and choose from various macros to reassign the device's keypads to these macros. The keypad will be reading as additional function keys such as F13 or any other unused keys on the keyboard. Hence, we will need software integration and/or scripts to run that can change the functionality of that key press to something that they would desire.

This reassigning macro software integration must be compatible with the selected coding language. For replicating this project in the future, we found that with our limited amount of time it was more efficient to have our design plans focus on importing an open source library to integrate our program with rather than reinventing the wheel so to speak.

The application must also have some sort of small database setup to store the user's preset macros. For example, say a user designs and creates a number of macro presets but they might not want to "upload" them to the keypads right away. Hence, even if they quit the application running, compiling (or running it since Python isn't a compilation language) it again should still show all of their presets. The coding language must have an integration with an existing database language or create our own database system such as writing to a text file.

*Figure 46: Python vs Java Comparison*

| | **Python** | **Java** |
|---|---|---|
| Built-in Libraries | Yes | Yes |
| AHK Compatible | Yes | Not clear documentation from AHK themselves |
| Write files to text file | Yes | Yes |
| SQLite Compatible | Yes | Yes |

#### 3.4.1.1 Python

Python has an extensive amount of built-in libraries as well as updated open-source libraries that are regularly updated. It is very compatible with connecting via hardware and software as well as showing data to the user in an graphical way.

For changing the functionality of a function key, further discussion and considerations are expressed in the programmable macros section. However, briefly we considered Python in this scenario with the well documented, resource heavy, and extremely compatible AutoHotkey (AHK) integration with Python. These scripting languages benefit us where the main design aspect can be focused on the scripting to change the macro key functionality where all Python has to do is open the scripts and run them (along with a few library imports).

This integration is very efficient where during the software design phase more time can be spent on something like the graphical interface that the user sees and interacts with. Python has very extensive GUI libraries where some even have drag and drop design tools where this will increase user satisfaction as well as overall usability of the application software.

Towards the end of this project as we are wrapping up the software design aspect, the final application product should be given as an executable file. In the real world industry this might be done in effort to hide your source code from competitors but that doesn't apply to us in this university setting. Therefore, the main reason for this goal of converting the code into an executable is to avoid having the user install any of the developer tools such as Python in order to run the application. We have to assume that the user doesn't know anything about Python where we have to limit their experience to just to the user interface of the application.

Since Python is not a compiled programming language, there isn't a built-in feature of converting your code into an executable file (.exe). Hence, an open-source library such as PyInstaller should be used to convert the code into system instructions and commands. However, since our program will involve running other scripting languages (such as AutoHotkey); we considered something a little more user friendly. Auto Py To Exe is an open source library that uses PyInstaller and presents the different options PyInstaller has to offer through a Graphical User Interface (GUI).

#### 3.4.1.2 Java

Java has a number of built-in libraries that can connect and communicate with a variety of devices as well as a built-in GUI library to display and interact with particular data and other functions. The GUI isn't exactly modern where the overall design might appear dated, potentially causing user dissatisfaction and/or confusion.

The documentation is fairly large but is outdated where it is harder to find open source libraries and software that are regularly maintained by reputable developers. There may be difficulty getting this frontend interface to interact with the firmware and might not be extremely compatible.

For the integration of scripting languages to remap function keys to different macros, the documentation isn't as extensive and along with the community not being resource heavy for Java. If this language is chosen you might have to rely on online forums and community examples to integrate Java with AutoHotkey (AHK) for example.

We also found that Java was compatible with both of our Data Storage considerations. With the ability to read/write to text files, as well as the ability to connect and send queries to a SQLite database.

### 3.4.2 Programmable Macros/Changing Key Press Functionality

Since the application's purpose is for creating macros and assigning them to the device's keypad, there needs to be software considerations ensuring that the operating system performs an automated feature after the key press.

Reviewing our firmware considerations, this application should communicate with the keypads themselves so whenever the user presses the keypad the Operating System will recognize that a function key (assigned to extra keys such as F13-F24) has been pressed. This is where the firmware functionality ends, the firmware can map and change which key was pressed but can not implement the automation functionality the shortcut macro offers. For example, one of the macro keys will always be recognized by the operating system as F13 due to the firmware, but there needs to be a software integration that will change what the function key F13 can do (what macro it will execute).

This software functionality could be written from the ground up, reinventing the wheel, so to speak. However, to greatly benefit this project we determined that using an open source software that can communicate directly with Windows (or other Operating Systems) should be used to perform the shortcut macro. Hence, the focus during the design phase can be on creating files using these scripting softwares to create more complex and useful macros for the user to choose from. A stretch goal would be to design a scripting program ourselves, but after reviewing the considerations, we didn't find a significant benefit that it would bring to the project. With the use of scripting software, the more complex macros that could be designed would improve overall user satisfaction and usability of the application software.

Overall, after these scripting software are considered and we have reached the design phase of the project; the main focus for software design is coding up the scripts that perform both simple and complex macros. For example, once the user decides to save their new macro to the PC and clicks the save button, the application program should open and run these scripts to update the device's functionality.

#### 3.4.2.1 Microsoft PowerToys

When considering other technologies to create macro automation scripts for particular keys, Microsoft PowerToys stood out due to Microsoft having created this product. We considered this early on during our technology investigation where we understood that we needed software to communicate with the Operating System such as Windows to reassign the functionality of a particular key press.

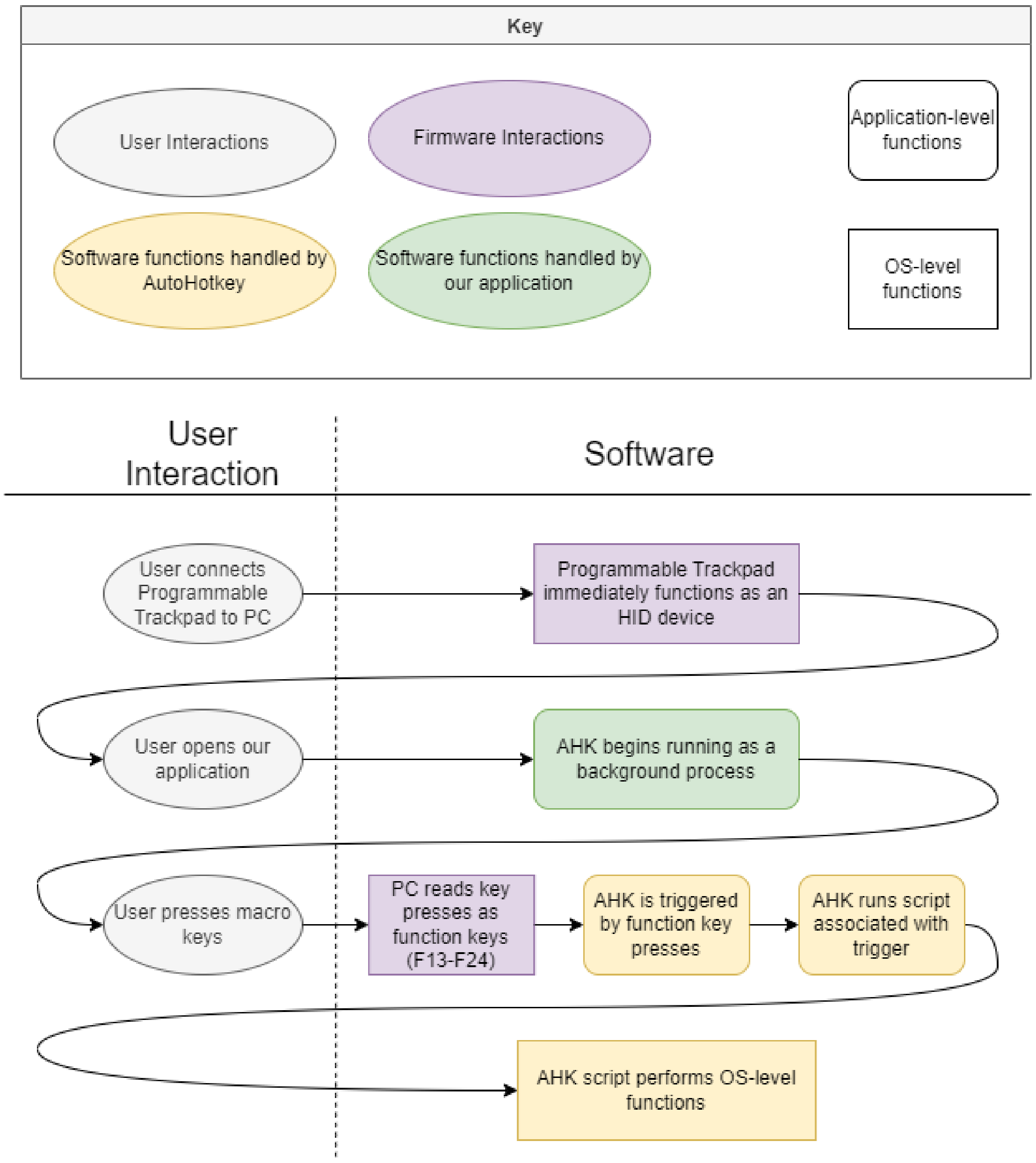
But under further investigations we found the documentation for this tool to be very minimal where the documentation focused on enhancing and changing the Windows experience rather than creating scripts for custom macros. We found documentation on remapping keys to another key press, changing shortcuts, and launching apps based on shortcuts. However, the ability to easily map these keys to a complex macro is not present in our investigations where more logic and software work must be done to meet the goals of this project.

#### 3.4.2.2 AutoHotkey

One popular option for PC users to create their own macros is an open-source program called AutoHotkey (AHK). This program defines a simple language for writing scripts that work on an OS level. Each script begins with a trigger, which is defined as some particular input to the computer. After the trigger, the script prescribes a function to be executed when the macro is triggered. The functions available with AutoHotkey are numerous, and they would certainly cover the scope of what a user may want to accomplish with our Programmable Trackpad.

Because AHK is open-source, it would be possible for our application software to come packaged with an installation of AHK. When the user starts up our application, it could simultaneously start the AHK process, automatically bringing all of the user's macros online. The following flow chart illustrates how this process would work.

*Figure 47: Flowchart of AutoHotkey Implemented in Application Software*


Key
User Interactions
Firmware Interactions
Application-level functions
Software functions handled by AutoHotkey
Software functions handled by our application
OS-level functions
User Interaction
Software
User connects Programmable Trackpad to PC
Programmable Trackpad immediately functions as an HID device
User opens our application
AHK begins running as a background process
User presses macro keys
PC reads key presses as function keys (F13-F24)
AHK is triggered by function key presses
AHK runs script associated with trigger
AHK script performs OS-level functions

Due to the flexibility and open-source nature of AHK, our team has decided to use it as the basis for macro functionality in our application.

### 3.4.3 Graphical User Interface (GUI)

In order for this device and associated software to be user friendly and to have a low learning curve, the frontend technology must be simple enough yet detailed. Any buttons should be self explanatory or at least a description indicating its functionality.

*Figure 48: GUI Library Comparison*

| Library Name | Coding Language | Advantages | Disadvantages |
|---|---|---|---|
| PyGUI | Python | - Available on all platforms<br>- Open source | - Not pre-installed |
| PyQT | Python | - Drag and drop design tools<br>- Available on all platforms<br>- Advanced widgets to if app upgrades into higher scale | - Large and complex<br>- Documentation is very minimal<br>- Not free and not open source |
| Tkinter | Python | - Built into Python<br>- Open Source<br>- Lots of resources and documentation | - No design tools (QT designer)<br>- Too simple, difficulty when program expands |
| Kivy | Python | - Drawing tool<br>- Modern GUI<br>- Open source | - Minimal documentation and resources<br>- Not pre-installed |
| javax.swing | Java | - Simple<br>- Decent documentation<br>- Old but abundant resources | - Forces us to use Java where compatibility with other tech is not guaranteed. |

After considering these different GUI libraries and the two coding languages, we are heavily leaning towards the Python language with the abundant libraries it has to offer. Further testing and prototyping will need to be done to determine the best GUI library to use and what tools it offers to improve and quicken the GUI design phase of the project.

### 3.4.4 Macro Preset Storage Considerations

During the process of creating a macro-key, the user will have a variety of options to design their own presets of their favorite automation macros. Instead of learning a new scripting language, reading through different documentation and online forums on how to create this automation process; the user will simply have to interact with a few buttons and dropdowns to create their own macros.

In this application, an example workflow would be to create a new macro, select the “Run (user-specified program)” preset, give the macro a name, assign a macro key to this script, and click save. The user would then have a functional macro without having to understand the AHK scripting language. The macros that the user creates will be used many times over an unknown time period, so it is important that they are stored on the PC for future use. Therefore, for this section we will be considering how to store these user presets on the application program to further improve the user’s experience. If we didn’t implement this feature, the user would be forced to recreate all of their macros all over again if they swap them out frequently. This way, it is as simple as selecting a previously created macro from a dropdown list.

Running the application, this macro information shouldn’t only be stored just in RAM. Presuming the macro is stored in various string variables, the user should be able to store as many macros as he/she wants. This data must be stored and written on the computer where it should be present even after terminating the program and/or restarting the computer. This is important for our goal of creating a user-friendly application. For example, say you want to switch a key to a new custom macro; even though the old macro is being overwritten, it should be stored and selectable for the user in case the user ever wants to swap back.

There are various different methods of storing and writing data to the computer where each has its own technical advantages and disadvantages. For the goals of this project, an online database will not be considered extensively due to forcing the user to have an internet connection to modify his/her Programmable Trackpad. The benefits of an online database are only significant if there are multiple concurrent users. This wouldn’t apply for this product since there is only one local hardware device. Therefore, the variety of databases that will be considered will be found locally on the user’s computer.

#### 3.4.4.1 Text File Storage

The simplest way to store data in case of termination/exit of our application program, is to store strings and write it into a text file. Then after restarting the application during initial runtime, the data is then read from the file and put into variables.

However, for someone to replicate this project without the limits of a university setting (limited time and budget), it would be much more difficult to expand upon or upgrade the application software with this text file based database setup. If we have more variables that need to be stored, the older text-file does not contain these new variables. If no action is taken such as rewriting the text database or including additional logic in the code; the data could be read incorrectly causing inaccurate macros for example.

For this project, if we limit the software to a single user, the text-based storage is advantageous for its simplicity. However, if we pursue our advanced goal of having multiple users per device, there would have to be more thought and effort in storing and retrieving data accurately for all users. For example, if

John has 20 custom macros, Susie shouldn't be able to see any of those and vice versa.

Lastly, regarding the previous sections on the different programming languages we considered; both Java and Python support creating a text file and then being able to read and write text which can be then interpreted into data through programming logic.

#### 3.4.4.2 SQLite Database

Even though we may not be storing extensive information in these custom macro presets, if not designed correctly the software storage design will be very limited and un-scabable. To ensure future software features within this application, there is assumed new data that must be stored for the associated user. Hence, a pre-built and extensively tested database language such as SQL could benefit this project.

SQL was first developed in the 70s where it is reliable having very minimal bugs where thousands of developers use and consistently test. For this project, we found that SQL could offer stability, high performance, and compatibility with our other considered technologies. SQLite is compatible with both Python and Java where we could integrate data created within the application, make a connection to the local database, and send queries to retrieve or store preset macro information.

If we went with the other consideration of simply storing data in a text file, we are leaving ourselves open to potential bugs and/or unintended features when designing all of the logic could cause inaccurate information displayed. Hence, extensive testing would be required for a text file storage where with SQLite we only have to test a few queries such as select, insert, and delete. SQL might be overkill for the scope of this application where the only challenge would be creating the database and ensuring that we could receive and update data accurately. However, overcoming this challenge could bring a lot of efficiency and value to this project.

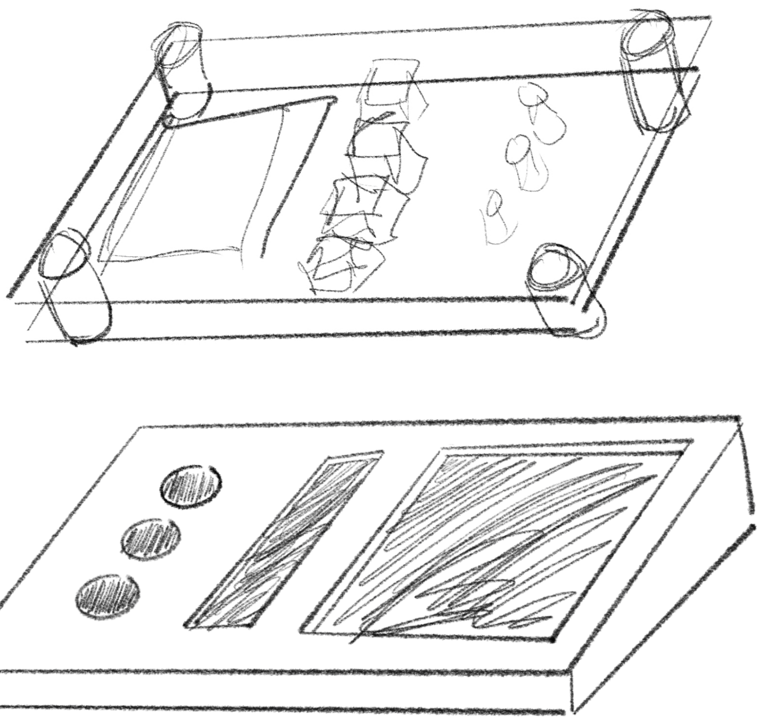
## 3.5 Chassis and 3D printing

One of the aspects we wanted for the programmable trackpad, was to have a chassis that was affordable, light, non-obstructive, and easy to produce. For this, we looked to 3D printing. Through some of the preliminary designs we needed areas for the mechanical switches, rotary encoders and trackpad to be easily accessed without obstructing each other. Initially we thought a more simple approach by putting the PCB in between some acrylic plates and having standoffs on the corners support the structure but we felt that it didn't look consumer friendly. The final design for the chassis we ended up using was an enclosure with openings for each of the functionalities: the rotary encoders, mechanical switches, and the trackpad. The materials we considered for the chassis were PLA and PETG each coming with their pros and cons. Both are very good options, but there were also some other elements that we were able to consider.

Figure 49: 3D Printing Material Comparison

| PLA | PETG |
|---|---|
| Renewable and made of natural raw materials, makes it biodegradable | Thermoplastic made of PET making it recyclable but not biodegradable |
| Weaker than PETG but stronger than ABS and stiffer than both | Water, chemical, and fatigue resistant, making it more durable |
| Slightly cheaper | Can get pretty pricey |
| Lower extrusion temperature | High extrusion temperature |

Figure 50: Chassis Sketches



To actually access the PCB for production or testing, the bottom of the chassis will be attached to the top using screws. Inside of the casing will also be standoffs to hold the PCB in place. The chassis will be a flat design to accommodate for the user's dominant hand. As we are all slightly unfamiliar with CAD-ing, some research on the structure of the chassis will be heavily researched.

Through this research there are many things that should be considered when building an enclosure, and the material used affects the parameters we make for the enclosure. The table below is a list of things we should consider as well as a description of why.

*Figure 51: Chassis Design Considerations Table*

| Guidelines/Suggestions | Reason |
|---|---|
| 2 mm wall thickness | Enclosure structure |
| Radii/fillets to corners | Helps reduce stress at corners and edges, also offers ease in printing |
| 0.5 clearance for internal electronics | Compensation for distortion, expansion, shrinkage or internal components |
| Extra 0.25 mm to diameter of screw and fastener holes | Allows for extra clearance for self drilling screws |
| Subtract 0.25 mm from diameter | Self taping holes, allows screws to bite to the casing |
| 2 mm port clearance | Ease of placing internal electronics |
| Add lugs, lips, and cut outs (5 mm in width) | Aids in alignment of the enclosure |
| Ribs and gussets | Improves integrity and reduces stress |
| Bosses (1 hole diameter around the hole as a start) | Reduces likelihood of bulging, distortion, fracturing around screw holes |
| Uniform wall thickness | Good design practice |

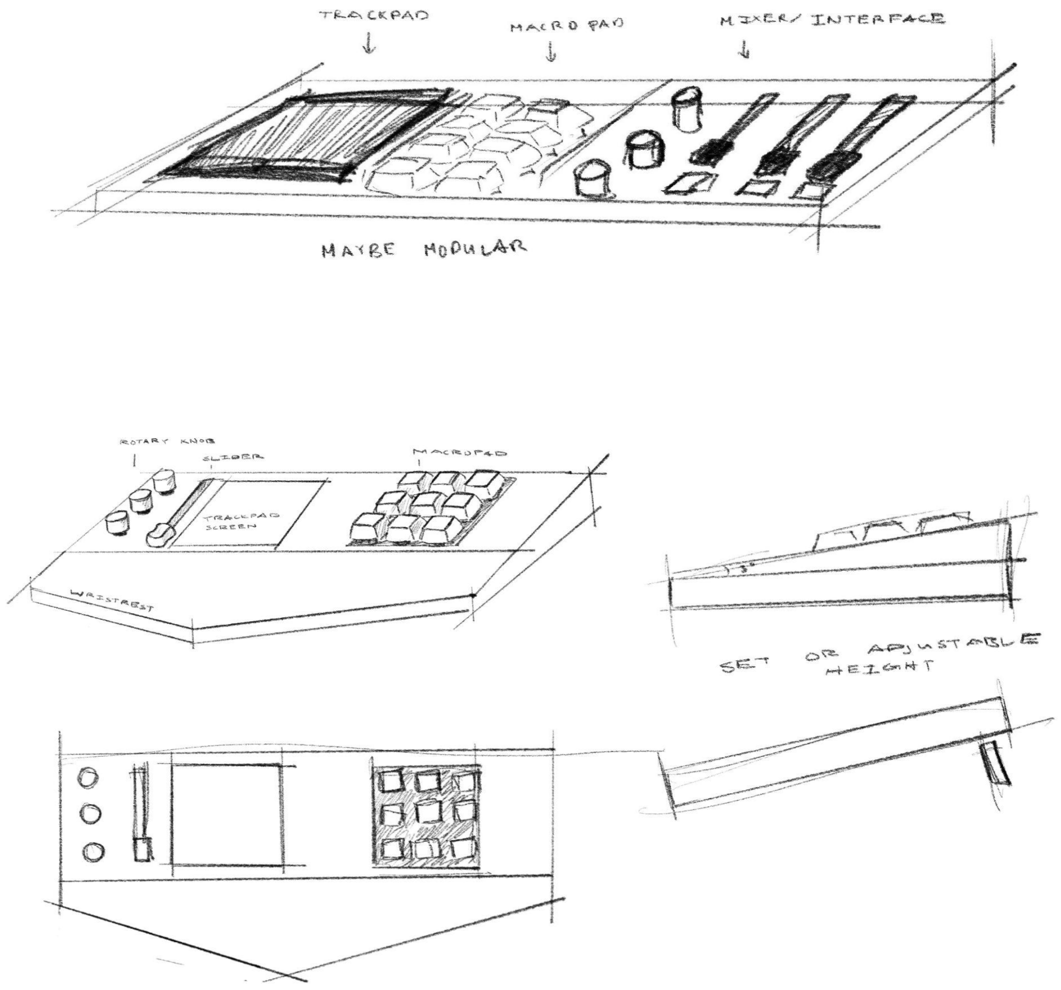
Possible stretch goals we explored were higher quality metals as the material for the chassis and maybe get into plates in between the top and the PCB which will enhance the acoustics of the device. Some other materials for the chassis would include aluminum, stacked acrylic, or even polycarbonate; as for the plates, aluminum, brass, FR4, and polycarbonate are all potential options.

# 4 Design Details

## 4.1 Preliminary Designs

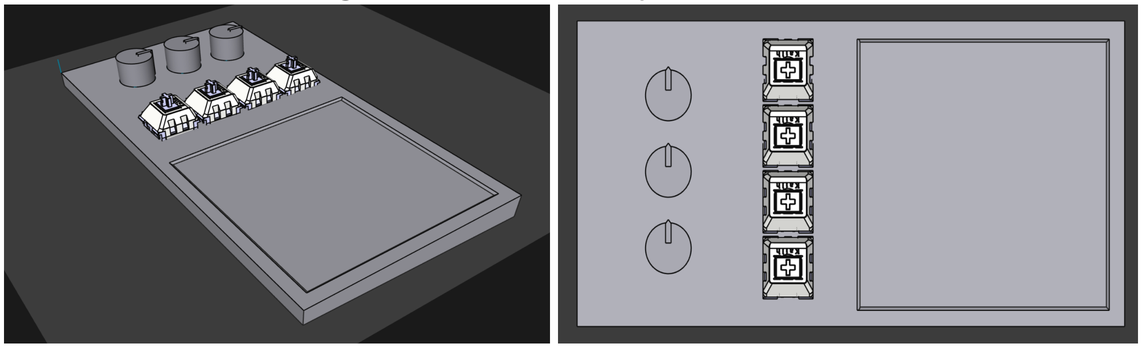
We went through a few iterations when coming up with the design for the Programmable Trackpad. Finding the balance between functionality and a desired form factor was part of the motivation for the design. The functionalities we wanted to implement ended up being a part of the final design but the unit to achieve said functionalities narrowed down to a few input units. For example, some audio sliders were considered since they could also be programmed a certain way, but there were some complications with the firmware and the input converters.

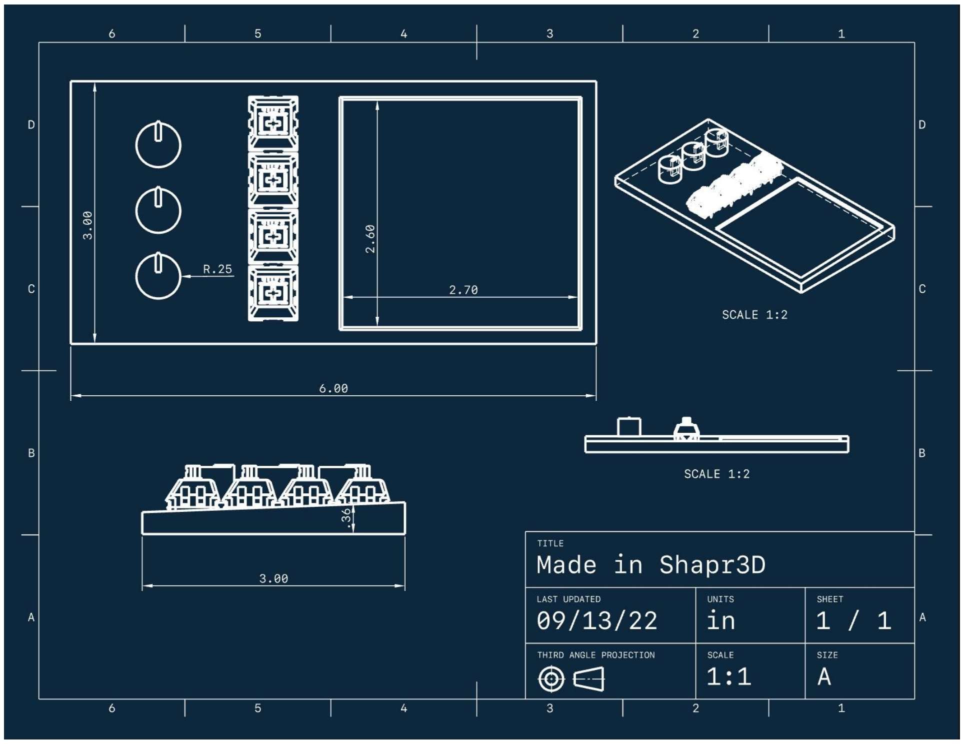
*Figure 52: Initial Concept Sketches*



Our main functionalities we ended up sticking with were macro control, rotary encoders, and a trackpad control. From there, it was a matter of how much we should have on a unit which would decide its form factor. Another aspect that was explored was the ergonomics for both right-handed and left-handed users. This is where the Bluetooth implementation comes in handy allowing the user to place it anywhere that is comfortable for them without the hindrance and limitations of a wired connection. Eventually we came up with the final design that caters to our desired functionalities, ergonomics, and a portable form factor.

*Figure 53: Initial Concept CAD Model*





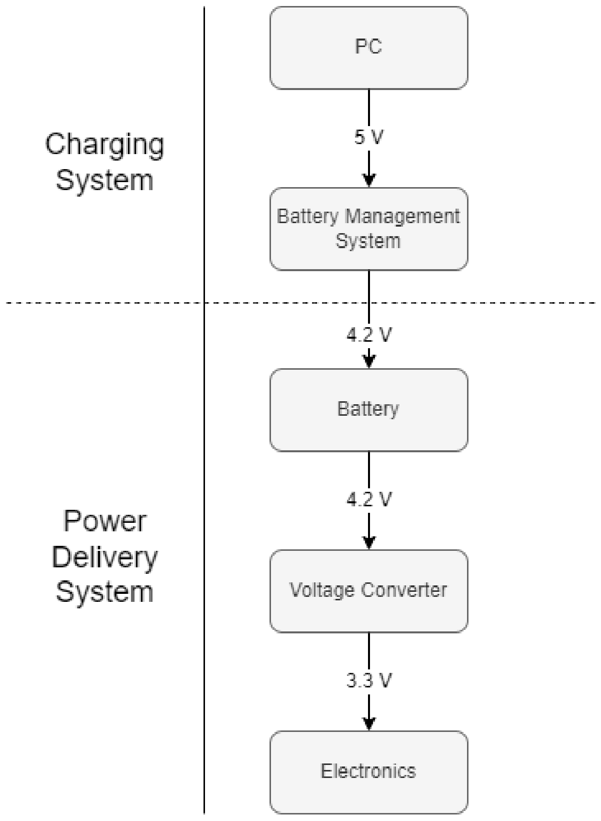
## 4.2 Hardware

The Programmable Trackpad's hardware is the first element of the product with which users will interact. It's important that its design is user-friendly and functional. The following sections go into depth on the design of each element of the system's hardware.

### 4.2.1 Power System

The device's power system is made up of three major components: the battery management system, the battery, and the voltage converter. The battery management system provides constant current and constant voltage to the battery. The battery is the static source of power for the entire device. The voltage converter provides constant current and constant voltage to the device electronics. The general flow of the power system as it pertains to our design is that there is a charging system and a power delivery system. The electronic components in between exist to regulate voltage and current for the next stage. This flow is shown in the chart below.

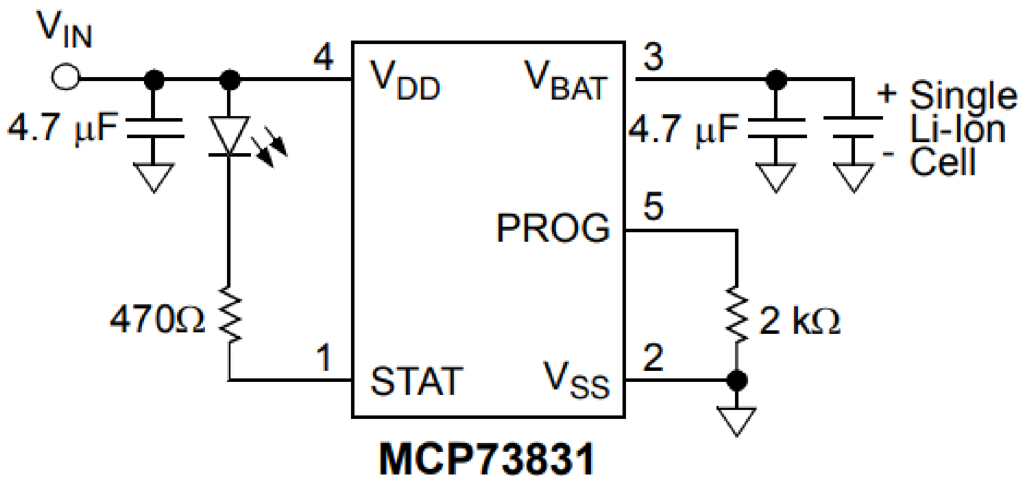
*Figure 54: Generalized Power System Flow Diagram*



### 4.2.1.1 Battery Management System Electrical Schematic

The chip used for the battery management system is the MCP73831. That part's documentation includes a circuit diagram showing its typical application as a lithium battery charger. Since that application is identical to our goal, we replicated the circuit in our device. The given circuit diagram is shown below.

*Figure 55: MCP73831 Circuit Diagram from Datasheet*



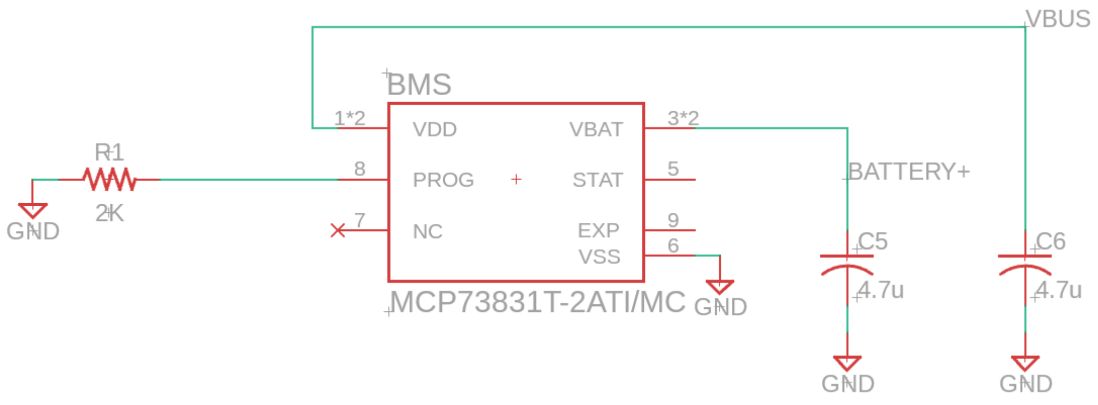
The $V_{DD}$ pin of the MCP chip is the input pin, which accepts USB voltage. The $V_{BAT}$ pin is the output pin, which supplies 4.2 volts. It is recommended to connect both of these pins to 4.7 µF capacitors to ground. The PROG pin sets the current regulation value based on the value of the resistor connected to the pin, according to the equation below.

*Figure 56: MCP73831 Current Regulation Equation*

$$I_{REG} = \frac{1000V}{R_{PROG}}$$

A 2 kilohm resistor attached to the PROG pin results in a maximum output current of 500 mA, which is also the maximum current of USB. The $V_{SS}$ pin sets the ground for the rest of the chip. Finally, the STAT pin is used to signal when the battery is charging. If an LED is connected to this pin, then it will light up during the charging process and extinguish when the charging process is complete. This functionality could be applied to the Programmable Trackpad, but we chose not to. As such, in our design, the STAT pin is left open. The following schematic shows how the chip is connected in our circuit board.

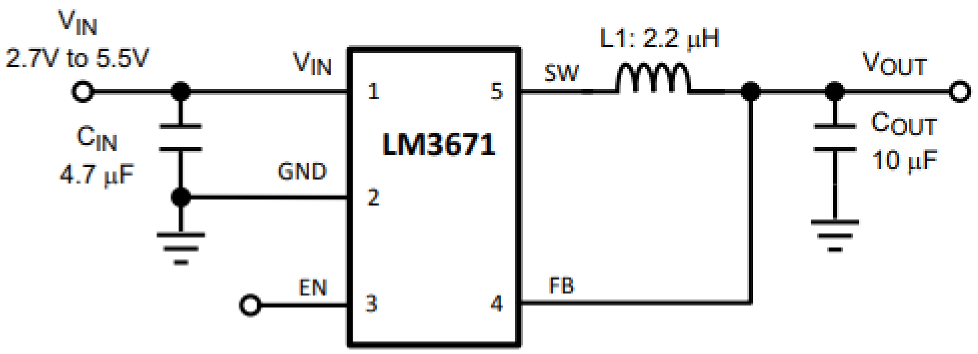
*Figure 57: MCP73831 Circuit Schematic*



### 4.2.1.2 Voltage Converter Electrical Schematic

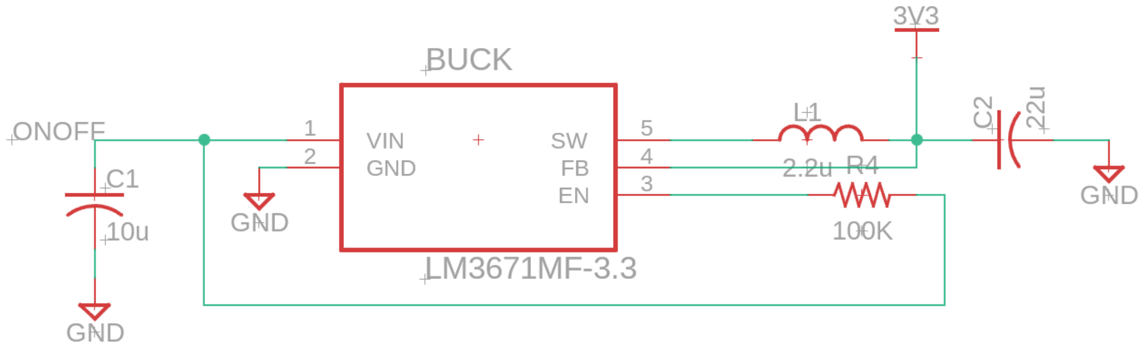
The voltage converter used to step-down the battery voltage to 3.3 volts is the LM3671 chip. The documentation for this component includes a circuit diagram for the typical application of using the chip as a buck converter. This circuit was replicated in our own design. The given circuit is shown below.

*Figure 58: LM3671 Circuit Diagram from Datasheet*



The $V_{IN}$ pin on this chip connects to the voltage supply (the LiPo battery in this case). In practice, this voltage ranges from 3.7-4.2 volts. The FB pin on this chip is the output pin, which supplies a constant 3.3 volts. It is recommended that the input is attached to a capacitor of at least 4.7 μF, and the output is attached to a capacitor of at least 10 μF. The SW pin is used to specify the chip's mode of operation. It is recommended to use a 2.2 μH inductor to connect this pin to the output voltage. The EN pin enables or disables the device. It is recommended that the device only be enabled when the input voltage is greater than 2.7 volts. It is specifically recommended that this pin not be left floating. For our design, we attached this pin to the input voltage through a 100 kilohm resistor because this is how it is connected in the development board we used for our prototype. Similarly, our device uses the capacitance values from this prototype board for the input and output capacitors. The following schematic shows how we connect this chip in our circuit board.

*Figure 59: LM3671 Circuit Schematic*



Note that the $V_{IN}$ pin on this chip is connected to the battery indirectly through a switch. This physical switch is the on/off switch for the whole device. As stated previously, the switch opens or closes the circuit between the battery and the voltage converter because the voltage converter draws current when connected. When the device is turned off, the battery is disconnected to save energy.

### 4.2.2 Microcontroller

The device electronics outside of the power system are centered around a microcontroller, the nRF52840. This device accepts input from the macro keys, rotary encoders, and touchpad. It then processes this input and outputs HID commands.

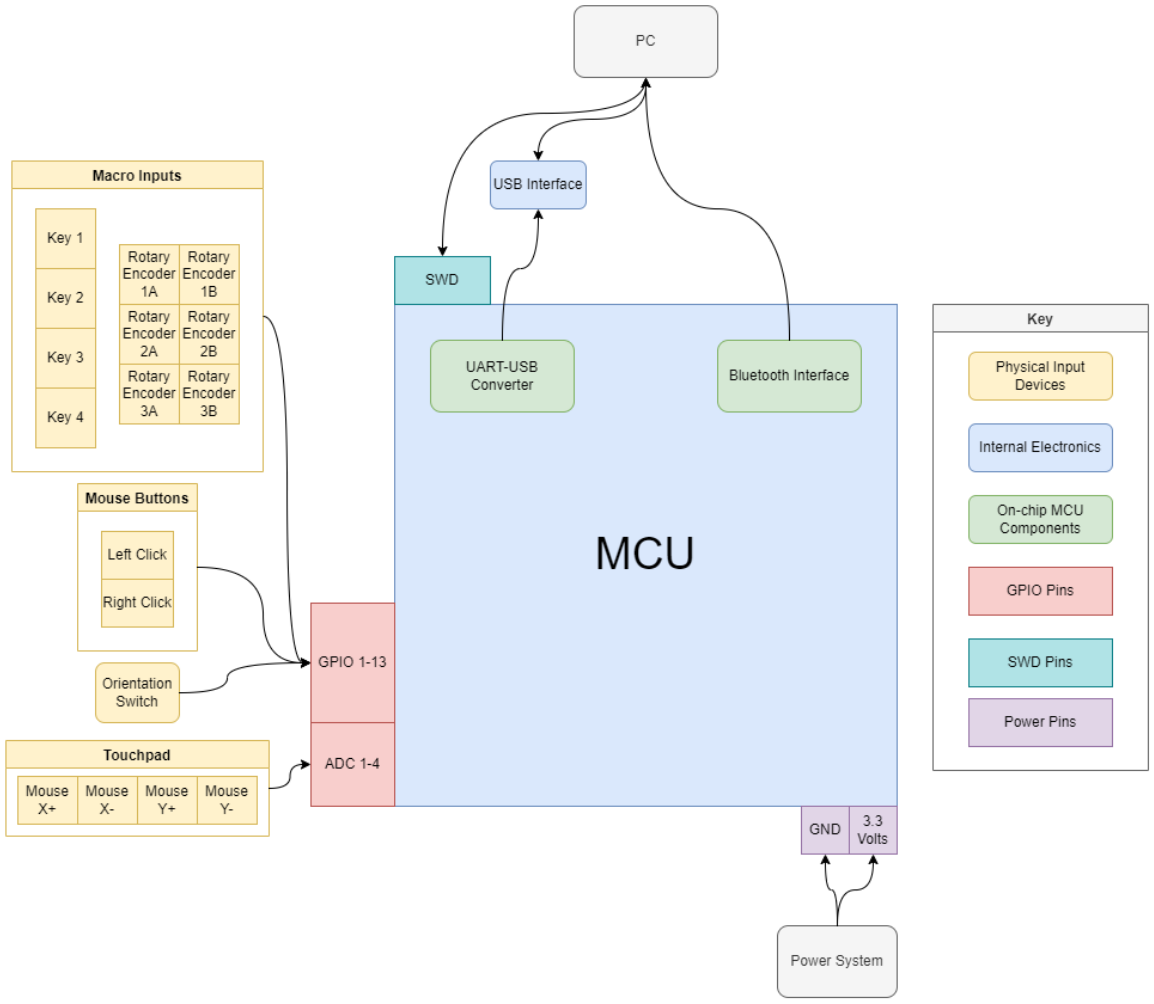
The microcontroller has a built-in USB output and a built-in Bluetooth module on the chip itself. Because these systems are integrated in the microcontroller, there is no need to connect separate chips with these functions to the microcontroller. The only connections that the microcontroller has on the circuit board are to the input devices (and associated controllers), output, programming, and power.

The touchpad communicates using analog signals on ADC pins, and all other input devices connect to the MCU with GPIO. The microntroller’s output is connected to two channels, the internal USB interface and the internal Bluetooth interface. The Bluetooth interface has no connections on the PCB because it is a wireless device. The USB interface in the MCU connects to the USB connector on the PCB. There are multiple inputs for the microcontroller which include up to 51 GPIO pins with respective functionalities, some being also analog inputs, and having trace data functionalities.

Two pins on the MCU are designated for SWD, the programming interface for the chip. These two pins will ordinarily be left floating. However, during the development process, they can be connected to a PC through an external SWD programmer.

The following figure illustrates all of these internal and external connections that the MCU makes.

*Figure 60: nRF52840 Connection Diagram*



### 4.2.2.1 Bluetooth Module

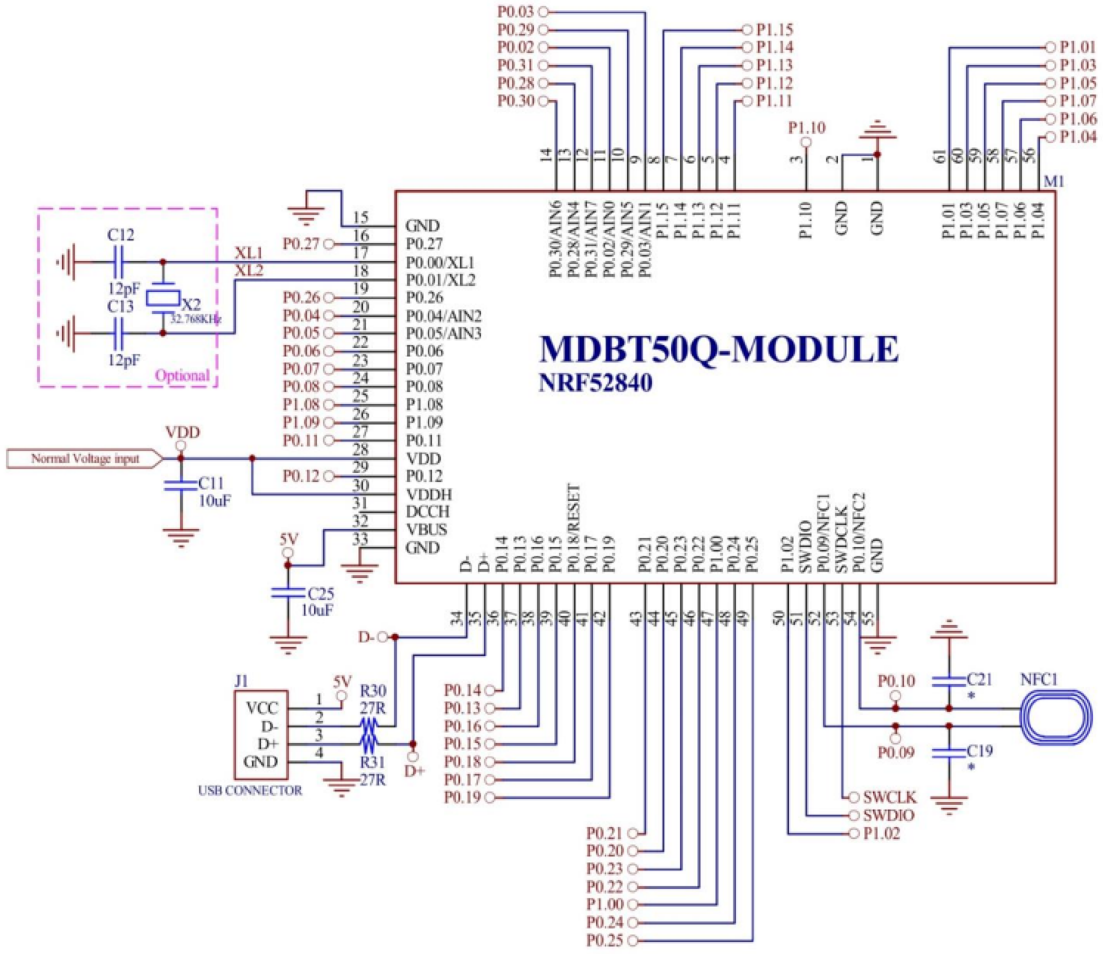
The MDBT50Q-1MV2 is the Bluetooth module used for the final iteration of the PCB. It uses the nRF52840 microcontroller and integrates the antenna solution for Bluetooth. It features 48 general purpose I/O pins and a Bluetooth antenna with an excellent connection of up to 2 Mbps data rate through Bluetooth 5. The following table describes the features of this module that were concerns when designing the device.

Figure 61: MDBT50Q-1MV2 Technical Specifications

| Module | Bluetooth | Available Interfaces | Dimensions (mm) | Supply Voltage |
|---|---|---|---|---|
| MDBT50Q-1 MV2 | Protocol Support: BT 5.3 | GPIO, SPI, UART, I2C, I2S, PMD, PWM, ADC, NFC, and USB | 15.5 x 10.5 x 2.05 | 1.7V to 5.5V |

The documentation for the MDBT50Q includes several useful diagrams, including a circuit diagram suggesting how to connect the module to a circuit board and the PCB layout of the module. The circuit diagram was used as a reference for our own schematic. This diagram is shown below.

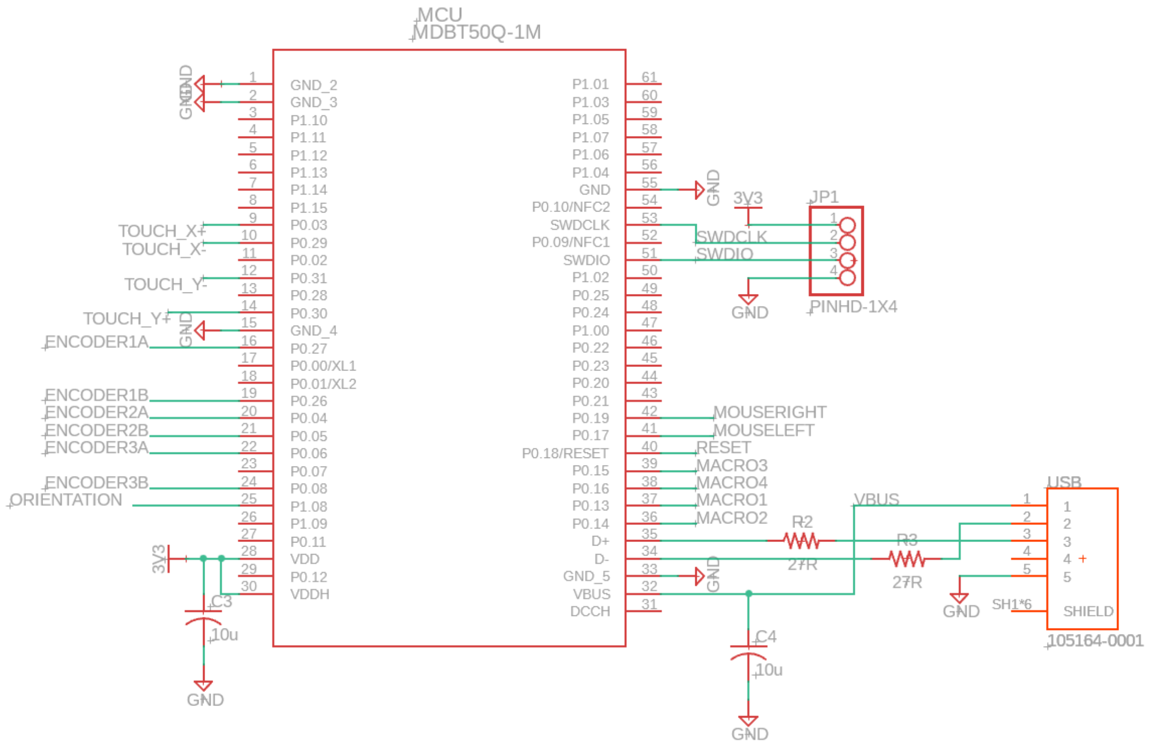
Figure 62: MDBT50Q Circuit Diagram from Documentation



This schematic shows how to connect the pins of the MDBT50Q for necessary functions. In the documentation, it is explained that the “optional” crystal component is necessary to regulate input voltage with the module’s LDO mode. Our circuit uses an external voltage regulator, so the optional crystal setup was ignored in our design. Additionally, the schematic shows how to connect an NFC device to the microcontroller’s NFC pins. This functionality was not included in our design, so this component and its associated capacitors were ignored. The USB pins (VBUS, D+, and D-) are shown to connect to a USB connector in the schematic. The data pins are connected through 27 ohm resistors, and the VBUS pin is connected to a 10 µF capacitor. The VDD and VDDH pins are shorted and connected to a 10 µF capacitor to ground. These pins are connected to the constant 3.3 volt supply from the buck converter. All of these pins are used in our design as they are used in this schematic.

Additionally, the GPIO pins on the module are connected to each of the input devices on the Programmable Trackpad. Remaining GPIO pins are left floating. The following schematic shows how the module is connected in our design.

*Figure 63: MDBT50Q Circuit Schematic*



The following table lays out the same pinout described by the schematic above, but with explanations of the pins in use in our design.

*Figure 64: MDBT50Q Pin Description Table*

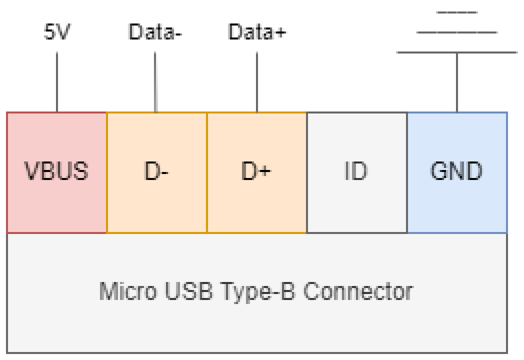
| Pin | Type | Connection | Explanation |
|---|---|---|---|
| 0.04, 0.05, 0.06, 0.08, 0.26, 0.27 | GPIO | Rotary Encoders | These GPIO pins were selected according to the datasheet for least interference with the Bluetooth antenna. |
| 0.13, 0.14, 0.15, 0.16 | | Macro Keys | |
| 1.08 | | Orientation Switch | |
| 0.17, 0.19 | | Mouse Clicks | |
| 0.03, 0.29, 0.30, 0.31 | ADC | Touchpad | The touchpad's four pins send variable resistance values to the MCU, which decodes the signals using ADC. |
| SWDIO, SWDCLK | SWD | Programmer | Programmer attaches to these pins to upload firmware to the device. |
| D-, D+ | USB | USB Connector | On-chip USB interface allows for USB connection to these pins. |
| VBUS | Power | | 5-volt power directly from PC. |
| VDD, VDDH | | 3.3 volt | VDDH is used to determine if voltage regulation is necessary. When shorted to VDD, no voltage regulation is necessary. |
| GND/2/3/4/5 | Ground | | |

## 4.2.3 USB

The only physical component needed for the USB system is a Micro USB Type-B female connector. This connector is soldered onto the PCB in a position where it is accessible for the user. From there, the user can plug a cable into it.

The Micro USB Type-B female connector part has 5 connections. Only four of these connections are necessary; the remaining pin is unused. The unused connection is not wired into the MCU on the PCB. The following diagram shows which connections will be soldered and which will remain open.

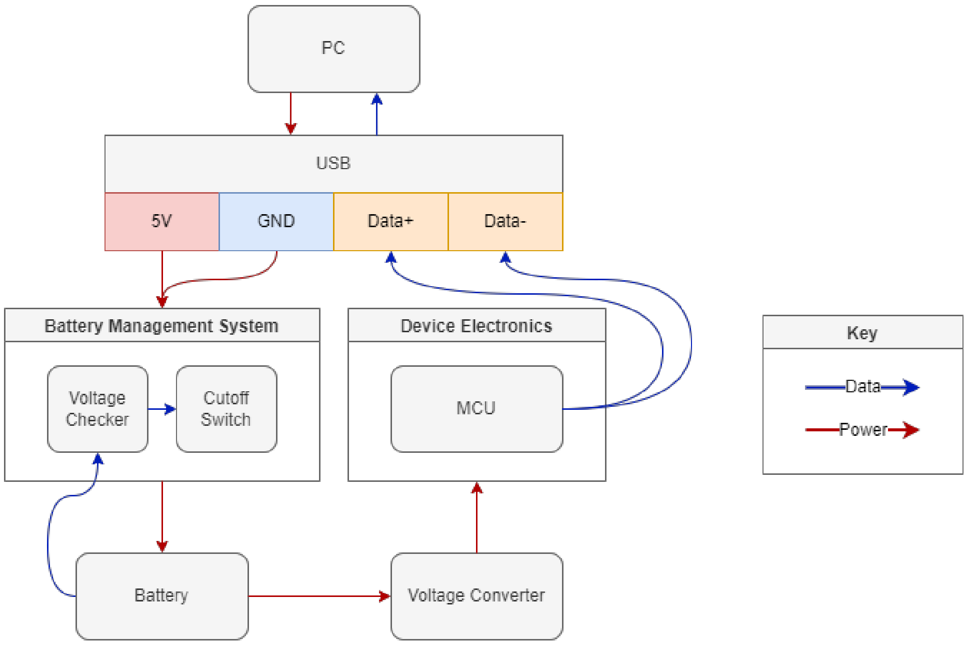
*Figure 65: USB Breakout Diagram*



In this diagram, each rectangle above the connector represents a connection. The blue connection is connected on the ground plane. The red connection is connected to the MCU on a 5 volt node. The orange connections labeled "Data+" and "Data-" connect to the USB interface on the MCU.

The USB system can be simplified into a device with four pins (5V, GND, Data+, and Data-). This system interacts with several other subsystems within the device, as shown in the following diagram.

*Figure 66: USB Block Diagram*



### 4.2.4 Touchpad Control

The touchpad uses resistive technology to isolate the location of a finger press on the device. This location is expressed in x and y-coordinates through 2 x-labeled pins and 2 y-labeled pins. These pins connect to ADC pins on the microcontroller, which then interprets the signals in firmware.

### 4.2.5 Miscellaneous Input Units

As mentioned in previous sections, one of the goals of the Programmable Trackpad is that it be accessible to right-handed and left-handed users alike. Because of the layout of buttons on the device, it may not be comfortable for everyone in one orientation. For the purpose of hardware design, the relevant information is that a basic switch on the device informs the microcontroller whether the device is in left-handed mode or right-handed mode.

Additionally, the circuit board includes three rotary encoders for macro inputs and four Kailh hot-swappable sockets for the macro keys. The mouse buttons are controlled by four basic buttons on the PCB (two left clicks and two right clicks). The schematics for all of these devices are shown below.

*Figure 67: Circuit Schematic for Macro Keys*

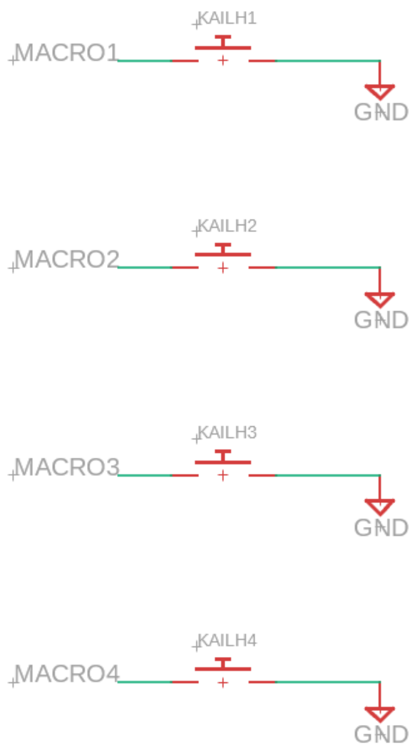


*Figure 68: Circuit Schematic for Mouse Buttons*

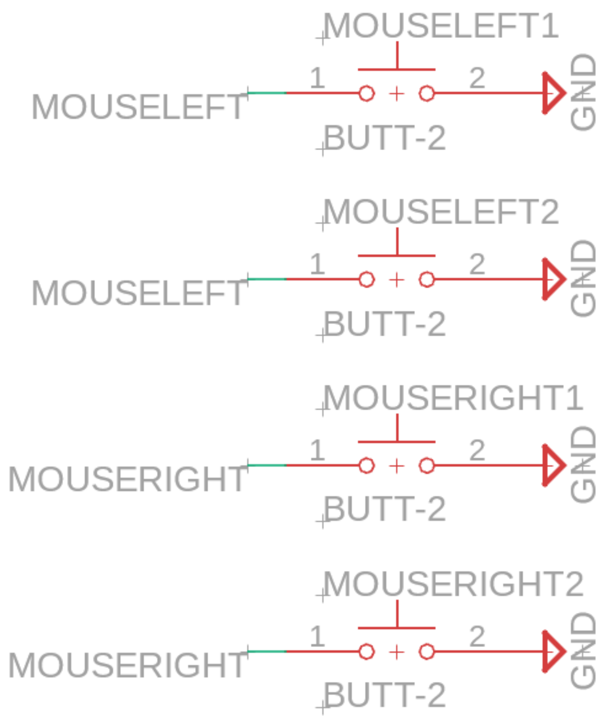


*Figure 69: Circuit Schematic for Rotary Encoders*

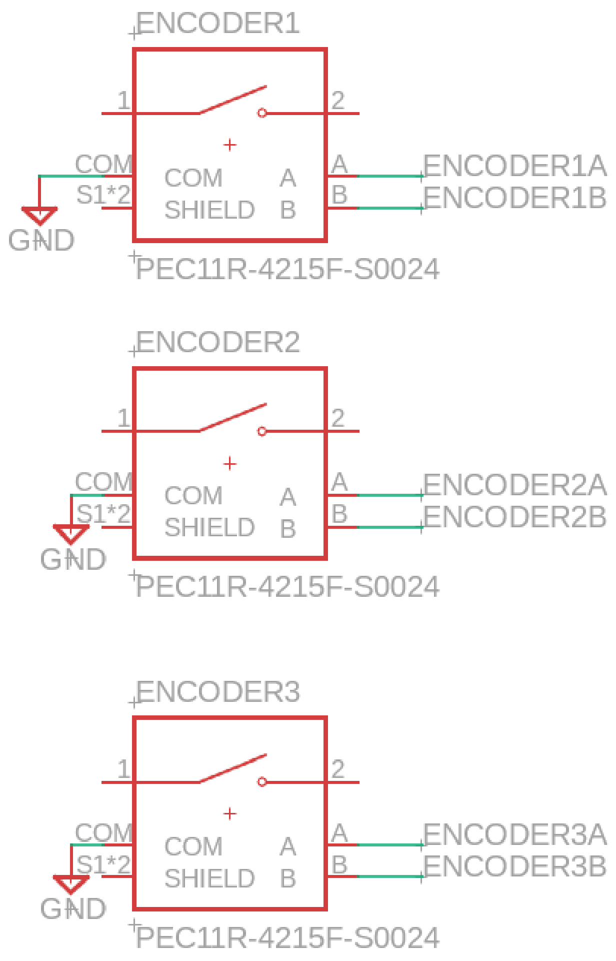
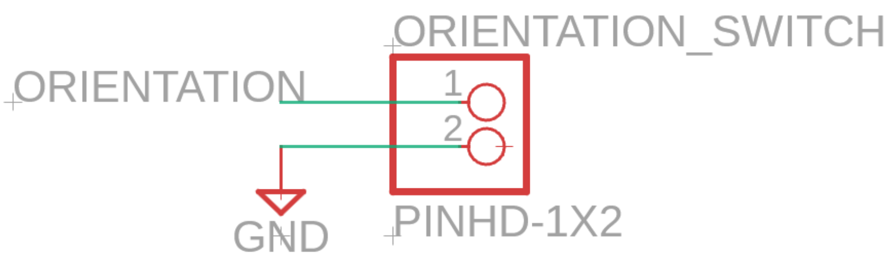


*Figure 70: Circuit Schematic for Orientation Switch*



Note that all of these input units are referenced to ground in these schematics. The typical application of input devices such as these requires pull-up resistors on each of the input pins to reference the open state of the input device. These resistors are absent from our schematics because they are configured internally in the microcontroller through firmware.

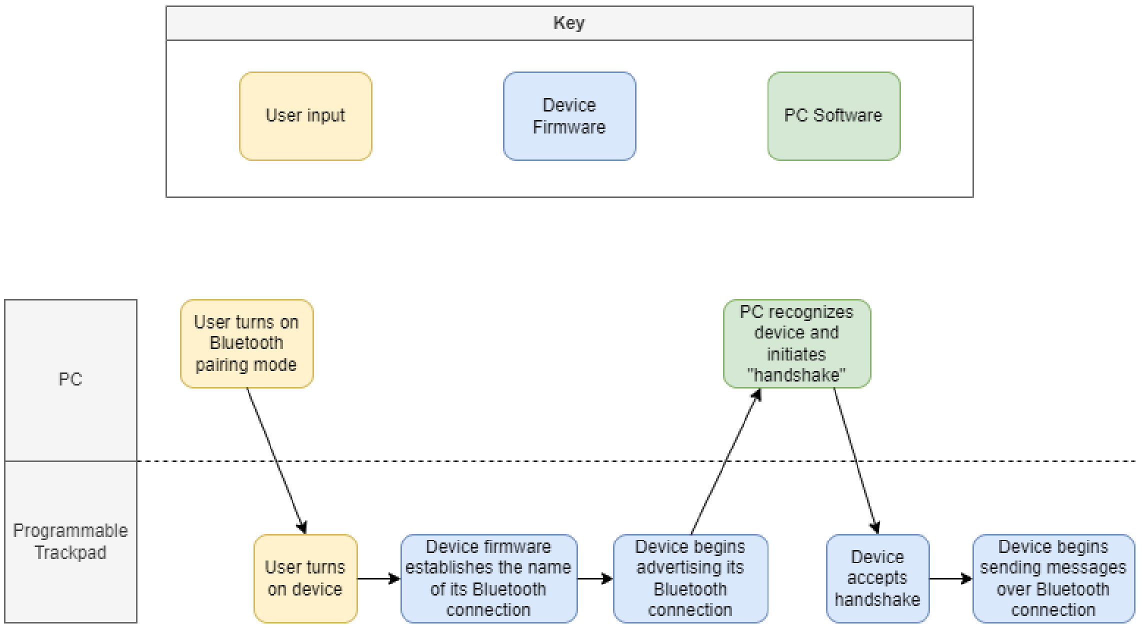
## 4.3 Firmware

The hardware of the Programmable Trackpad includes multiple computing devices which are responsible for processing input and output. The design challenges associated with such devices include writing code for programmable devices and selecting communication protocols for the devices to use. The following sections detail the device's firmware design.

### 4.3.1 Establishing Connection

When the device is powered on, it immediately begins searching for a Bluetooth device with which to pair. The user must configure this Bluetooth connection on the PC. As long as the device stays on and the PC does not sever the connection, then the Programmable Trackpad will stay paired with the PC. The following flowchart explains the pairing process that the microcontroller must facilitate.

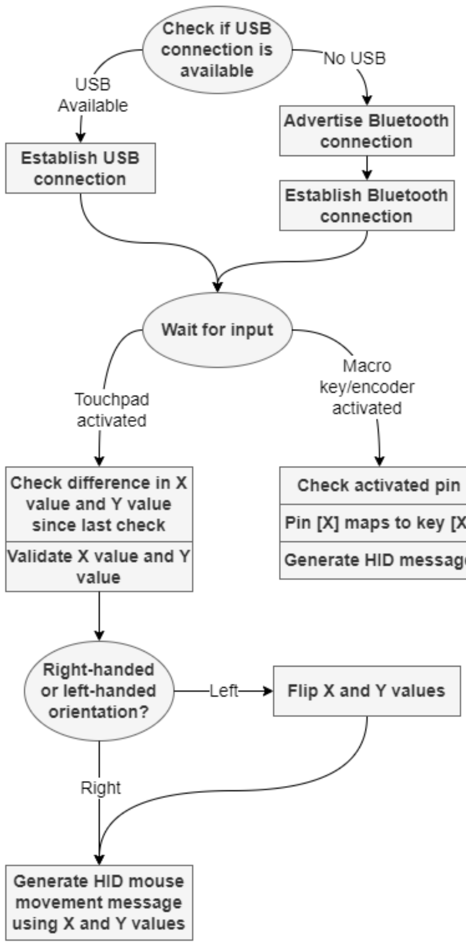
*Figure 71: Bluetooth Pairing Process Flowchart*


Key
User input
Device Firmware
PC Software
PC
Programmable Trackpad
User turns on Bluetooth pairing mode
PC recognizes device and initiates "handshake"
User turns on device
Device firmware establishes the name of its Bluetooth connection
Device begins advertising its Bluetooth connection
Device accepts handshake
Device begins sending messages over Bluetooth connection

### 4.3.2 Hardware Control

The majority of the input/output functionality of the Programmable Trackpad is handled by an nRF52840 microcontroller. Its responsibilities include: processing input from input devices, translating input into meaningful output, and routing output to Bluetooth or USB. The following chart explains how each of those responsibilities is carried out.

*Figure 72: Hardware Control Flow Diagram*


Check if USB connection is available
USB Available
No USB
Establish USB connection
Advertise Bluetooth connection
Establish Bluetooth connection
Wait for input
Touchpad activated
Macro key/encoder activated
Check difference in X value and Y value since last check
Validate X value and Y value
Check activated pin
Pin [X] maps to key [X]
Generate HID message
Right-handed or left-handed orientation?
Left
Flip X and Y values
Right
Generate HID mouse movement message using X and Y values

### 4.3.3 Firmware Uploading

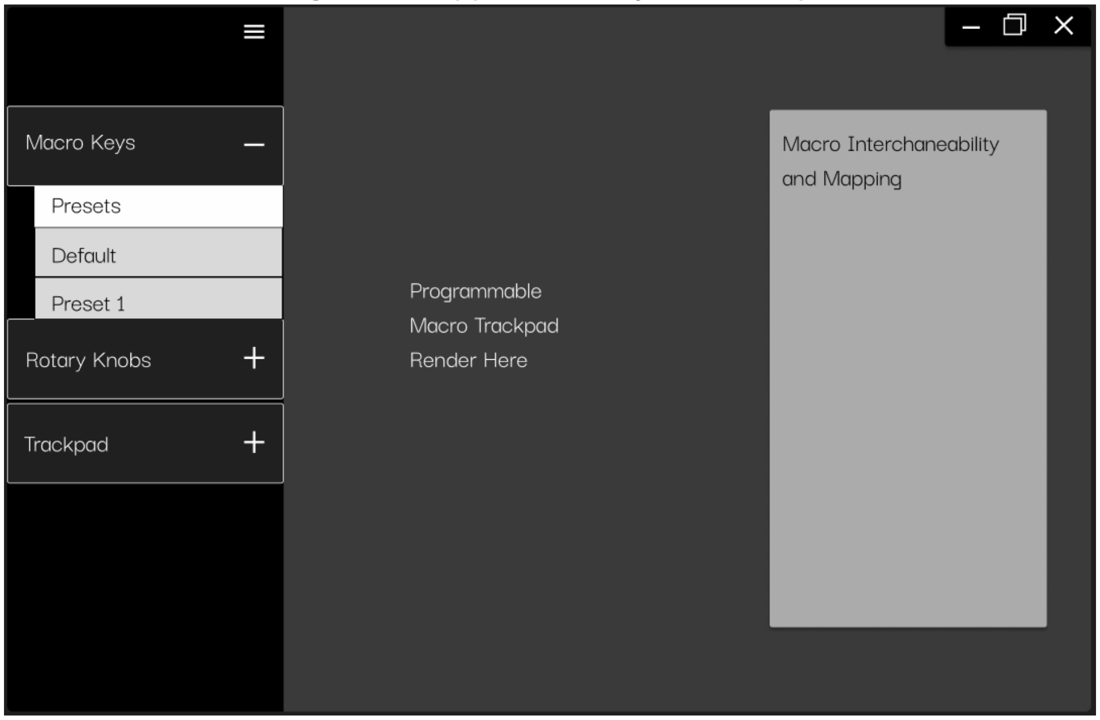
Once the microcontroller's code is written and compiled, it must be loaded onto the chip in a process known as "flashing." In order to flash the firmware, the microcontroller must have dedicated programming pins connected to a PC over SWD.

The keyboard profile that is loaded into the MCU in order to map MCU pins (and by extension their attached input devices) to various Windows functions is programmed into the device one time during the development process, and then it is never programmed again. In order to ensure that the device is not reliant on any background-running application to fulfill its basic mouse functionality, the mouse movement is handled in firmware.

## 4.4 Application Software

Through using the Python GUI libraries, we have made a user friendly software interface that is both efficient and has a sleek modern design. We started off with a baseline and referenced many other applications that allowed users to customize their device. Some of what we looked at included Logitech's LGHUB software, the Elgato Stream Deck, and VIA. We wanted the user to be able to select the key or encoder shown in the UI and options will be shown on what they want the selected input device to do whether it be a macro that either is a combination of keys or be able to open a program/application. As for the rotary knobs, we wanted to present options such as desktop volume, mic volume, monitor brightness, etc. Shown in the figure below, there are also presets found in the application folder. Ultimately, we want the application to be easy to navigate and provide full customizability for the device.

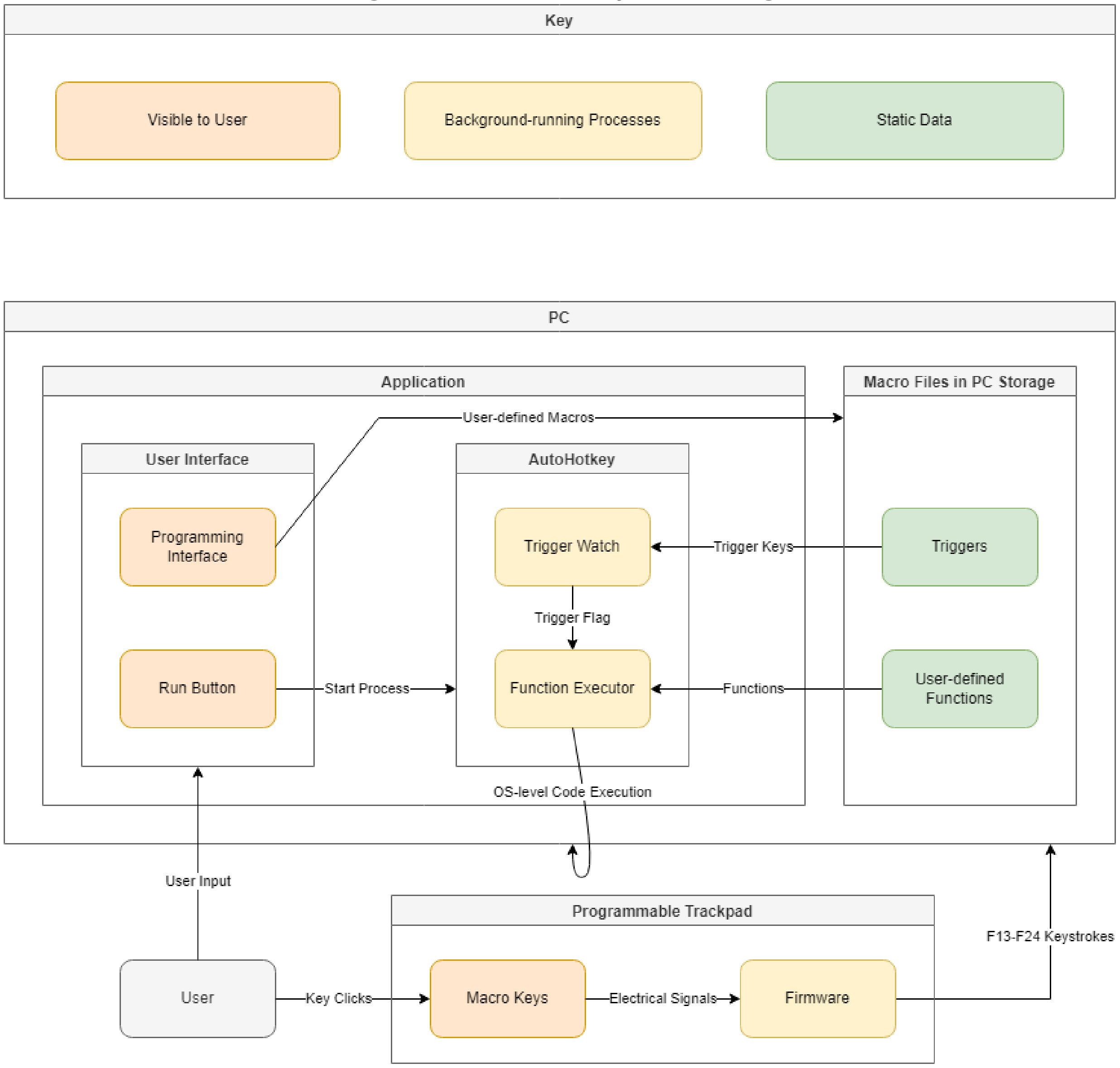
*Figure 73: Application Layout Mockup*



## 4.4. AutoHotkey

AutoHotkey (AHK) is a scripting language designed for Windows that focuses on creating macros and other various automated features with hotkeys or shortcuts. With these script files in place, containing simple to complex macros; the user application simplifies this process of creating a macro and “mapping” it to one of the external keypads. The application triggers these AutoHotkey scripts to change the functionality of F13 to something new like Open Chrome. AutoHotkey can be used standalone, but we have designed our own pre-made scripts for the user to choose from. This way, they don’t have to spend all of their time reading up on AHK forums to code up a script where instead they can just have their favorite macro right away. In the figure below, we showcase how AutoHotkey interacts with other software technologies in this project.

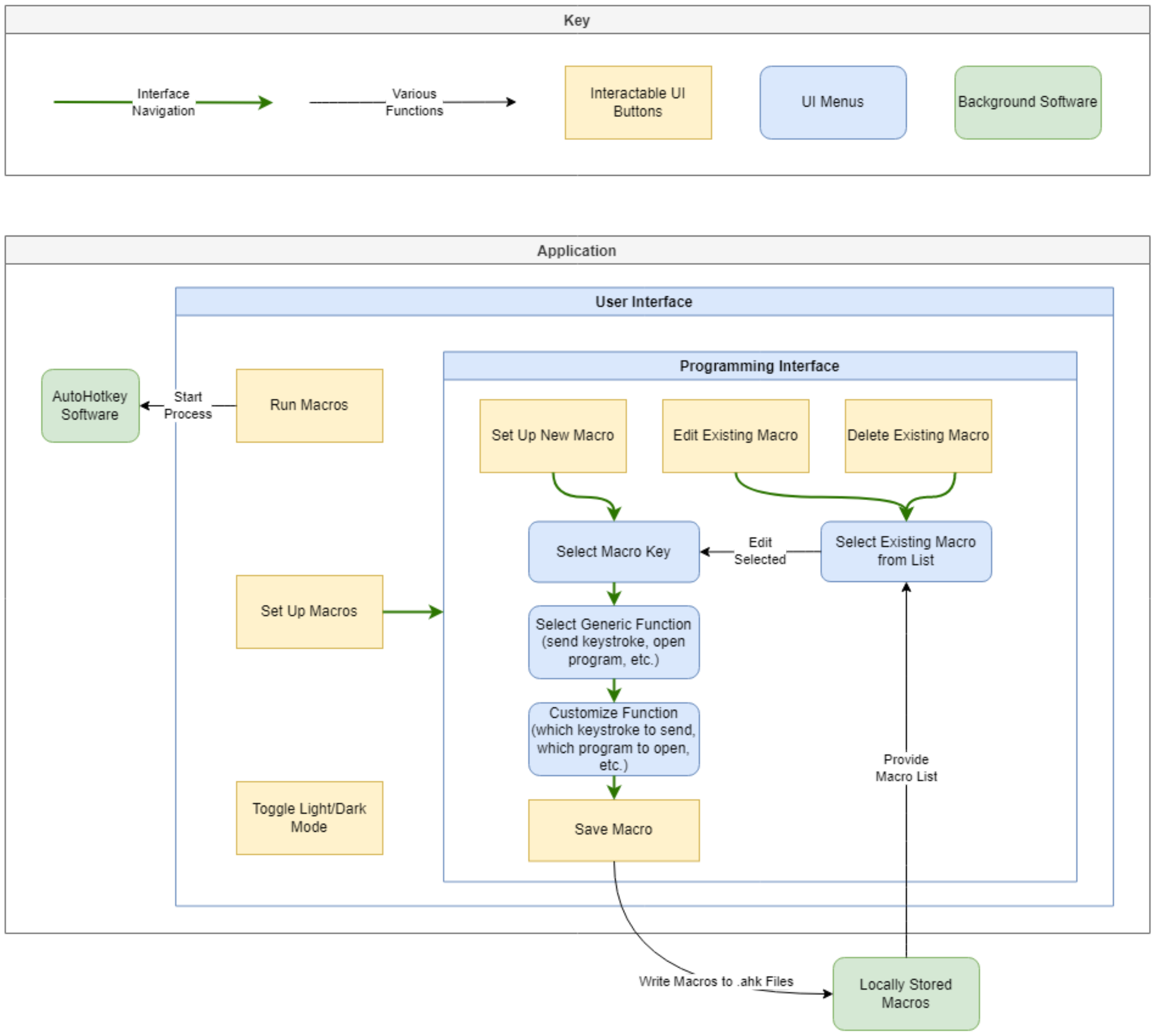
*Figure 74: AutoHotkey Block Diagram*



## 4.4.2 User Interface

For the UI, there aren't too many complicated features where we mainly allow the user to create a macro, delete a macro, and upload a macro to the keypads. The frontend displays these features in a simple way where the user can select preset macros in a dropdown and search bar. The background then interacts with a scripting language to change the function keys macro automation shortcut feature. Below is a diagram of how the UI is laid out and how it interacts with other technologies within the project.

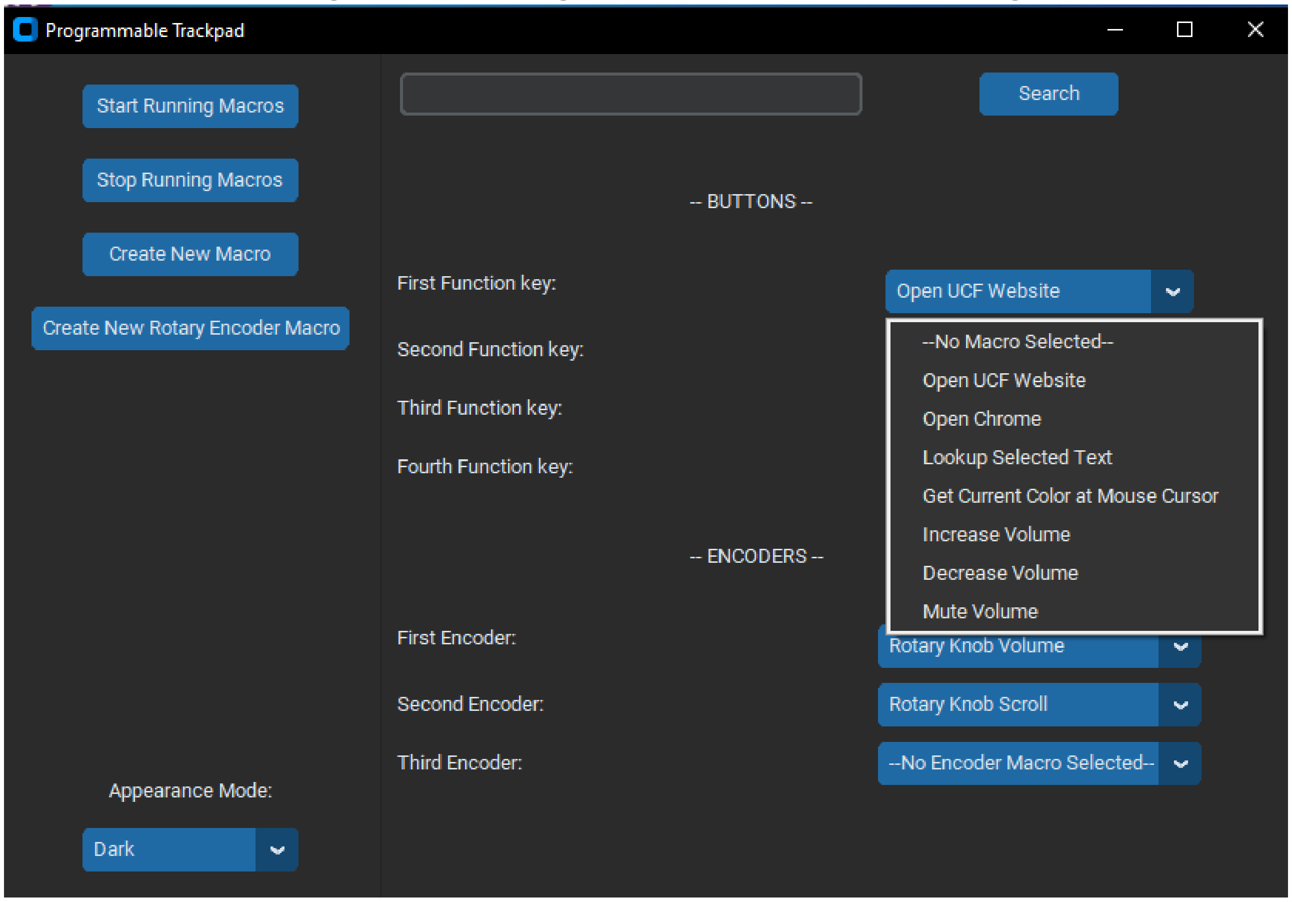
*Figure 75: User Interface Block Diagram*



In the design figure below, the main user interface function is the frontend design menu that allows them to create a macro preset. The user is able to give the preset a custom name and then choose from a list of default shortcuts or macros. All of this can be found in the left side of the ‘Create New Preset’ popup menu where the custom name gives the application a personal touch and gives the user familiarity with their macros. On the right side of the menu is where the macro is assigned to on the board. If the user decides that the preset shouldn’t be on the board but would like to save the macro for a later time, this menu allows them to do that.

The color design was different from the initial mockup where we didn’t want to set in stone a theme that overall wouldn’t be beneficial and/or pleasing to look at for the user. To give the user more customizability in the application, they can select a Light or Dark Theme mode in the settings. This pushes towards our advanced goal of giving the user more ability to customize their macro experience. If this project would be expanded upon and was replicated with a team without the constraints of being in an university environment, having these customizable settings could improve usability and user friendliness. Having software that appears outdated or looks too complex at first glance will heavily decrease the amount of daily users where that would be very negative if the device was placed in a real world business or marketing environment.

*Figure 76: Creating New Macro Preset Menu Design*

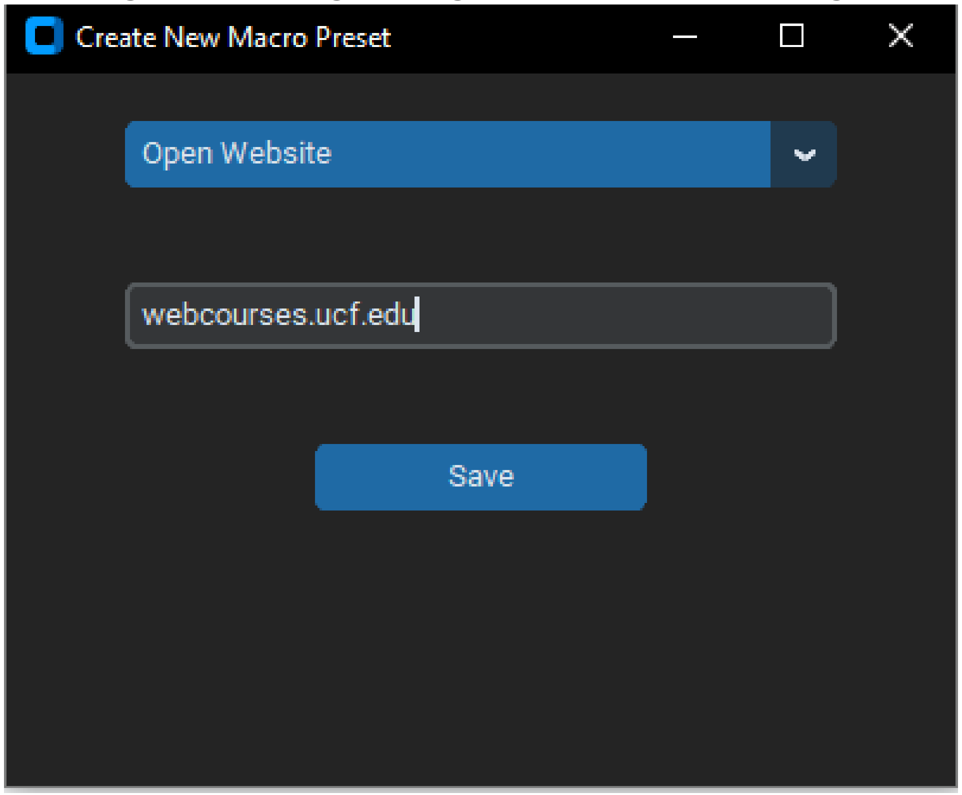


First Function key: Open UCF Website
Second Function key:
Third Function key:
Fourth Function key:
First Encoder: Rotary Knob Volume
Second Encoder: Rotary Knob Scroll
Third Encoder: --No Encoder Macro Selected--
Appearance Mode: Dark

With the previous figure, we are able to see how the user can customize their macro experience by creating any number of presets they want without having the limitations of the number of keys on the device itself. It is also essential for the user to edit them after the initial saving and creating of the preset. As seen in the figure below, the design has a lot of the same features as the Create Preset menu except for of course the Delete button.

Another design feature this diagram showcases is the additional option that appears after selecting the 'Run Chrome' macro. This example is showing that some macros give the user an additional parameter where they can go even further in their macro customization. So in this case the macro will open chrome and immediately go to UCF's webcourses site.

*Figure 77: Editing Existing Macro Preset Menu Design*

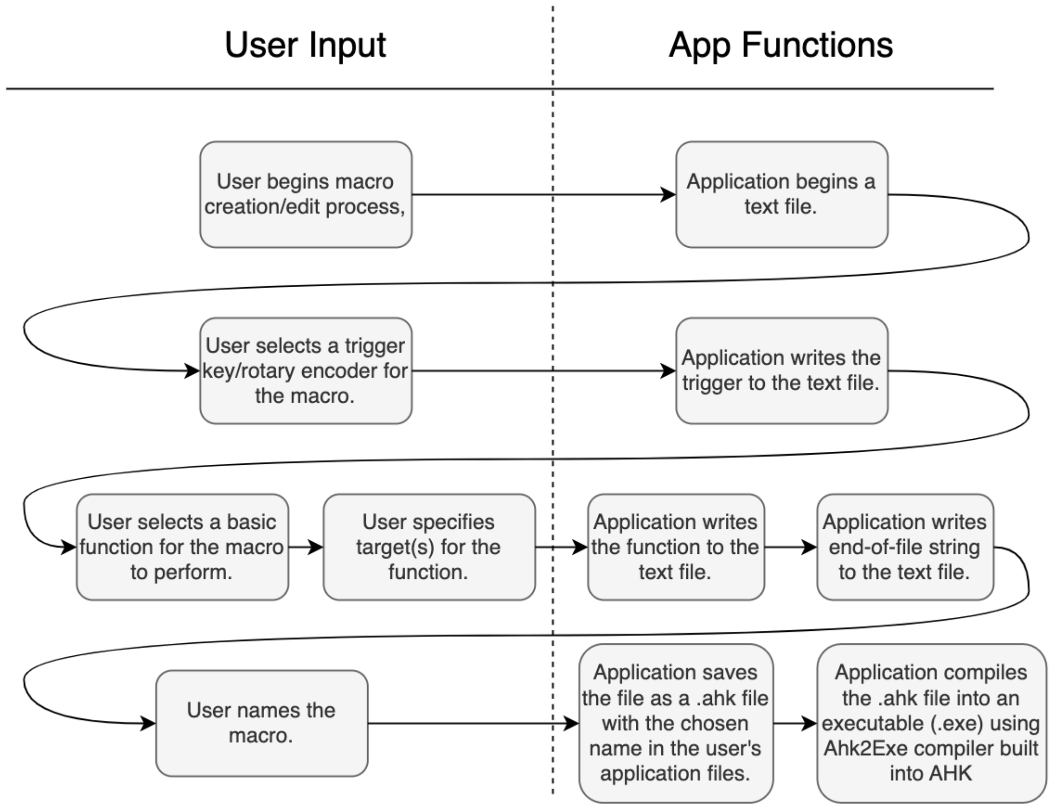


## 4.4.3 Macro Generation and Saving

When a user creates a macro in the GUI, that macro is saved locally as an AutoHotkey file. AHK files use a special syntax to specify scripts. Since the user is not writing these scripts, the application itself must generate the scripts. To do this, the application has a built-in list of strings that it can add to the AHK file. When the user selects a macro function, he/she is choosing from this list of strings.

Because the user only interacts with a GUI, the application is responsible for writing the code that will be run using AutoHotkey. Once the macro is finished, the application then saves it to a .ahk file stored in the user's local data. All macros are stored in the same location on a user's PC so that the application can access them on subsequent uses. At this point, the application has generated a single .ahk file remapping the functionality of the device's keys. The application will then compile the .ahk file into an executable using the built-in AHK compiler "Ahk2Exe". The application will then run this executable to produce the desired macro functionality. The following flow chart illustrates this process in greater detail.

*Figure 78: Macro Generation Flow Chart*


User Input
App Functions
User begins macro creation/edit process,
Application begins a text file.
User selects a trigger key/rotary encoder for the macro.
Application writes the trigger to the text file.
User selects a basic function for the macro to perform.
User specifies target(s) for the function.
Application writes the function to the text file.
Application writes end-of-file string to the text file.
User names the macro.
Application saves the file as a .ahk file with the chosen name in the user's application files.
Application compiles the .ahk file into an executable (.exe) using Ahk2Exe compiler built into AHK

Along with the essential feature of creating and writing an .AHK script file for the macros to run on the user's computer, there are also a number of other text files that are read and written to that give the user some quality of life features. Meaning it enhances their experience with the app but if this feature wasn't there it would break the overall functionality expected from the app.

The first file that is created and read into is the text file that stores all of the macros that you created during execution of the program. During this execution period of the program, the custom macros data is stored in an object array containing the macro's name, type, user input, and a unique ID. All of this data is used at some point during the application whether it's to send it to the .AHK script, display the macro name for the user to understand in the table and dropdown, and an ID to uniquely identify the macro so we can ensure to efficiently edit or delete the macro.

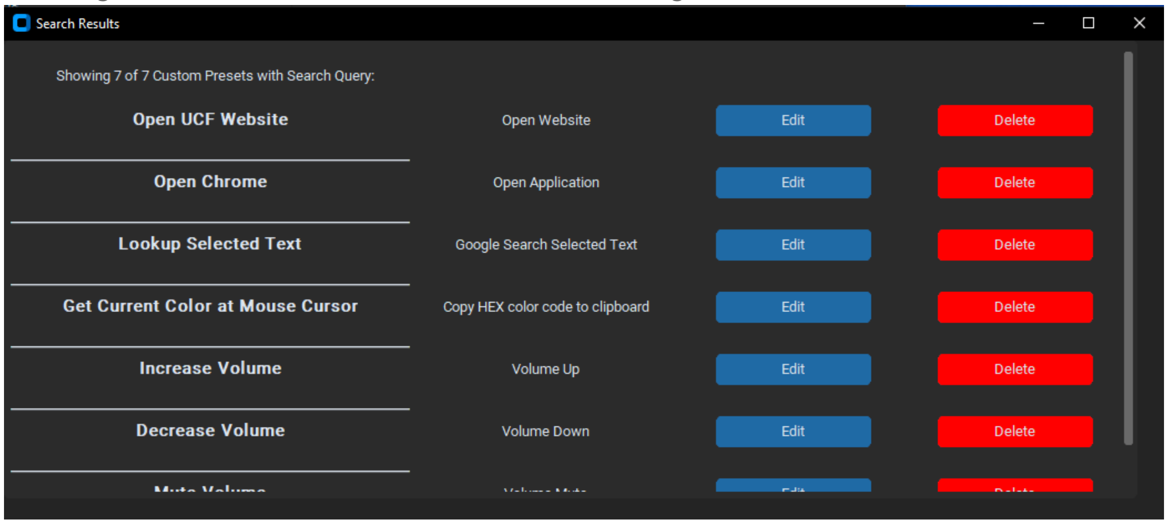
On exit of the program, the program goes through a function that takes that object array and stores all of the data in a format most similar to JSON. Due to the object's simplicity and not much variation between them we decided not to actually use a JSON library. However, if this project was going to be expanded upon with more and more complex data and macros it might be essential to swap to a more efficient data storage method such as using JSON files or having a DB file where we are using SQL queries to fetch and store data.

Therefore, once we stored the macros on a file, naturally the next function that was implemented was during initial load of the program that reads the data on the file and then creates that same object array for the user to use like they never even closed the program. The only other issue that was presented later is if the user selects a number of macros and “programs” them to the board, they would of course have all of their macros listed but they wouldn’t know which ones were selected after restarting the app. Therefore, another file was created to store the unique IDs associated with the selected macros which are then pre-populated into the dropdowns on the home screen.

Since all of the data is being successfully stored where each macro has a UUID (Universally Unique Identifier) on its creation, the last main functionality was to give the user the ability to delete and edit their macros after it’s creation. Overall the functions wouldn’t be very lengthy in terms of getting it to work where we simply have to remove the macro from the object array by searching for its ID. The edit functionality was a little bit more intuitive interacting the frontend Tkinter where the ‘Create Macro Window’ class was adjusted to take parameters and prefill the form inputs where the user can go through the same process and make any adjustments.

The logic behind these functions were not complicated where the most intuitive part was creating a table using our frontend Tkinter library that displayed important information for the user along with a few buttons giving them access to the edit and delete functionalities. Along with this we wanted to make the table dynamic in the sense where it would be created by the search query from the search bar on the homepage showing the results (by name or type) as well as any changes they make would be reflected accurately in the table and in the dropdowns on the home page. For example, if the user deleted one macro it should be removed from the table as well as from any of the dropdowns on the home page (including if that macro is currently selected). In the figures below you will see the overall format of the table and the example case of having a blank search query and then searching by name.

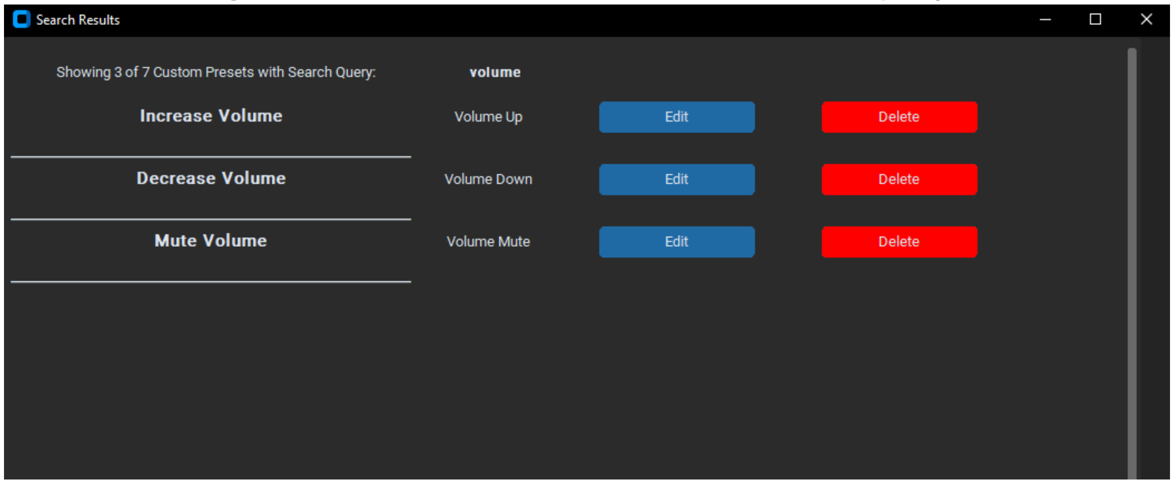
*Figure 79: Search Results Window showing all macros the user created*


Search Results

Showing 7 of 7 Custom Presets with Search Query:

| | | | |
|---|---|---|---|
| **Open UCF Website** | Open Website | Edit | Delete |
| **Open Chrome** | Open Application | Edit | Delete |
| **Lookup Selected Text** | Google Search Selected Text | Edit | Delete |
| **Get Current Color at Mouse Cursor** | Copy HEX color code to clipboard | Edit | Delete |
| **Increase Volume** | Volume Up | Edit | Delete |
| **Decrease Volume** | Volume Down | Edit | Delete |

*Figure 80: Search Results Window with search query*


Search Results

Showing 3 of 7 Custom Presets with Search Query: **volume**

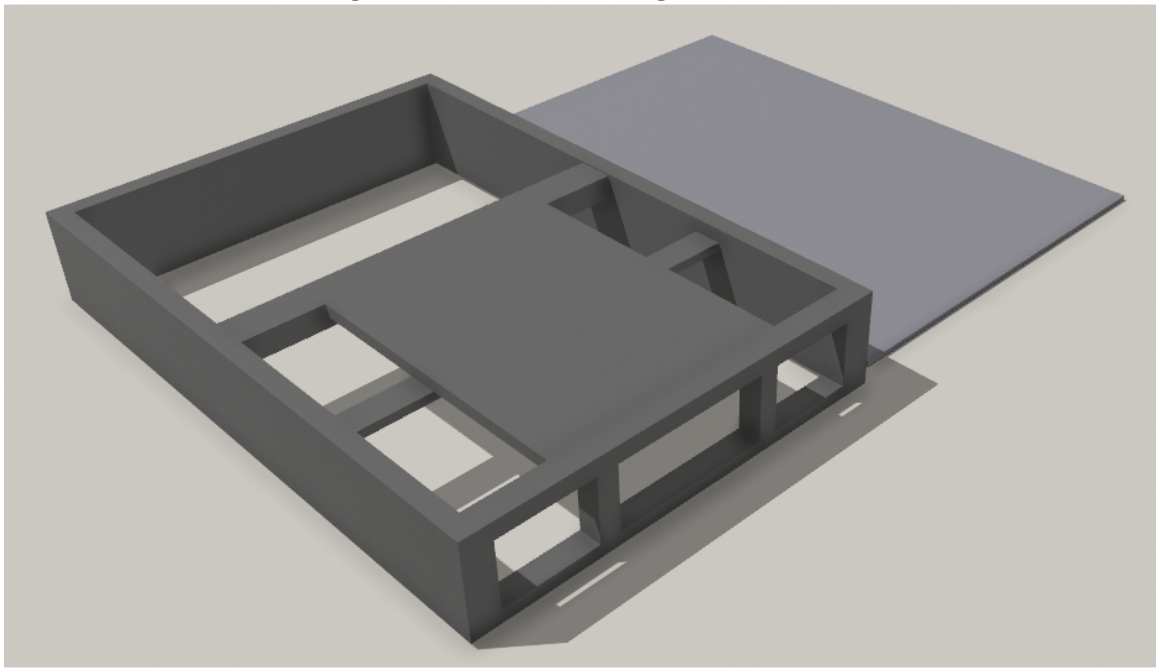
| | | | |
|---|---|---|---|
| **Increase Volume** | Volume Up | Edit | Delete |
| **Decrease Volume** | Volume Down | Edit | Delete |
| **Mute Volume** | Volume Mute | Edit | Delete |

## 4.5 Chassis

In the final product, all of the electronic components of the device are securely enclosed within a chassis made of mostly 3D-printed parts. This chassis is designed specifically with the needs of our PCB and input devices in mind.

### 4.5.1 3D-Printed Enclosure

*Figure 81: 3D Rendering of Chassis*

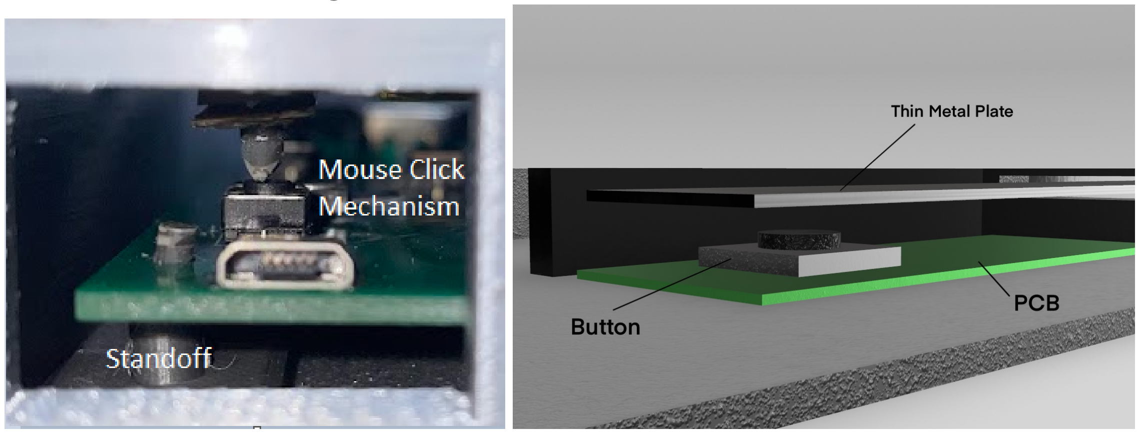


Shown in the figure is the 3D rendered chassis. In the physical implementation, there is a bottom plate where the PCB and electronics are mounted and the top casing to cover or surround all the necessary electronics. The top case has support for the trackpad as well as where the metal plates would be. The bottom plate is just a single plate where we would print standoffs for the PCB to help the ease process of mounting the PCB. The 3D modeling software we used was Shapr3D which is a beginner friendly software that allows visualization for the rendered object and we're even able to render different materials. The main concern that affected how the chassis looks on the inside is how we laid out the PCB for the touchpad to include the buttons. Based on our constraints and standards, we also needed to keep the chassis within the designated measurements, meaning a calculated way of forming the inside of the chassis as well as the PCB size.

### 4.5.2 Mouse Buttons

One design concern we came across was finding the right method for the touchpad buttons. We wanted a functionality that doesn't stray away from the traditional idea of the trackpad. The idea was to use a thin metal plate with some flex so that the finger is able to press on the button through the touchpad. Shown in the figures above, there is a slit on the bottom plate where the plate is inserted easily. That plate has enough flex to press onto the button under the plate to input the respective right or left mouse button. All of this functionality is under the trackpad unit without interfering with the main functionality. A more accurate representation can be seen in the figure below.

*Figure 82: Mouse Button Construction*



# 5 Prototypes and Testing

## 5.1 Hardware

In the prototyping phase of development, our team acquired several pre-built circuit boards using the chips we intend to utilize in our PCB. In order to better understand these components and the circuits that each one requires, we ran tests verifying that the parts can fulfill their prescribed purposes.

Once we ran the necessary tests, we developed a development board of our own, to test the components one by one, especially the microcontroller. After those tests were run, we moved on to creating and developing with the final PCB with all the final components.
The following sections explain the testing process for the various tests that we have run and intend to run in the future.

### 5.1.1 Battery Charging

The two battery management system boards discussed in section 3.2.1.4 were both used for prototyping. All power system tests use a 502248 battery, which is a generic 500 mAh lithium polymer battery. The goal of prototyping a charging system is to ensure that the selected chips and battery are capable of fully charging without overcharging over a generic USB connection within a reasonable amount of time, defined in Section 2.3 as < 3 hours.

#### 5.1.1.1 Procedure

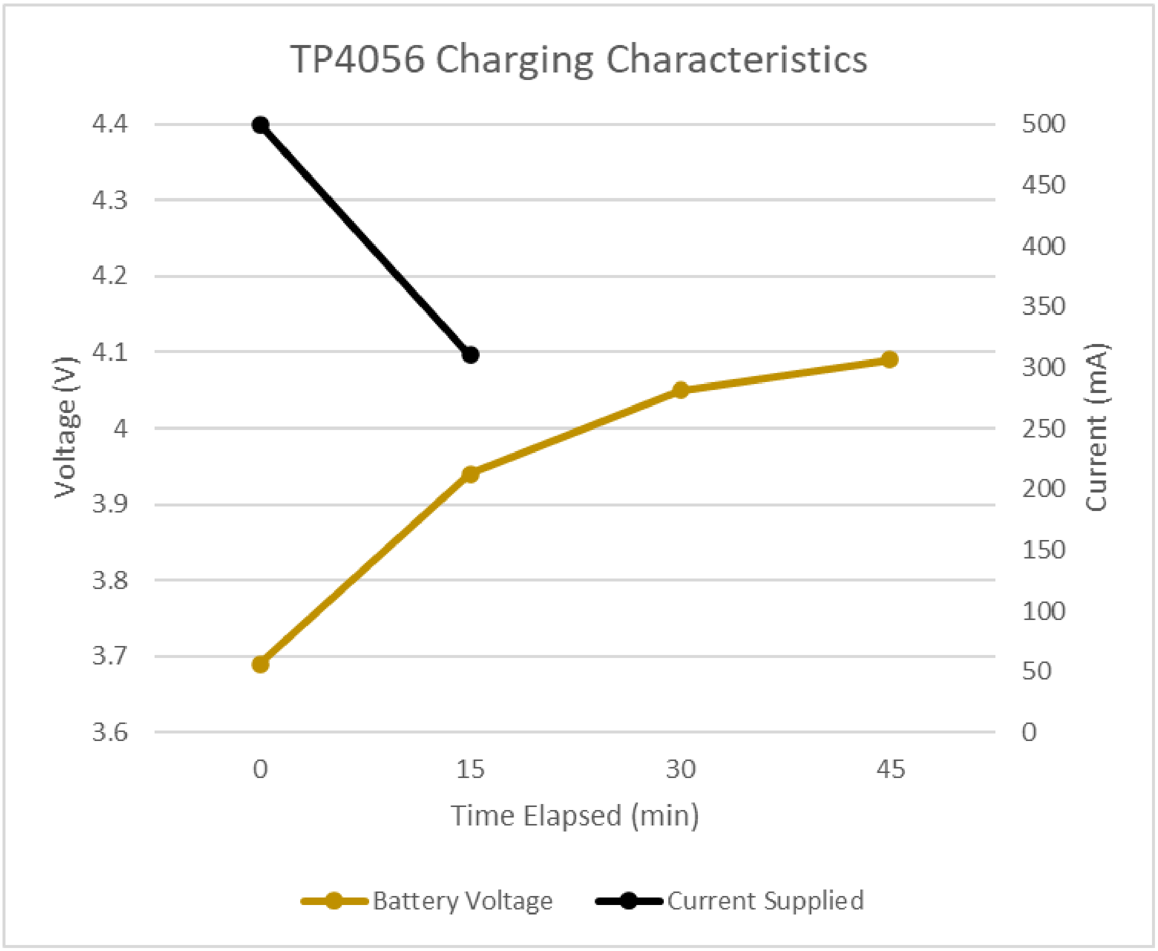
In order to test the charging system, we first discharge the battery a noticeable amount by connecting it to a motor. Based on the data given in Figure 14, it is reasonable to expect that the battery will read 3.7 volts when it is significantly depleted. If discharged further from this point, its voltage will drop rapidly.

For the purpose of prototyping, once the battery reads under 3.7 volts on a multimeter, then it is noticeably discharged. Next, the battery management system board is connected to a PC over USB. The battery is then connected to the output of the battery management system. This process should charge the battery. At regular intervals, the voltage of the battery and current flowing from the battery management system board will be checked using a multimeter. If the procedure is successful, the voltage will steadily rise until it eventually reads ~4.2 volts, at which point the battery management system will cut the flow of current to the battery. If the voltage does not steadily rise, if the voltage rises significantly higher than 4.2 volts, if the voltage does not reach ~4.2 volts in a reasonable amount of time, or if any components appear to sustain physical damage, then the procedure is a failure.

#### 5.1.1.2 Results

The initial test was carried out using the TP4056-based board. The experiment's results are shown in the graph below.

*Figure 83: TP4056 Charging Characteristics*

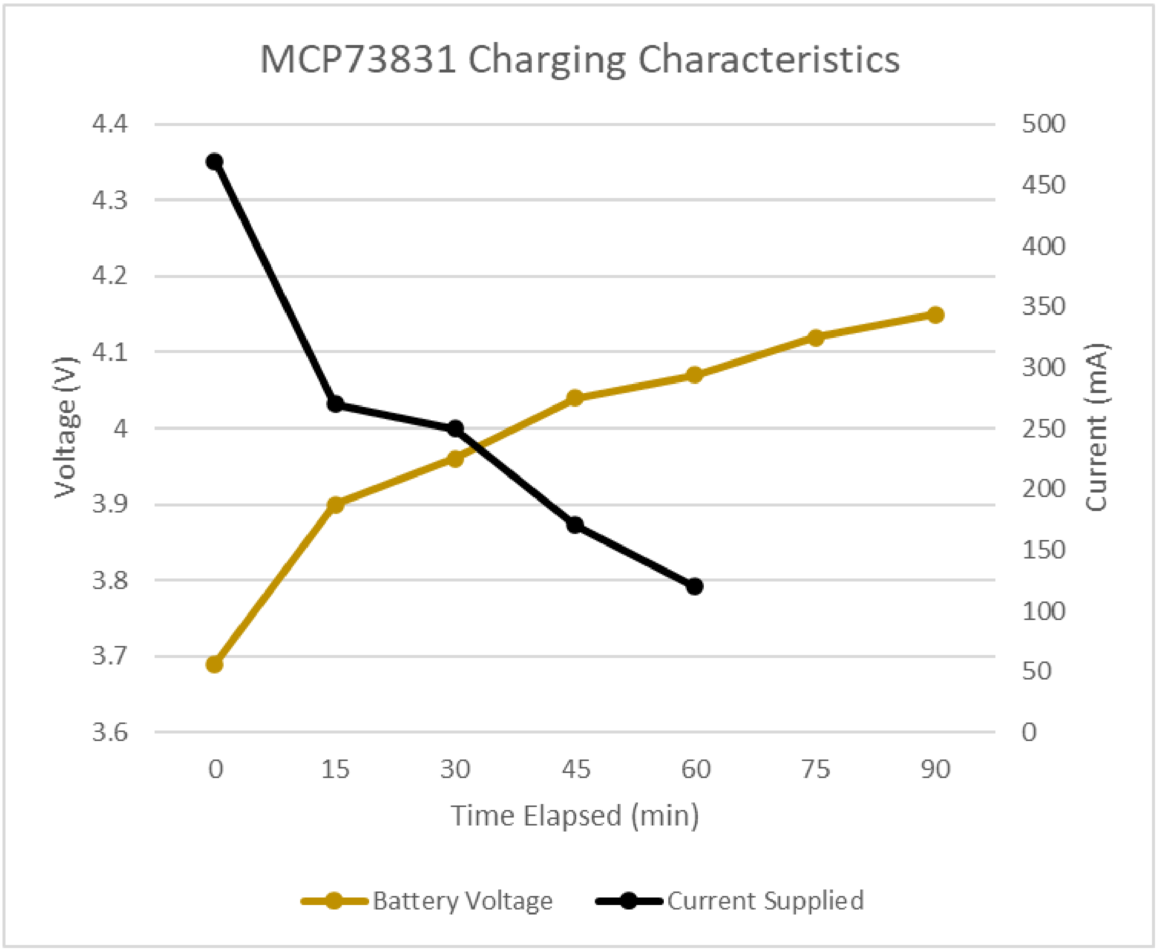


As is shown in the graph, the battery's voltage was 3.69 volts before charging. It rose quickly to ~3.94 volts, and then it gradually rose to 4.09 volts, at which point the battery ceased charging. This process took 45 minutes in total, which is a reasonable amount of time to expect a user to wait for a full charge. The current supplied to the battery steadily decreased as the voltage increased until it could no longer be read by the multimeter.

Because the battery was fully charged without overcharging, this procedure was a success. This proves that the TP4056 is a valid possible solution for the battery management system in our product.

A second test using the MCP73831-based board was performed after another full discharge of the battery. The results of this experiment are shown in the graph below.

*Figure 84: MCP73831 Charging Characteristics*



The results of this experiment were very similar to that of the TP4056. The battery's initial voltage was 3.69 volts. When connected with the MCP73831 board, its voltage quickly rose to ~3.9 volts, and then it gradually rose to 4.15 volts, at which point it stopped charging. This process took 90 minutes in total, which also falls within a reasonable time frame for the user to wait on a full charge. Like in the last experiment, the current decreased steadily until it was too low to read using a multimeter.

Because the battery was fully charged without overcharging, this procedure was a success. This proves that the MCP73831 is also a viable solution for the product's battery management system.

As is stated in Section 3.2.1.4, the MCP73831 is a more attractive option because of how simple it will be to implement in our design. Since both prospective chips succeeded in the prototyping phase, there is no reason not to use the MCP73831 in the final design.

## 5.1.2 Simultaneous Transmission and Charging

Once the battery charging capabilities of our components are verified, it is essential that we verify that the USB connection can be used for data transfer as well as power transfer.

### 5.1.2.1 Procedure

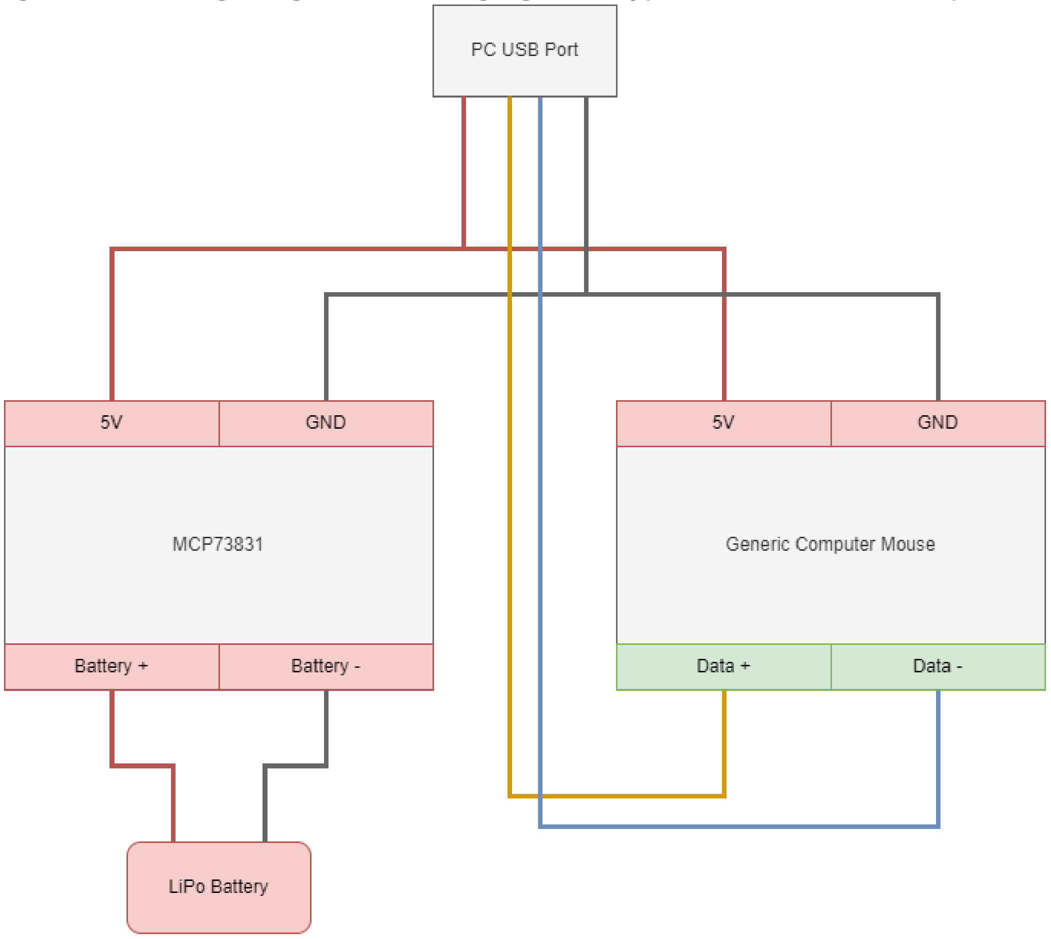
As in the previous prototype, the battery must be discharged at the beginning of this experiment, although it is only necessary to discharge the battery to the point that it will trigger the battery management system when connected. For the sake of the prototype, the battery will be discharged until it reads 4.0 volts on a multimeter.

After discharging the battery, the battery management system must be connected to a PC via a USB cable. The four wires of the USB cable must be broken out to another USB peripheral (such as a generic mouse), similar to the way the USB interface will be arranged on the PCB. If the procedure is successful, the USB peripheral will be functional the entire time it is connected to the PC, and the battery's voltage (measured by a multimeter) will increase over time. If the USB peripheral does not function as intended during the procedure or if the battery does not charge while connected to the battery management system, then the procedure is a failure.

### 5.1.2.2 Results

In the initial attempt at this experiment, we used a generic computer mouse with its four wires broken out onto a breadboard. These wires were then connected to the battery management system in the configuration shown below.

*Figure 85: Wiring Diagram of Charging Prototype With Generic Computer Mouse*



When configured like this, the MCP73831 would not activate, and the computer did not recognize the mouse. In an attempt to understand why the experiment failed, each node was probed with a multimeter. The following table shows the results of this probing.

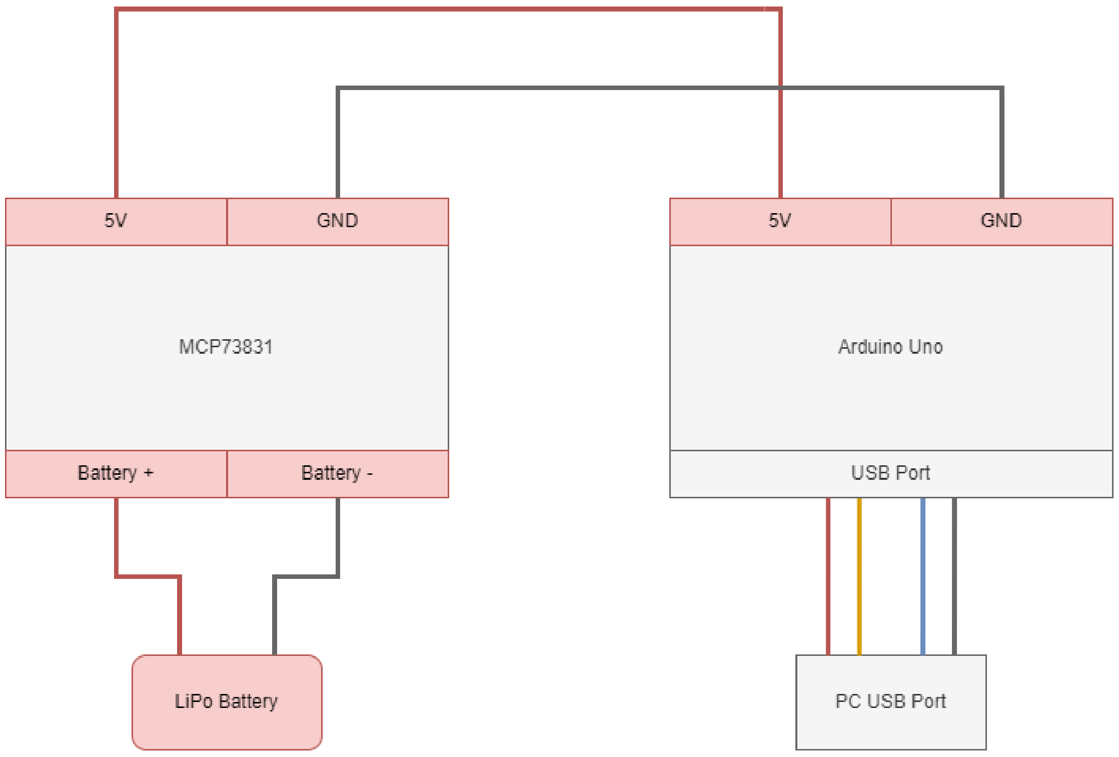
*Figure 86: Table of Voltages During Failed Simultaneous Transmission and Charging*

| Node | Connected Devices | Expected Voltage | Actual Voltage |
|---|---|---|---|
| USB VBUS | MCP and Mouse | 5 V | 2.5-3.5 V |
| USB VBUS | Mouse only | 5 V | ~3 V |
| Battery+ | MCP and Battery | 4.2 V | Battery Voltage (3.9 V) |
| Battery+ | MCP only | 4.2 V | 1-3 V |

These results confirm that the MCP73831 was not activated. In order to activate, the MCP73831 requires an input voltage of at least 3.75 volts, but in this experiment, it was only receiving 2.5-3.5 volts. When activated, the MCP73831 supplies a voltage of exactly 4.2 volts. However, in this experiment, the supply voltage pin was not constant. When attached to the battery, it matched the voltage of the battery. When floating, it fluctuated rapidly between 1 and 3 volts.

In addition to the MCP73831 not working, this procedure also shed some light on why the mouse would not work. When the mouse is plugged in on its own, its VBUS reads approximately 3 volts. Ordinarily, USB devices are supposed to operate at 5 volts. The fact that this mouse operates at 3 volts instead of 5 volts, even when it is entirely disconnected from the MCP73831, shows that it is not a good candidate for this prototype. A better candidate would be any device that is well known to operate at exactly 5 volts when plugged into a computer's USB port. In order to solve this problem, we used an Arduino Uno as our USB device. The following diagram shows how we set up this prototype.

*Figure 87: Wiring Diagram of Charging Prototype With Arduino Uno*



The Arduino Uno has its own breakout pins for 5 volt and ground, so the MCP73831 was plugged directly into those pins. Before wiring the system, the Arduino Uno was programmed to send basic serial data to the PC, similar to the way a generic computer mouse would. Upon wiring the system, the MCP73831 activated, the battery charged as expected, and a serial monitor revealed that the computer was receiving and interpreting the Arduino Uno’s transmitted data correctly.

As per the procedure described in the previous section, this prototype was a success. However, it did leave some questions about the solution in practice. The Arduino Uno is a complicated board with robust control over its pins, including the 5 volt and ground pins used in the experiment. This procedure has proven that the MCP73831 will work while the device is simultaneously transmitting data, but only under the condition that the USB port is supplying a steady 5 volts to the system. The Arduino drew a steady 5 volts from the USB port, but the generic computer mouse did not. If the MDBT50Q draws a steady 5 volts from the USB port when connected at its USB interface, then we can expect simultaneous transmission and charging to work. Without further prototyping, however, we could not be certain that the MDBT50Q's USB interface will draw the necessary 5 volts. It was possible that a 5 volt regulator would be necessary in order to ensure that the 5 volt components in the device receive 5 volts.

Once we began running prototypes with the custom test bed PCB, we were able to address this concern. When powering the MDBT50Q with a battery, the LM3671 buck converter, and the MCP BMS, the device was still able to communicate with a PC over USB. This USB link automatically charged the battery through the BMS, proving that the system does not require an auxiliary 5-volt converter.

### 5.1.3 Configuring the Touchpad Controller

In the beginning stages of testing the Adafruit Resistive Touchpad, we purchased the AR1100 breakout board to gain an understanding of how the AR1100 chip works. The AR1100 by itself is a universal resistive touchpad controller microchip that was supposed to be used in the final design, but we ended up scrapping this component because it added no noticeable improvement to functionality of the device. The breakout board contains the AR1100 chip, a 4 pin FFC connection port, a Mini-USB port and  In the prototyping phase, we will use the AR1100 breakout board as a way to easily test and troubleshoot the functionality of our touchpad. Once we have an understanding of how this breakout board works and what we can use it for, we will reverse engineer the board so that all of its components will live on our printed circuit board separately. This will include the AR1100 chip, the 4 FFC connection port, and any various resistors that are necessary for the circuit. The Mini-USB port will be left off in favor of through-holes. Below is an image of the breakout board and its PCB layout.

*Figure 88: AR110 Breakout Board*

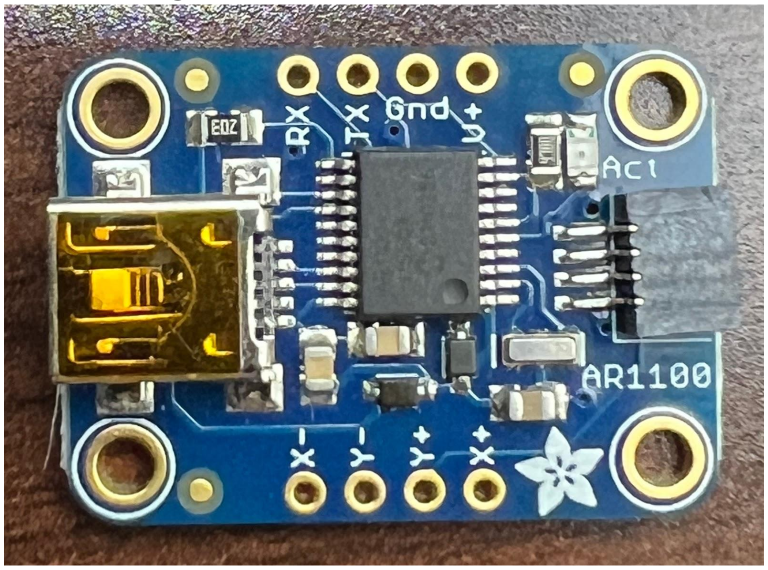
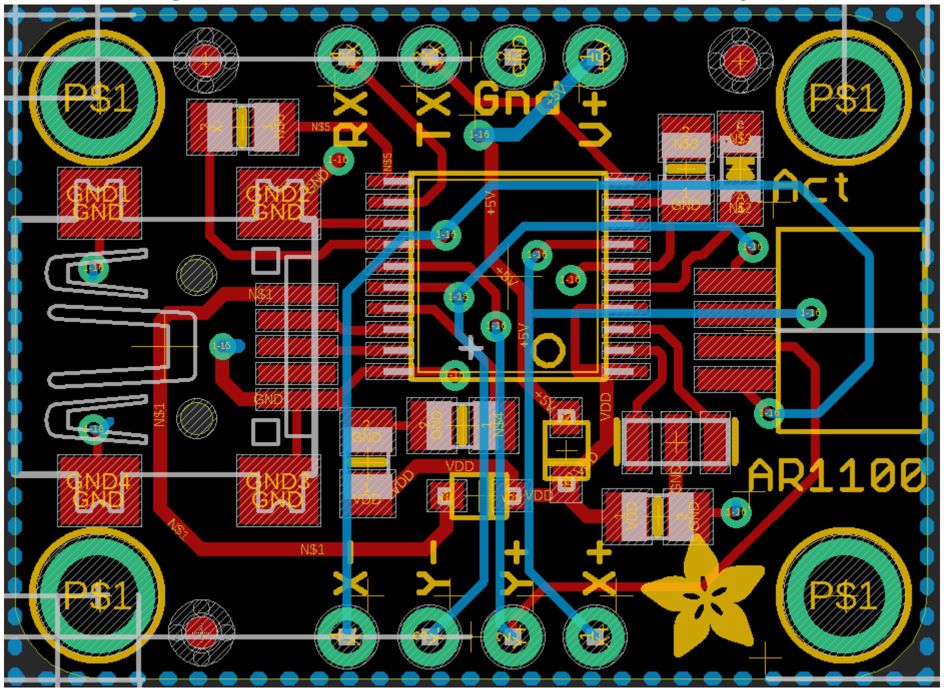
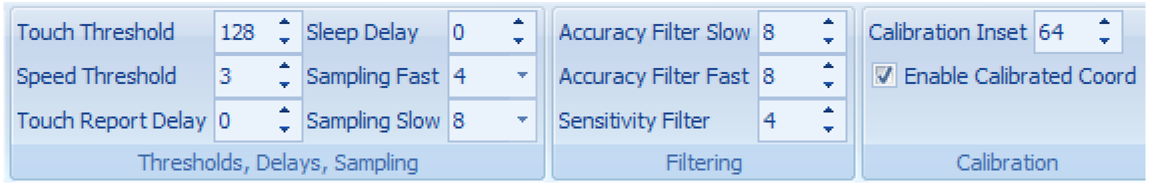


Figure 89: AR110 Breakout Board PCB Layout



In order to calibrate and configure the touchpad, it must first be connected to a computer that is running the configuration software. This is done by taking the 4 pin FFC on the touchpad and connecting it to the AR1100 breakout board, then using a Mini-USB cable to connect the board to a computer. At this point, the config software will detect the board and ask you what type of chip is on the board, how many pins it has etc. Once these values are set, communication will be established between the board and the software. In order to make configuration changes, the board must be in HID-Generic communication mode. In this mode, the touchpad will not act as a mouse. From here, there are a dozen different settings which can be altered, but I found that the touch threshold and sensitivity filter have the greatest impact on the use of the touchpad to control the mouse. Below is an image of some of the configuration settings.

Figure 90: AR110 Breakout Board Config Settings

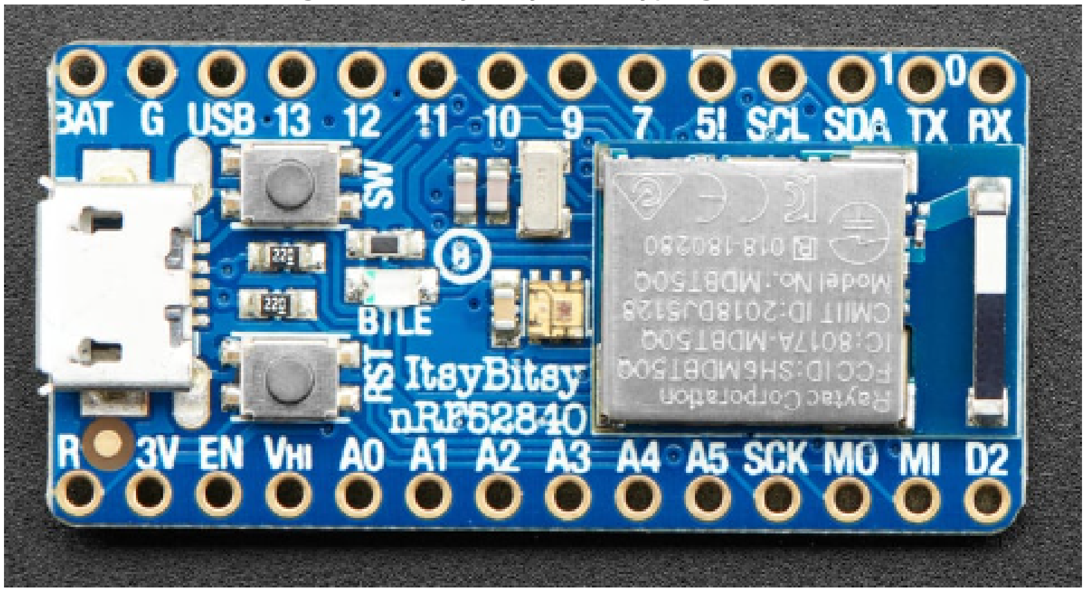


| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Touch Threshold | 128 | Sleep Delay | 0 | Accuracy Filter Slow | 8 | Calibration Inset | 64 |
| Speed Threshold | 3 | Sampling Fast | 4 | Accuracy Filter Fast | 8 | ✓ Enable Calibrated Coord | |
| Touch Report Delay | 0 | Sampling Slow | 8 | Sensitivity Filter | 4 | | |
| Thresholds, Delays, Sampling | | | | Filtering | | Calibration | |

The values in the image are the default settings when the AR1100 is first configured. These defaults serve as a great starting point, but require some tweaking. As it was, these settings leave the mouse pointer too “jittery.” So the first setting I adjusted was the speed threshold. Increasing this to 8 keeps the cursor more steady as it glides across the screen. Next, we raised the touch threshold to 225 so that it was not interpreting various points on the pad of my finger as inputs. This was enough to eliminate the jittery effect from the mouse cursor. If we actually incorporated the AR1100 chip in our final design, we would configure this chip a single time once it is mounted on the final PCB. The user would never have to perform that step. To reiterate, we did not use the AR1100 chip in our final design. We connected the touchpad directly into the MCU.

### 5.1.4 Microcontroller Selection

*Figure 91: ItsyBitsy Prototyping Board*


BAT G USB 13 12 11 10 9 7 5! SCL SDA TX RX
ItsyBitsy nRF52840
R 3V EN VHI A0 A1 A2 A3 A4 A5 SCK MO MI D2

The microcontroller development board we ended up using for prototype testing was the Adafruit ItsyBitsy nRF52840 Express. It uses the Nordic nRF52840 Bluetooth LE processor built around the 32-bit ARM Cortex M4 CPU running at 64 Mhz and featuring a 1 MB flash with 256 KB of SRAM. The main thing we'll be working with is the Arduino IDE and Circuit Python using the native serial information transferring and the keyboard and mouse HID libraries catering to our macro key and touchpad control. NRF also comes with their own SDK supporting the nRF52 Series with development of Bluetooth Low Energy integrating the Zephyr RTOS and more. This would in theory make ZMK more streamlined but KMK would provide more options in this case for mouse HID. From here we are able to test whether using KMK or using the native Arduino libraries and Circuit Python will be more optimal for what we want to achieve.

#### 5.1.4.1 Procedure

The relevant system to verify in the nRF52840 is the Bluetooth connectivity. Other prototypes can use the microcontroller's GPIO pins and other systems, and the functionality of the microcontroller will be verified in those prototypes. For Bluetooth specifically, however, it is important that we ensure the process of connecting the device to the PC is possible and simple enough to be replicated by the end user.

The first step of the prototype is to upload firmware to the prototype board that configures the chip as a Bluetooth device advertising a connection. This firmware can be taken from example code in the CircuitPython libraries. Once this firmware is uploaded, the Bluetooth connectivity can be tested.

To test the chip's Bluetooth connectivity, the prototype board must first be powered through an external source. Then, the PC's Bluetooth must be turned on. When the PC's Bluetooth is activated, it will automatically begin searching for devices with which to form a connection. Within a few seconds, a device should appear to the PC with the name specified in the firmware. The developer will click the connect button on the PC to form the Bluetooth connection. If the PC displays a message saying that the connection was successful, and the prototype board's Bluetooth LED changes to a constant on-state, then the prototype was successful.

We initially went with a few iterations of our development board before deciding on the final design. The main procedure was being able to program the microcontroller. This required connecting to the microcontroller using an SWD programmer. From here we can install the necessary bootware to program the MDBT50Q.

#### 5.1.4.2 Results

Simply testing the keyboard and mouse HID code through USB connection was very simple. The only difference between the USB connection and the Bluetooth LE is that we need to add code initially to set up a radio for the Bluetooth antenna in order to connect to the computer and make the device discoverable. Once that is done, all the information being thrown at the computer is set between a while loop for the bluetooth connection. Much testing was done to go about getting this working and details are shown in section 5.2.2. Setting up a connection was as simple as advertising the board as a peripheral as well as through UART connectivity. It was also very important to emphasize that we needed to import the HIDServices() function to set the board as a peripheral, otherwise it would connect the board to the computer as a central computer.

We found that installing the necessary bootware for CircuitPython did not work on our custom PCB and this led us to shifting towards using Arduino. Although all testing the input units worked, bluetooth was unsuccessful despite looking to online forums and professional advice.

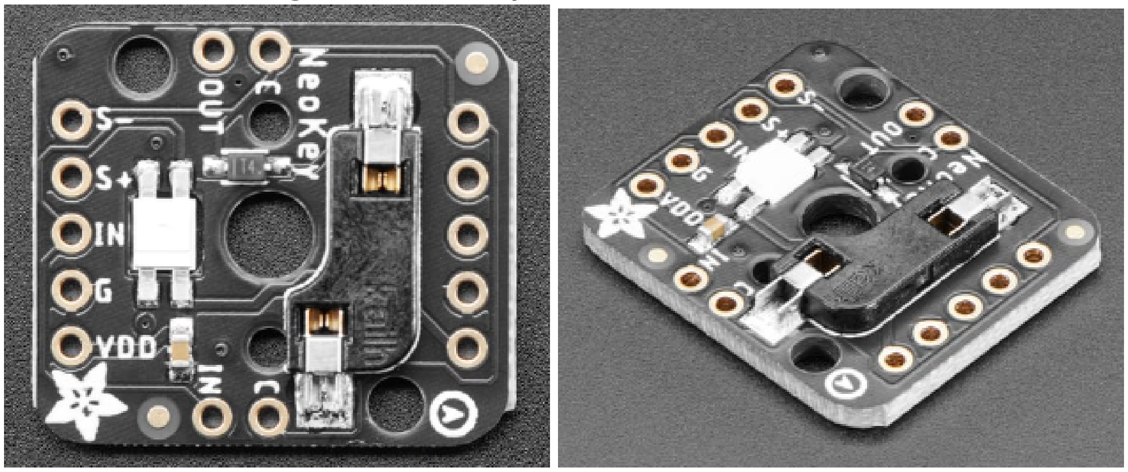
### 5.1.5 Input Unit Testing

This section focuses on the testing methods used for the overall functionality of the device. We first focused on testing through USB connection to implement the firmware but once we found bluetooth to be unsuccessful we scratched that idea out.

### 5.1.5.1 Mechanical Switches

In order to test the Kailh hot-swappable sockets, we used the NeoKey Socket Breakout for Mechanical Key Switches, a breakout board that includes the circuitry for each macro key on the device.

*Figure 92: NeoKey Socket Breakout Board*



It's a 0.75" x 0.85" PCB that can fit any Cherry MX and Gateron style switch. With this we can physically map out the key switches to the microcontroller. Once we are able to get things rolling using the breadboard we could move onto using a PCB with sockets for the microcontroller including all the connections in between for the switches and encoders.

Testing this device was simple. We connected the switch to a breadboard with an LED. If pressing the switch closes the circuit to light up the LED, then the procedure is successful, and the Kailh hot-swappable socket is proven to work for our purposes. This test is somewhat trivial, but it is important to have the switch on hand so that we may verify that the Gateron keys can fit the switch as intended.

Later prototypes used this device in conjunction with the microcontroller to simulate realistic use cases. The generic rotary encoders and mouse buttons also were tested in the exact same way as the Kailh hot-swappable sockets since they behave as simple digital input devices.

### 5.1.5.2 Rotary Encoders

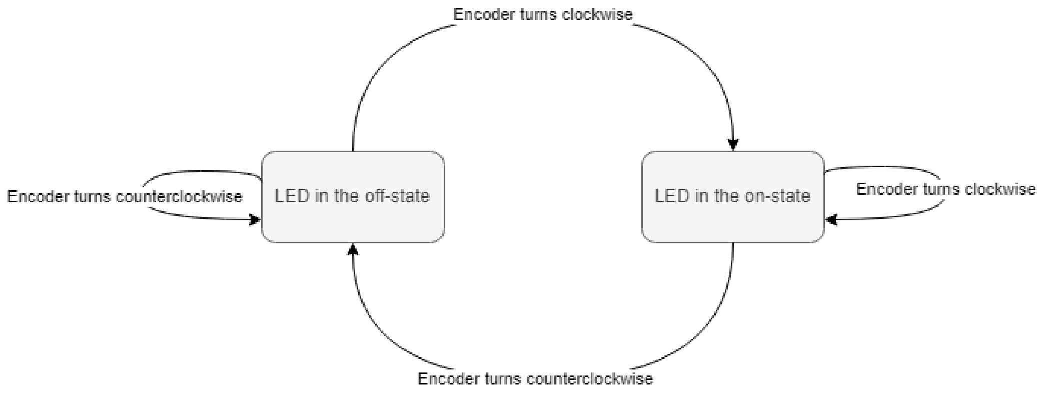
The rotary encoder we used is the PEC11 series 12 mm incremental encoder shown in the figure below. It features a 4 PC pin configuration with 0 detents meaning it is able to have 12, 18, and 24 pulse readings. It also has a push switch to be used as an input/output option as well. The main use for these rotary encoders will be for audio digital inputs which can be configured for application audio, microphone levels, desktop brightness, and window switching. In conjunction with Arduino, we are able to acquire information based on its position using the native libraries which includes volume control and play/pause features. In practice, however, these functions will only be implemented through AutoHotkey.

Figure 93: PEC11 Rotary Encoder

In order to test the rotary encoder, we connected it to a microcontroller. A simple circuit would not be sufficient to gauge the effectiveness of a rotary encoder because the encoder relies on positive and negative edges to deliver data.

The rotary encoder was configured as an input to the microcontroller. If the microcontroller detects a clockwise turn, then it will turn an LED on. If the microcontroller detects a counterclockwise turn, then it will turn the LED off. The following state diagram illustrates the proper functionality of this prototype. If each of the arrows can be followed in practice, and each one results in switching to the next correct state, then the experiment is a success.

Figure 94: Rotary Encoder Prototype State Diagram



## 5.1.6 Reading Touchpad Inputs

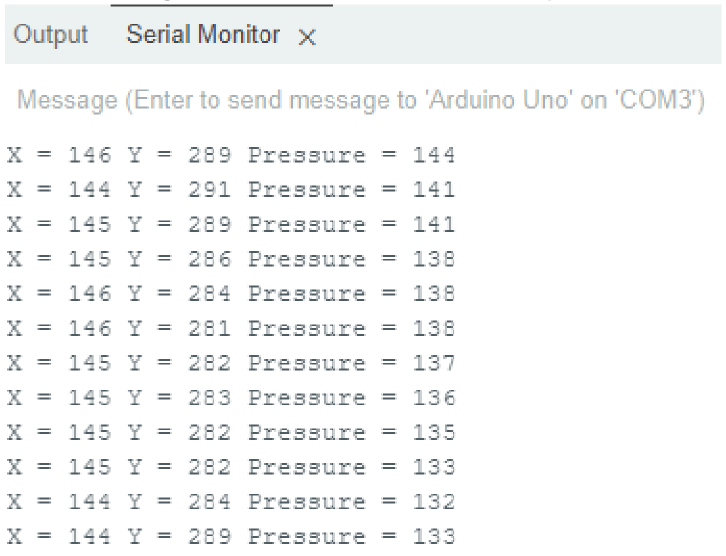
The touchpad is one of the main cornerstones of this project and as such we considered a few different methods of implementation. One idea was to route the inputs of the touchpad through a resistive touchpad controller (AR1100). This method has been discussed already in previous sections. Another idea was to cut out the middleman entirely and have the touchpad feed user inputs directly into the microcontroller, which is ultimately what we did in the final product. In this subsection, we will discuss the procedure that we used when creating a prototype circuit of the touchpad and MCU, whether or not it is possible to operate this way, and how we interpret the outcome of this experiment.

### 5.1.6.1 Prototype Environment

Functionality of the touchpad was first tested using an Arduino Uno development board. Using the Arduino Uno for initial tests was more convenient since one team member could conduct tests on the touchpad, while another team member could perform separate tests with the MCU, simultaneously. Beginning to write some test code in the Arduino environment was also very convenient to use as a starting point because of the massive amounts of documentation and other resources that exist. Moreover, Adafruit has released a C++ library for their resistive touchpads which comes with several useful functions and classes.

The class used in this prototype was “TouchScreen.” This class is used to instantiate an object that will hold a few important variables. It is meant to hold the pin number of the positive X/Y pins, pin number of the negative X/Y, and the resistance of the touchpad. The resistance is used when calculating the pressure threshold that will be considered a valid input, which is very useful for avoiding any accidental touches. This touch pressure is designated as Z. In looking at the code, the first thing that must be done is define certain pins as variables. In my case, the positive X and positive Y pins on the touchpad were linked to analog pins, and the negative X and negative Y pins were linked to digital pins. In the setup function, serial communication is started at a 9600 baud rate. Moving onto the infinite loop portion of the code, the first step is to create a TouchScreen object and pass through the 4 pin variables, as well as the resistance. Now we can instantiate a second object of class type “TSPoint.” This class serves as a structure to hold the actual values of the X/Y coordinates as well as the touch pressure, Z. After this object is instantiated, we call the member function “getPoint().” This function will collect several samples from the touchpad using the 4 pins from the touchpad, then average out the values to account for noise. Upon completion, the exact value of X, Y, and Z will be stored within the TSPoint object. To view these values upon a new touch, we simply print these 3 values to the serial monitor. An example of the serial monitor output is shown in the figure below.

*Figure 95: Arduino Serial Output*



```
X = 146 Y = 289 Pressure = 144
X = 144 Y = 291 Pressure = 141
X = 145 Y = 289 Pressure = 141
X = 145 Y = 286 Pressure = 138
X = 146 Y = 284 Pressure = 138
X = 146 Y = 281 Pressure = 138
X = 145 Y = 282 Pressure = 137
X = 145 Y = 283 Pressure = 136
X = 145 Y = 282 Pressure = 135
X = 145 Y = 282 Pressure = 133
X = 144 Y = 284 Pressure = 132
X = 144 Y = 289 Pressure = 133
```

The hardware configuration was very simple. We started by connecting the touchpad to the AR1100 breakout board as a way to interface with its connections via header pins, The AR1100 chip was NOT used in any way. From here, negative X was connected to analog pin A3, negative Y was connected to digital pin 9, positive Y was connected to analog pin A2, and positive X was connected to digital pin 8. The arduino was connected to a laptop for power. Below is an image of the set-up.

Figure 96: Touchpad Prototype using Arduino Uno

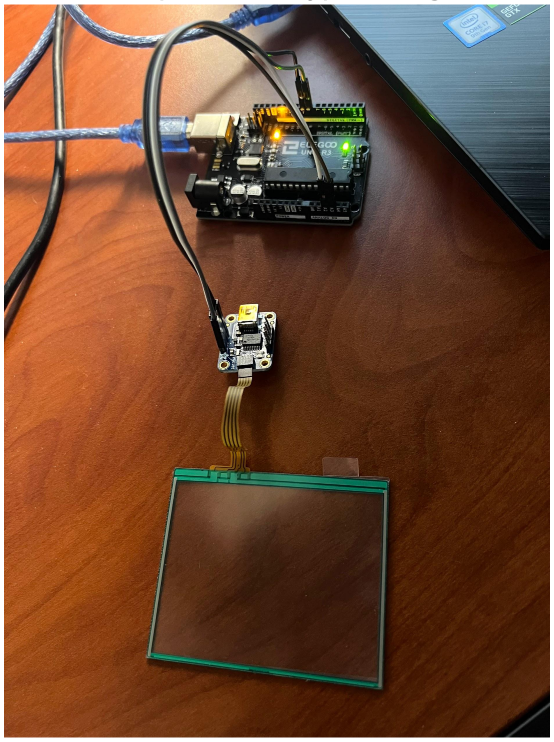


Figure 97: Additional Prototype Close-ups

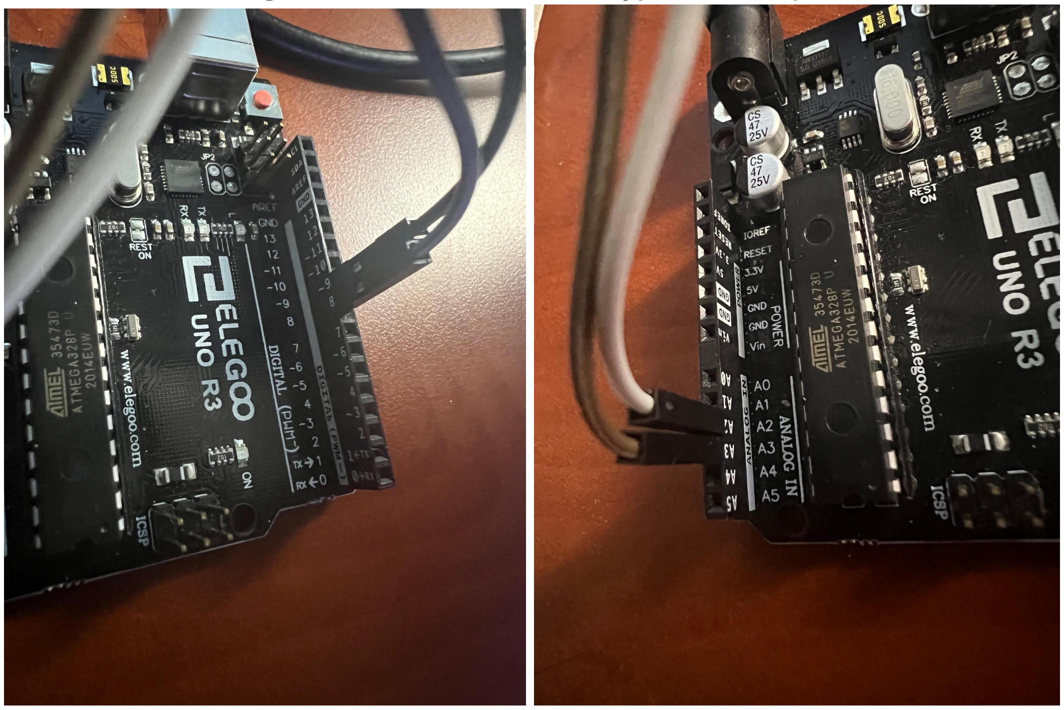
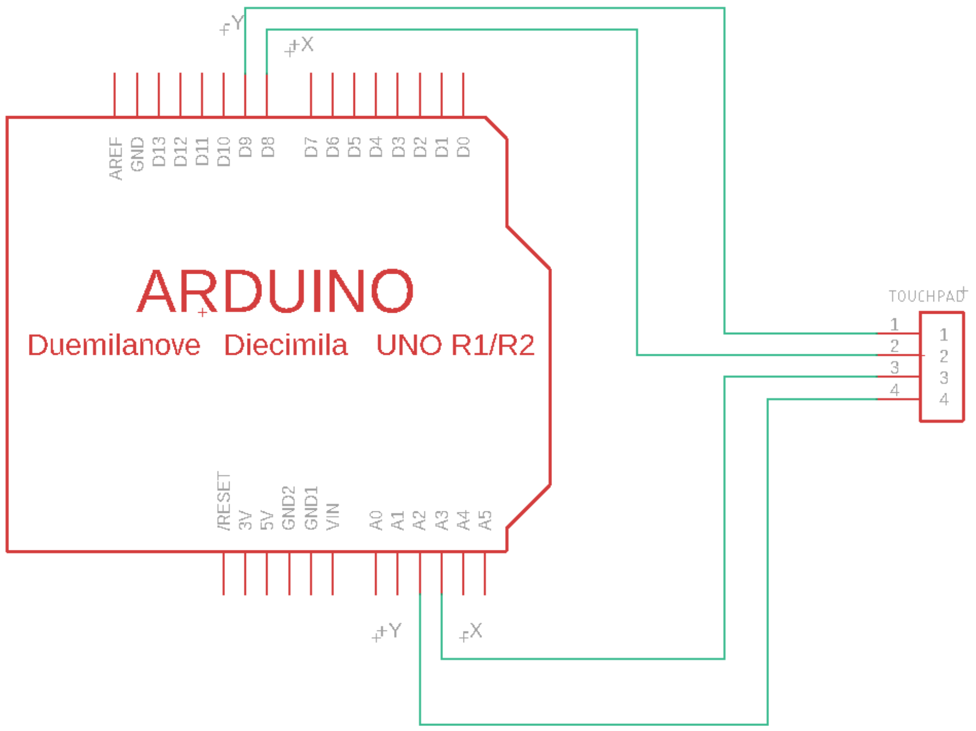


*Figure 98: Touchpad Prototype Schematic*



### 5.1.6.2 MCU Environment

The experiment in the prototype environment was a success, so we shifted our focus towards the actual MCU environment. The purpose of this next experiment is the same as before, to ensure that we can retrieve user inputs from the touchpad without the use of an additional touchpad controller chip. This time we will be testing with the ItsyBitsy nRF52840 Express. In this scenario, the nRF52840 will be programmed using CircuitPython, and the serial output will be read using an extension within Visual Studio Code. Conveniently for us, Adafruit has released yet another library for resistive touchpads. This library is for Python and contains many useful functions that makes interfacing with the touchpad a fairly rudimentary task. The following paragraph is the test procedure.

The first step in writing this test script is to initialize a new object of class type “Touchscreen.” This class is defined by the Adafruit library and it contains several useful functions. Firstly, it initializes a few important variables. It initializes 4 objects as pins, sets the touchpad resistance value, number of samples, touch threshold, calibration values, and size of the touchpad. For the simplicity of this test, this is all the preparation that needs to be done. Now, we call the member function “touch_point” on the Touchscreen object. This will print the X and Y coordinate values, along with the Z value. The Z value in this case is the amount of pressure detected on that particular input.

Due to the success of the previous touchpad experiment, and the successes of all the other modules we have tested in conjunction with the nRF52840, we were confident that this test will also be successful when we conduct it later on with all the components together. In another test in conjunction with the nRF we connected 4 wires going from the touchpad to the ItsyBitsy board and the ItsyBitsy board will be connected to a PC to receive power and to output to a terminal. Three of the touchpad wires will go to three digital pins, and the fourth will go to an analog pin. Then the script will be run on the board, and we will monitor the serial output for incoming data. If we receive valid X and Y coordinates, then we can deem the test a success. This prototype piece also led us to believe that with all the final components together, the trackpad would work. It wasn't until we had gotten our PCB final design with all the input units put in conjunction with another, the tests we ran to see if it worked gave us interesting results. Apart from the accuracy of the trackpad, we saw that this input unit worked perfectly on the final PCB.

## 5.2 Firmware

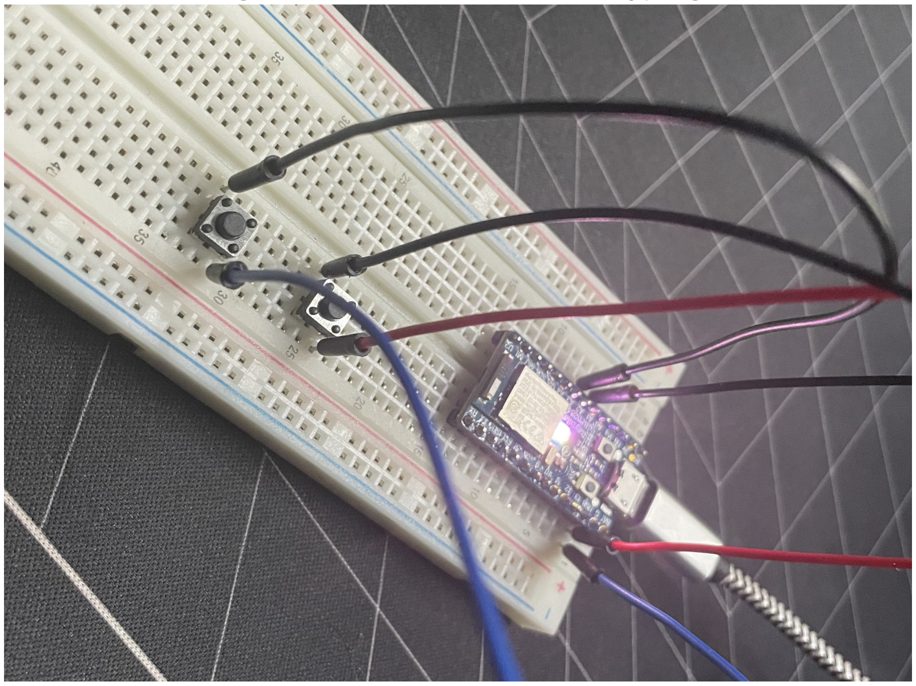
Firmware was a very big component that we needed to get right. Since it was something that would be developed in production before it is brought out to the consumer. One thing to note is that using the microcontroller development board will give us a bit of a boost in prototyping, however where in actually developing our own PCB we would need to use a SWD programmer to program the chip if we were to use any of the following support programs.

### 5.2.1 Macro, Rotary Encoder, and Mouse Functional Development

The CircuitPython and the Arduino IDE have library folders for both keyboard and mouse HID functionalities. The library offers keycodes and based on the key that is pressed, it will execute the action given to it. Before we map the keycodes, we must first map the pins on the board to the respective NeoKey and give it a function keycode. This also can be applied to the connected buttons for the touchpad. From there we can then write code to act as a macro, and with the combination of Python scripting with AutoHotkey, we are able to bridge the gap between the user and the microcontroller. The mouse HID is a very similar situation where we are able to take the x and y serial information then the mouse will act respectively.

Our main method of coding the firmware, Arduino, has a library bundle that includes all the necessary libraries for the keyboard and mouse HID functionalities. Through some testing we were able to map the pins to a test button where that button would be mapped to a keycode, our case being the letter 'A'. This would turn that button into simply a pressed key that the computer would interpret. The picture shown below is the testing for two buttons to be interpreted as keyboard keys.

*Figure 99: Initial Macro Prototyping*



Adafruit comes with libraries for rotary encoder support and implementing this functionality was very similar to the keyboard HID. The PEC11 has 5 pins in which 3 are for the rotary function and 2 for the button. For now we will disregard the 2 pins for the button. Two of the three pins are connected to two separate pins on the ItsyBitsy and the last pin on the encoder is connected to ground. The two pins on the PEC11 are linked to channel A and channel B. This way we can set each channel to a pin and interpret them in the code and using the IncrementalEncoder function with the ItsyBitsy pins as parameters, we are able to set it to a singular variable as well as determine the relative rotational position based on two series of pulses. For further testing, along with instantiating the pins on the encoder we instantiated consumer control for volume control. We can now track the position of the encoder and map function accordingly.

As for the mouse functionalities, we only really wanted a way to interpret serial information from the X and Y coordinates. In lieu of the actual touchpad, we used a joystick to send X and Y coordinates then in the mouse HID code in CircuitPython, it would interpret it as mouse movement. We first instantiated the X and Y axis using the “analogio” library and setting the pins to the respective axis. From here we simply use the mouse function to move it to the direction we are getting the input from. Since for the purpose of testing we used a joystick, we worked around it by using the potentiometer values and obtaining the voltage for set X and Y coordinates. This initial testing was first done through CircuitPython, but once we found that through installing the necessary bootware with our custom development PCB we had to take the route of only programming the firmware through Arduino. Once we were able to have the components for testing the input units we could properly test each unit with the respective libraries all through the Arduino IDE.

#### 5.2.1.1 Backup Firmware

As a backup before we finalize the GPIO pins for the final board, we explored the option for setting up KMK as the main firmware option. Installing this firmware was simple by dropping the library into the plugged in development board. Supposedly we would want the negative pins on the key to connect to ground, but with KMK, since it was made for traditional keyboards, it implements a matrix like solution to have the keys activated. The example shown below represents the pins and how they interact with the keycode they are set to. For the sake of testing we can take GPIO pins A1-A4 on the testing board and create a 2 x 2 matrix. The switches that are connected to pins A1 and A2 are then shorted to return the input of F13 and so forth for the other pins.

*Figure 100: KMK Pinout Table*

| **ItsyBitsy Pins** | **A1** | **A3** |
|---|---|---|
| **A2** | Keycode.F13 | Keycode.F14 |
| **A4** | Keycode.F15 | Keycode.F16 |

KMK implementation for rotary encoders is also very straightforward and similar to the CircuitPython setup. We would import the module then define the pins. At this point we set the handler pins to a handler map and based on the matrix that is set, it will do the respective command that is set to it. At the moment KMK does not have support for mouse movement but the firmware should be able to work in tandem with CircuitPython commands if we end up seeing KMK as a fallback. Currently KMK has also developed a new firmware called KMKPython which is a fork of CircuitPython with many of the libraries included are optimized and updated, however this option is out of date and should not be used.

### 5.2.2 Bluetooth Connection

We were very adamant on making the connection be as flexible as possible with the option of a cabled connection as well as a bluetooth capability. Adafruit has a github with their integrated libraries for CircuitPython that we were able to use. They also had a quick guide on how to advertise the bluetooth. Once these were all imported, Bluetooth Low Energy was very simple to implement.

### 5.2.3 Procedure

With initial testing we came about a few problems when prototyping regarding the functionality of the microcontroller and its bluetooth capabilities. Adafruit has their own library for bluetooth connections using Bluetooth Low

Energy. All we needed to implement is their libraries consisting of the radio, advertising commands, and UART services. Once these are imported, we can start the advertisement. The thing with this method is that the documentation is quite thin so much testing and playing around with the settings was done to get the bluetooth to work. The main thing was to initially make sure the bluetooth was working with our testing boards, and this could be done through the Bluefruit LE Connection App, however this makes the testing board act as a central unit. Though we got this connection to work, the testing board being a peripheral itself still needed to be tested. Since we knew the testing board was working with the USB plugged in, we thought that all we had to do was make the board discoverable through the adafruit advertisement library. With this change we were able to make it discoverable on the computer. Although that was part of the answer, based on the HID capabilities we were implementing, there was also an HIDServices() function we had to implement also included in the adafruit libraries. This allows the testing board to appear on the computer as a peripheral for the keyboard and mouse HID capabilities.

The only thing here is that the bootware already installed in the development board we used for testing, we also had to take into account and rely on being able to replicate the same environment on our custom boards.

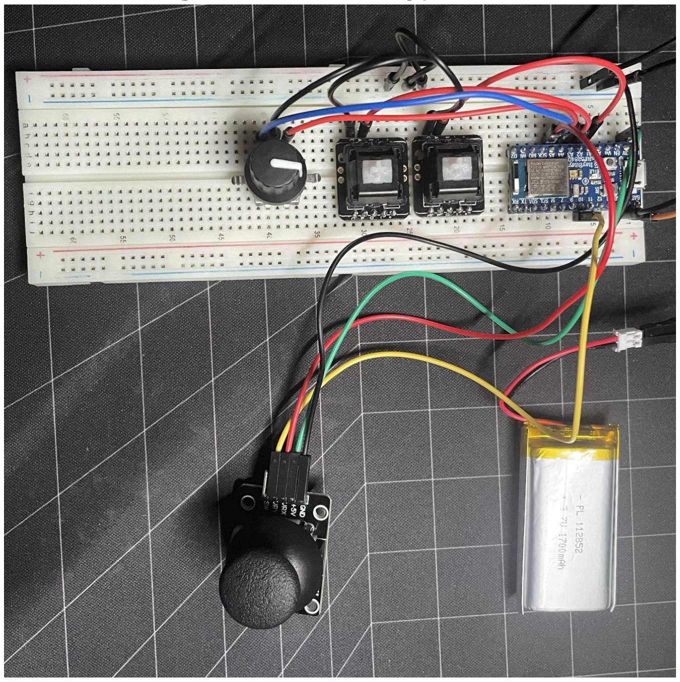
### 5.2.4 Results

To test the results we simply set the keycodes to the pin array to typable letters to make sure the output is working. Using the CircuitPython test code, we were able to edit it to our liking. The first test we conducted was simply through USB connection and having the buttons output the letter we set it too. This test was successful and laid a solid foundation for our bluetooth test. To test the bluetooth implementation, we initially used an external battery and simply did not work which led us to think that we needed the LiPo battery for it to work. With that information, we plugged in a battery to a buck converter then connected it to the board. This led to the buck converter overheating and being dangerous in practice. We ended up plugging the battery straight into the board since the pin was able to take 3.6V to 5.5V. Although this was successful, we ran into a few complications. One thing is that the functionality was not consistent. Initially we were only able to use the buttons for a few seconds before the board would stop working  and came to a few conclusions. One, the battery was not charged, but since the implementation was Bluetooth Low Energy we did not think it would take that much energy. Another conclusion was simply the code not being able to function after a certain time of operating, however in some cases were able to get the battery to work for longer periods of time. One other test we wanted to explore was if the USB connection was also usable with the Bluetooth connection, and that also was successful, but in preparation we had an idea of having a flag that would tell the bluetooth module if it is connected or not then make a decision on what connection to use. Since this was just like any other implementation of bluetooth, the module was also able to connect automatically to devices it was connected to before automatically.

With our success with Bluetooth through using CircuitPython, we hoped to be able to replicate on our custom board, but because of us having to change our environment to Arduino, Bluetooth connection also took a different route. Arduino has many native and community libraries at our disposal but to our surprise, none of them worked to simply connect to our computer or even using the Bluefruit app.

#### 5.2.4.1 Prototype v1.0

*Figure 101: Prototype v1.0*



Our first prototype was combining all the parts we had available including the Kailh NeoKey Sockets, the rotary knob, and the joystick. Due to the limited GPIO/analog pins on the ItsyBitsy, we had to test only some of the hardware, but this was not that big of an obstacle since we were only interested in testing the firmware for the final production of the hardware. The Kailh NeoKey Sockets included pins for LEDs but all we ended up using were the A and C pins located on the breakout board. We connected the respective analog pin to the switch anode or the positive pin on the NeoKey socket board and connected the switch cathode or the negative pin to ground. This allowed us to imitate the basic function of a button that was tested in the initial testing and prototyping of the firmware. Attached above the sockets are Durock POM linear switches that basically shorts the connection to execute an input. The rotary knob was connected to two analog pins that represent channel A and channel B. In the firmware we are able to obtain the number or pulse that is being input, then whatever function we end up setting to it, it will follow. For testing we had the rotation of the position set to increment the volume or decrement the volume, but again this is all testing and will be set to just numbers where they can later be customized for other functionalities in the consumer software.

Figure 102: Digital I/O Inputs on Prototype v1.0

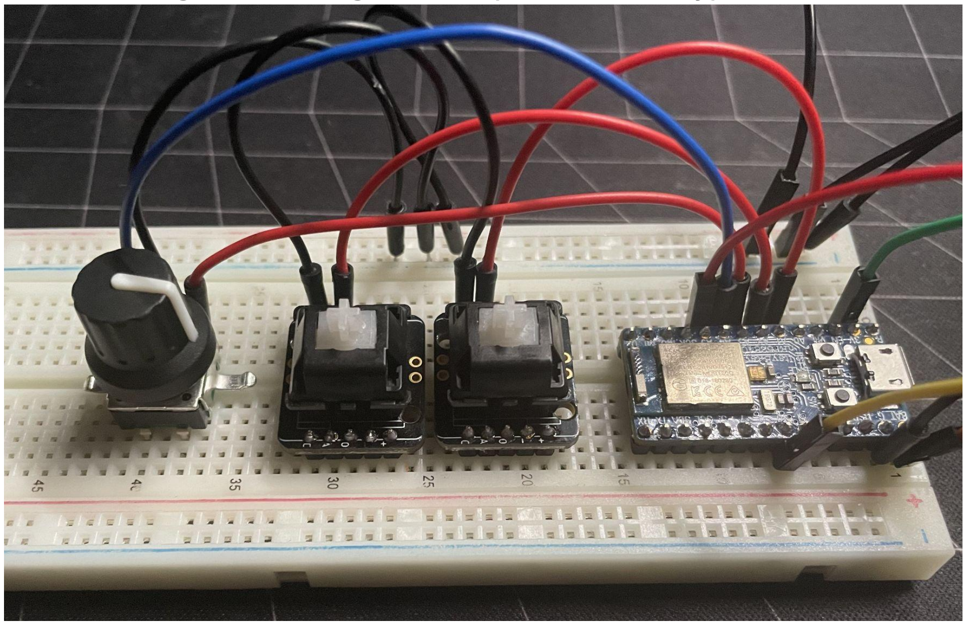
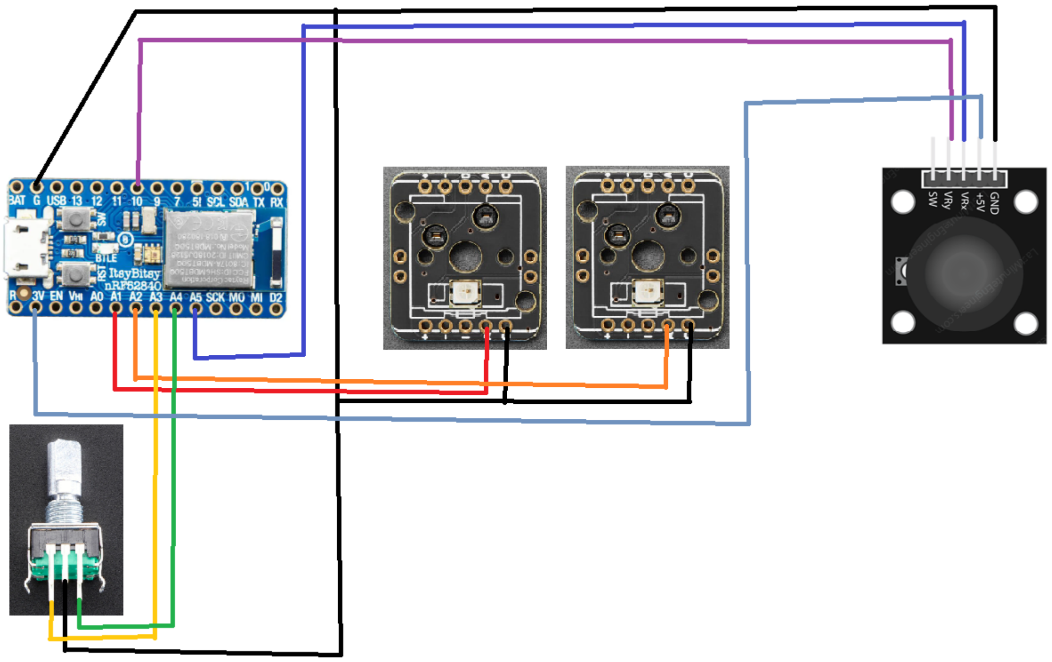


Figure 103: Pin Connection on Prototype v1.0


BAT G USB 13 12 11 10 9 7 5! SCL SDA TX RX
ItsyBitsy nRF52840
R 3V EN VHI A0 A1 A2 A3 A4 A5 SCK MO MI D2
SW VRy VRx +5V GND

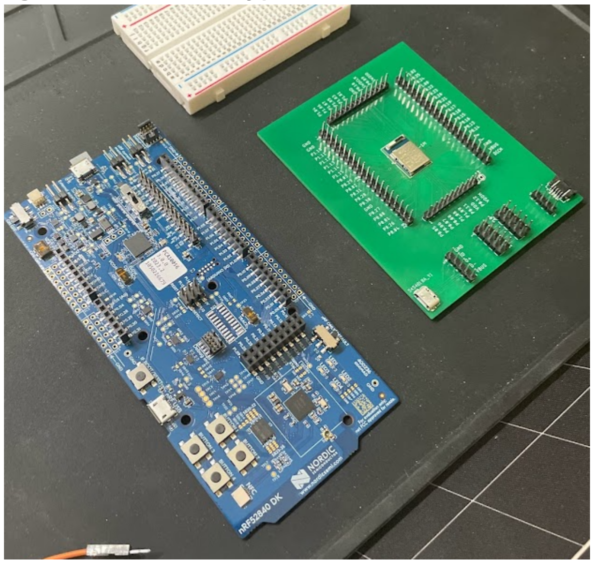
Lastly we needed a way to take in serial X and Y information to have mouse functionalities. The X and Y channels were connected to two analog pins, ground to ground, and the +5V voltage to the 3V output on the ItsyBitsy. Since the joystick also acts as a potentiometer, we had to work around it, but the main feature of taking X and Y inputs was successful.

Although we were able to get this prototype working there were a few things we needed to keep in mind for the following prototypes. As said before, using an SWD programmer to install the necessary firmware to the microcontroller is half the battle, and once we've come across that, the settings for the way the GPIO pins on the microcontroller will interact with the peripherals. Since the development board is recognized as a pre-build adafruit, many of the settings set to the pins and the board are pre-determined, however looking at the documentation and libraries included should aid when developing our own circuitry.

#### 5.2.4.2 Prototype v2.0

Our second prototype was simply replicating the development board we used for testing. We wanted to just be able to talk to the microcontroller using the SWD programmer. Initially, using the third-party J-Link programmer bricked 2 of our testing development board and we had to reevaluate how to install the firmware onto our board. Although there were many programmers that we weren't able to get at an affordable price we eventually found the PCA10056 to program our microcontroller, using the nRF Connect app by Nordic Semiconductors. Using the PCA was successful with programming the microcontroller. This allowed us to install the PCA10056 bootloader into our custom board and use the available pins. In practice the PCA10056 gives us access to all the necessary pins we need right off the bat so fixing the bootloader firmware was not needed. Once we installed the PCA bootloader, we can now access the board through the Arduino IDE and program it through there. The code we made in CircuitPython was easy to migrate to Arduino, the only concerning problem we ran into was using the bluetooth functionalities through Arduino. For this prototype we had to connect the microcontrollers to the necessary resistors and capacitors for further testing then use the available pins to our other testing units to make sure the communication to the microcontroller is good. Based on this prototype and testing through the Arduino IDE, all the units including the macro keys, rotary encoders, and trackpad all worked. Shown in the figure below is the second prototype along with the PCA10056.

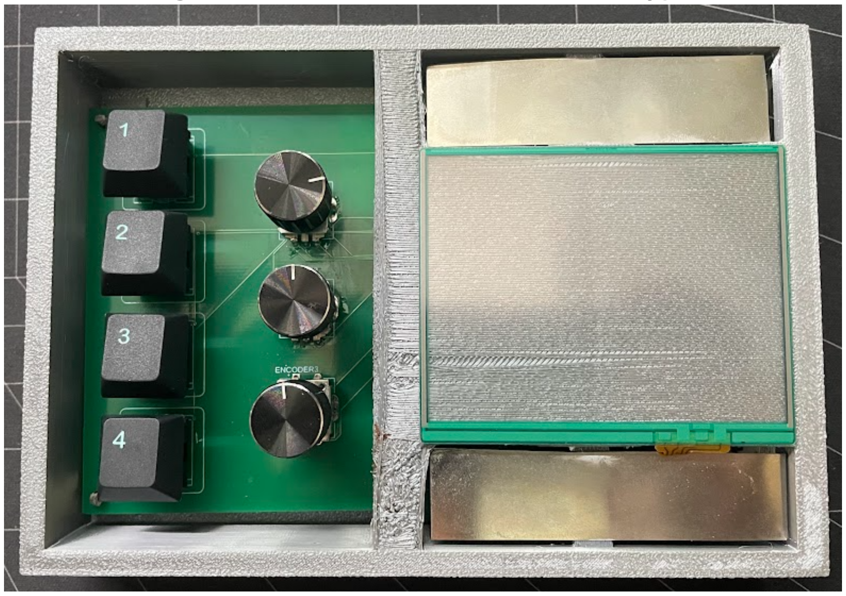
*Figure 104: Prototype PCB with PCA10056*



### 5.2.4.3 Prototype v3.0 (Final PCB)

Our third and final prototype was finalizing everything. In this process of development, we also had to make sure the pins we were using were correct for the respective input unit. The switches, rotary encoders, reset button, mouse left and right clicks, and orientation switch were finally mapped to set GPIO pins, and the trackpad X and Y capabilities were mapped to analog pins to be interpreted further. Because we had prepared the firmware using prototype v2, all we had to do for this version 3 was simply upload the PCA10056 bootloader and upload the firmware using the Arduino IDE. This was also where we added the extra input unit mappings in the firmware and tested all the units working in unison. From our testing we found that the macro switches, rotary encoders, and mouse keys all work from the code in tandem, however when the trackpad comes into the mix along with the orientation switch, we get different results. The trackpad firmware in tandem with the rest of the firmware created a lot of noise within the trackpad where the cursor would stay still or sometimes rather not even work completely, but with the trackpad firmware working alone, we found it worked perfectly. One other thing was the orientation switch. The simple slide switch also wasn't producing the desired outcome which was also affected by the unknown noise from the device. The orientation switch gave us conflicting values when we set it to a flag to set the correct mouse movement based on the orientation, because of this and the conflicting firmware working in tandem we had to show off the trackpad firmware working separately from everything else. However, we were very proud of the device coming all together and working as a whole. Shown in the figure below is the final prototype.

Figure 105: Final Assembled Prototype



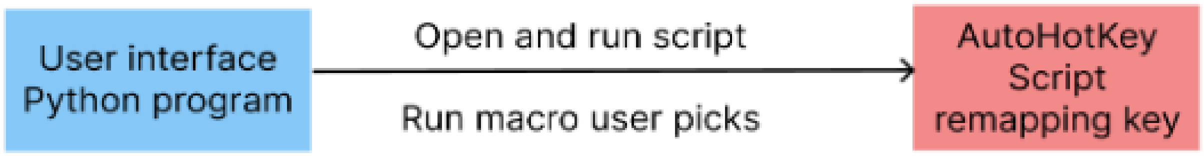
## 5.3 Application Software

The application is where the user will edit the function of the macro-keys. How this app will look and function is still being researched. However, we wanted for it to be a user-friendly GUI where the user can select each macro-key and rotary encoder and map it to a function of their choosing. We mainly used PyGUI for all the design elements and used Python for scripting in conjunction with AutoHotKey.

### 5.3.1 Running AutoHotkey Script Inside a Program

To verify we could change the functionality of a key inside the Windows Operating System, a number of tests were laid to verify that this fundamental feature was operational. After our technology investigation, documentation and other research supported that AutoHotkey could be installed, script files could be created, and then could run through a python program. But since this software feature is fundamental in our overall goals and objectives, testing and prototyping must be thorough to prevent potential problems in the future.

Figure 106: Overall Procedure Layout for Prototype



#### 5.3.1.1 Procedure

In order to test that AutoHotkey can change the functionality of a keypress, the first step is to install all necessary software and/or programming tools. For all text and program editing, VSCode (Version 1.73) and Sublime (V4) was installed. AutoHotkey version 1.1.35 and Python version 3.11.0 were installed. Note all tests and installs were on a Windows 10 Operating System.

*Figure 107: Software Versions Used During Testing and Prototyping*

| Software Required | Version Used |
|---|---|
| Windows OS | 10 |
| VSCode | 1.73 |
| Sublime | V4 |
| Python | 3.11.0 |
| AutoHotKey (AHK) | 1.1.35 |

The next test regards creating AutoHotKey scripts and being able to execute them standalone on the OS. Meaning, the test will be successful if you see the expected outcome after double clicking the AutoHotKey script and/or running it from Command Prompt. For this initial test, a simple 'Hello World' program is sufficient as the expected outcome. Overall, this test helps the procedure workflow become more smooth and efficient where troubleshooting and completing these different milestones won't appear as complex if they are taken one step at a time.

After verification that AutoHotkey can run by itself on the Operating System, the next step is to test whether or not a AutoHotKey script can be created to change the functionality of a keypress standalone. After double clicking the script, observations should be made on the expected outcome where 1) if it is present at all, and 2) if the new functionality remains on the key even after the script is terminated.

The final step in this procedure is to test the ability that a AutoHotKey script can be executed programmatically through a Python program. For this test a simple Python program can be created to quickly open an AutoHotKey file and then execute it. Similar to the previous test, if the AutoHotKey file remaps a key to a different functionality, observation of the result after termination of the Python program should be made to verify if the AHK file maintains the same expected outcome.

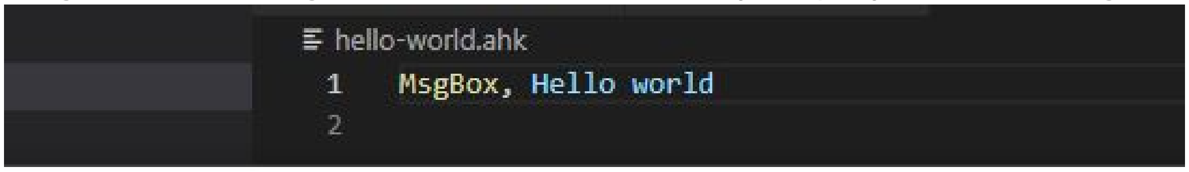
Figure 108: Procedure for Testing AutoHotkey

| Test | Description |
|---|---|
| 1 | Install all necessary software tools |
| 2 | Execute AutoHotkey script through Command Prompt |
| 3 | AutoHotkey script remap a function key's functionality |
| 4 | Execute an AutoHotkey script from Python |

### 5.3.1.2 Results

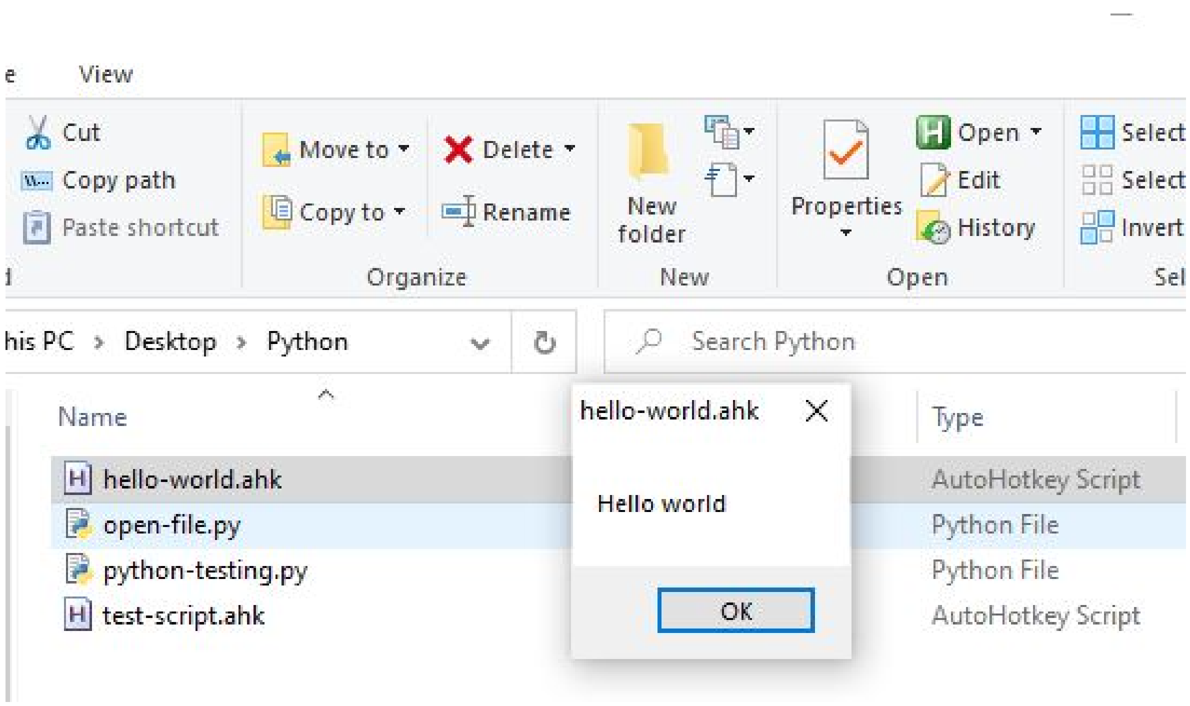
Following the different steps listed in the procedure the first step was setting up the prototyping environment with all of the listed software technologies needed to be installed. The main constraint is ensuring that both Python and AutoHotkey are running and installed properly. Installation was successful as well as running the first 'Hello World' script for AutoHotkey.

Figure 109: Running Initial Hello World AutoHotkey Script by Double Clicking



hello-world.ahk

```
MsgBox, Hello world
```



| Name | Type |
|---|---|
| hello-world.ahk | AutoHotkey Script |
| open-file.py | Python File |
| python-testing.py | Python File |
| test-script.ahk | AutoHotkey Script |

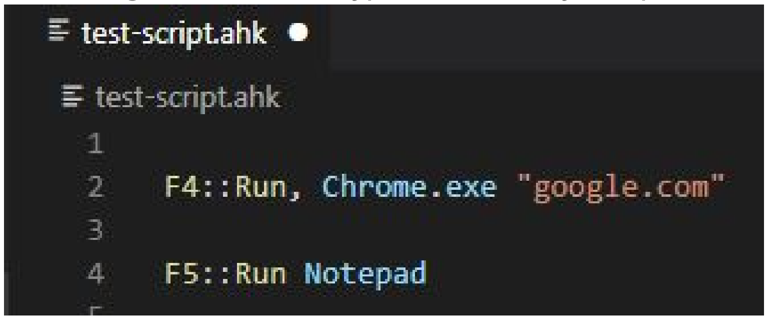
hello-world.ahk

Hello world

OK

After successfully getting the expected results from the figure above, the next step was to test AutoHotkey's ability to remap a function key to a new functionality. More observations were needed before diving into the testing for the integration between Python and AutoHotkey. Using AutoHotkey's remapping functionality found in their documentation, running the following script produced a successful test where there was a change in the functionality of keys F4 and F5.

*Figure 110: Prototype AutoHotkey Script*



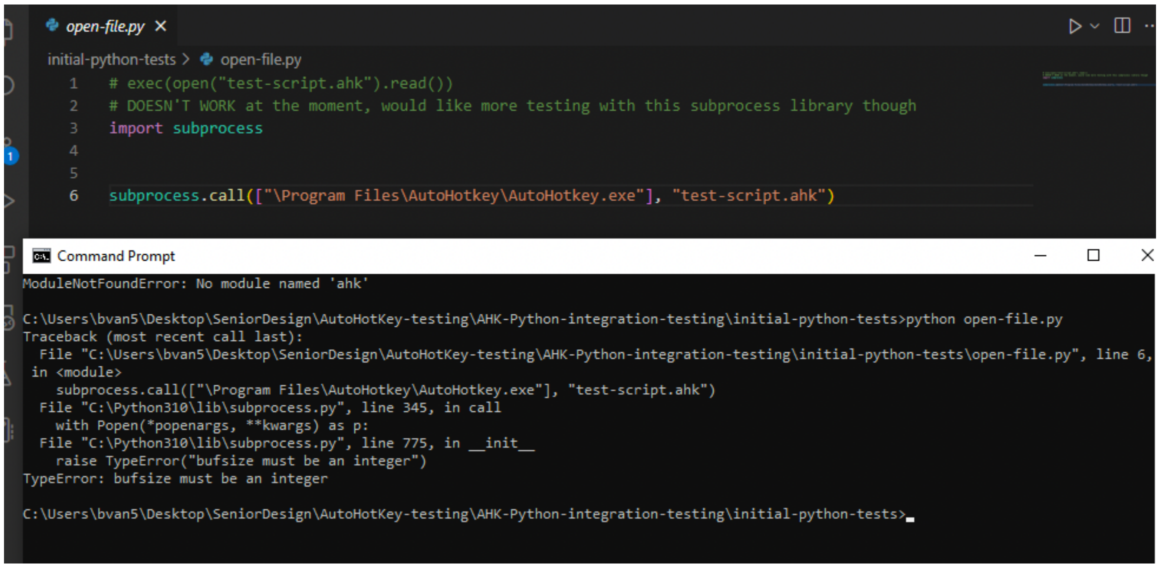
```
test-script.ahk

test-script.ahk

F4::Run, Chrome.exe "google.com"

F5::Run Notepad
```

With this test, as soon as the AutoHotkey script is executed, the remapping functionality would remain as long as the script is running where it would revert back to the default functionality on termination of the script.

For the tests regarding integrating Python and AutoHotkey, the first tests in Python attempting to open and run the AutoHotkey script were not successful. Further investigation is required if we go down the route of having a number of preset AutoHotkey files stored in the data files of our application software (such as the one in the figure above). However, the section port of the test showed results of eliminating the need for an AutoHotkey file itself and inserting all remapping functionality inside the python program. The only constraint is the python program would have to be running at all times which might not be ideal for the current design plans for the application software. The only issue was this is the user would have to keep the application software open at all times just to have macros on their device. This wouldn't be ideal where an AutoHotkey script running in the background without a Graphical User Interface (GUI) could be used instead.

More testing and investigations are required before finalizing the overall software design where during the design and prototyping phase having multiple ways of completing our tasks is fine until it's time to pick the more efficient version. More observations are needed to be made to test the ability to open and run an AutoHotkey script from Python. This way if the Python program was terminated the AutoHotkey script should still be running. However, during this first prototyping phase, there were errors trying to achieve this.

*Figure 111: Error Results attempting to run AutoHotKey.exe in AHK Program Files*



```python
# exec(open("test-script.ahk").read())
# DOESN'T WORK at the moment, would like more testing with this subprocess library though
import subprocess


subprocess.call(["\Program Files\AutoHotkey\AutoHotkey.exe"], "test-script.ahk")
```

```
ModuleNotFoundError: No module named 'ahk'

C:\Users\bvan5\Desktop\SeniorDesign\AutoHotKey-testing\AHK-Python-integration-testing\initial-python-tests>python open-file.py
Traceback (most recent call last):
  File "C:\Users\bvan5\Desktop\SeniorDesign\AutoHotKey-testing\AHK-Python-integration-testing\initial-python-tests\open-file.py", line 6,
 in <module>
    subprocess.call(["\Program Files\AutoHotkey\AutoHotkey.exe"], "test-script.ahk")
  File "C:\Python310\lib\subprocess.py", line 345, in call
    with Popen(*popenargs, **kwargs) as p:
  File "C:\Python310\lib\subprocess.py", line 775, in __init__
    raise TypeError("bufsize must be an integer")
TypeError: bufsize must be an integer

C:\Users\bvan5\Desktop\SeniorDesign\AutoHotKey-testing\AHK-Python-integration-testing\initial-python-tests>
```

## 5.3.2 Running AHK Inside Python Prototype V2

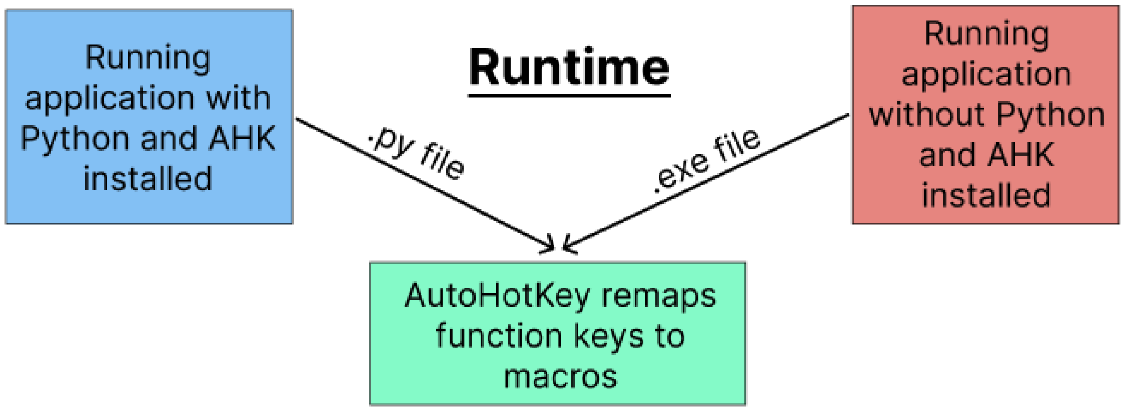
The initial prototyping phase discussed in 5.3.1 focused on installing, running, and remapping keys in AutoHotKey scripts where there wasn't much progress in integrating this into Python. Hence, the overall focus of this section was ensuring that the AHK scripts could be programmatically called upon without interfering with any other functionalities the Python program might be focused on.

The testing for this prototype version will also focus on the aspect of user friendliness in regards to installing our application software for the first time. Some of the goals for this prototype is the ability to run the application software in an operating system that doesn't have Python or AutoHotKey installed. If this can be accomplished, the user won't have to go through a lengthy process to use their Programmable Trackpad for the first time.

### 5.3.2.1 Procedure

The initial steps prior to physical testing involved more research and reading of documentation for both AutoHotKey and Python. The previous section and the AHK documentation showed successful results that AHK scripts can run through a command in Command Prompt instead of double clicking on the file itself. With this being said, the next step is to determine whether or not Python can use Command Prompt to send it commands programmatically. If this could be done, then the application software can have a 'Flash Board' or an 'Update Macros' button for the user to interact with. This would overall help usability and user-friendliness with the software rather than having them manually call these commands themselves.

*Figure 112: Runtime Goals Regarding Installation of Python and AutoHotKey*


Runtime
Running application with Python and AHK installed
.py file
Running application without Python and AHK installed
.exe file
AutoHotKey remaps function keys to macros

Expanding on executing these AHK scripts through Command Prompt, the previous section addressed errors trying to achieve this inside Python. One of the reasons that might be causing this is the fact that the AHK script requires the execution of the default AutoHotKey executable located in the Program Files. This would require the directory having to change for at least one of the files for the location of the script or the general Program Files for AHK in Windows OS. Hence, testing will involve solving this issue whether it’s verifying the correct file paths are being used and/or finding a way to simplify the .ahk script file into something easier for command prompt to run such as its own executable file.

#### 5.3.2.2 Result

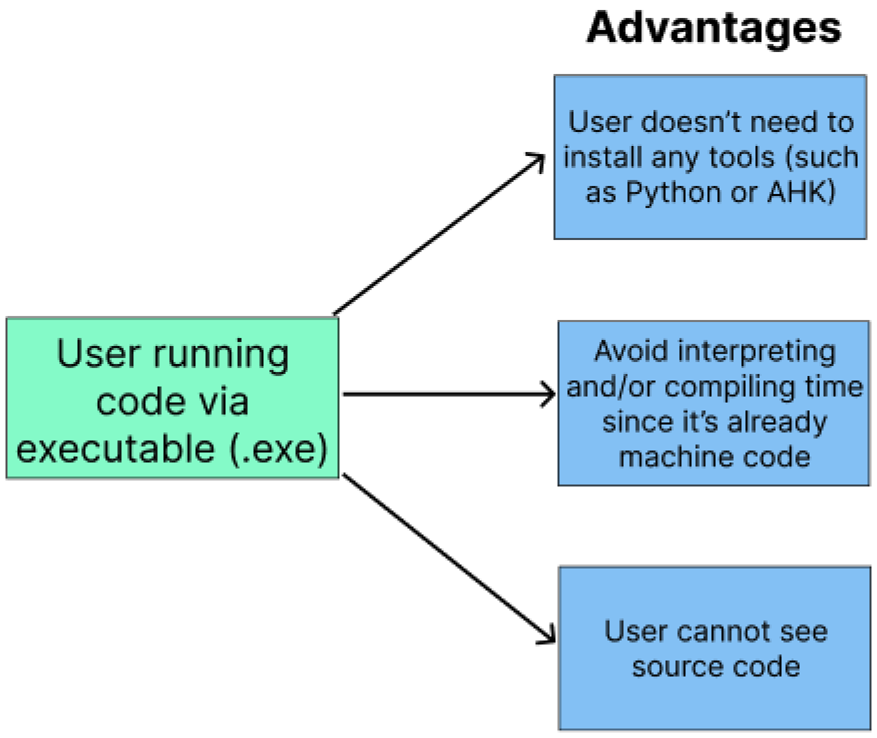
Upon reading the Python documentation for this initial test it was found that Python is able to open and run .exe files through the built in OS library where all it needs is the file name and the path location. With the current setup of running the .ahk file and calling the AHK.exe in Program Files, this can be possible but isn’t the cleanest solution where an extensive file path is required. In effort to find a cleaner solution, further research was looked into in the AutoHotKey documentation. In the Command Line Usage section, it showed how you could compile your .ahk script files into an executable (.exe file). Hence, in this test we converted the AHK script into an executable, setup Python to open & run this executable as if it’s acting as Command Prompt, and then observe the results. We also converted the final .py file into an executable file and recorded the results.

*Figure 113: Test Results converting files into executables*

| **Python running AHK script** | **OS With Python and AHK installed** | **OS with Python installed only** | **Default OS** |
|---|---|---|---|
| .py file running .ahk script | Successful | Unsuccessful | Unsuccessful |
| .py file running .exe script | Successful | Successful | Unsuccessful |
| .exe file running .exe script | Successful | Successful | Successful |

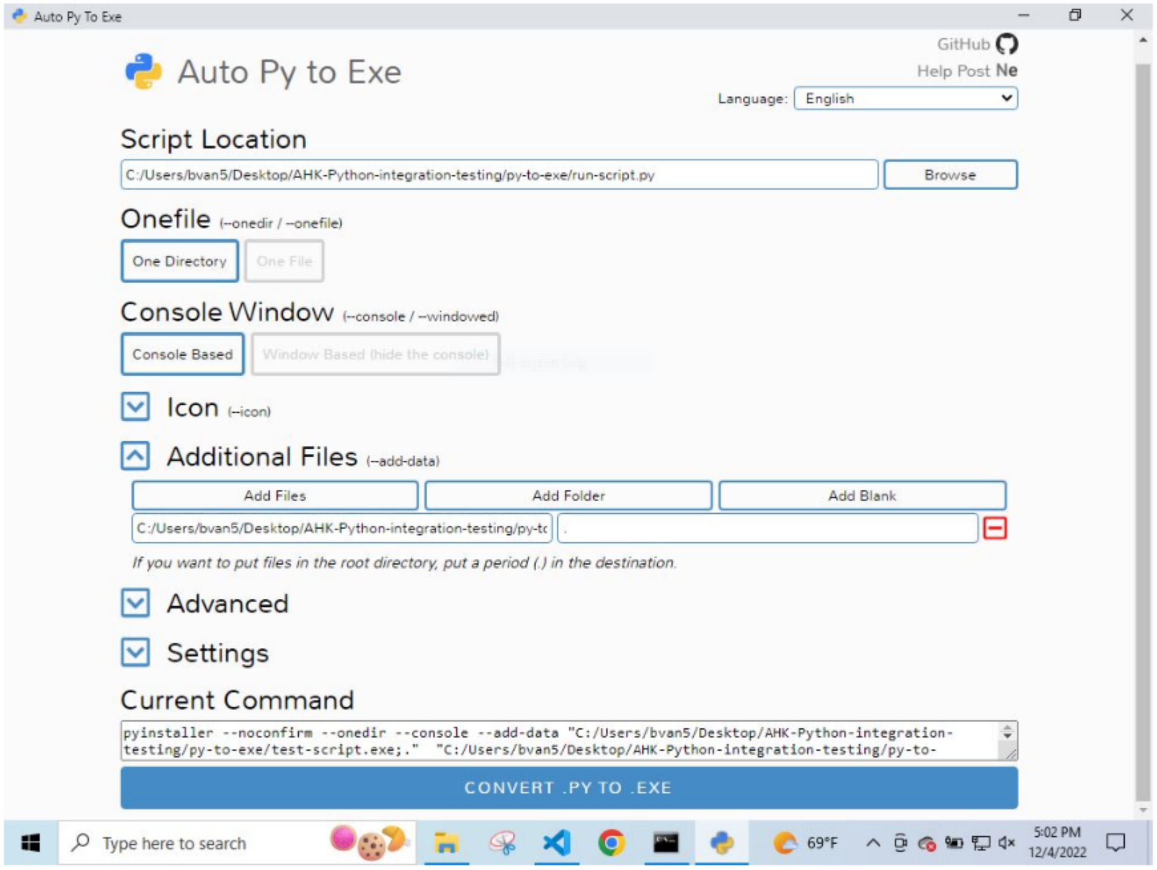
The intended result is the remapping function keys to opening different programs. The main effort to convert these files into an executable is it allows the user to run and use the application without having to install python and autohotkey. AutoHotKey has a built-in compile feature where an .ahk script is not required. Hence, on an Operating System without AHK installed, the intended result is the same as if AHK was installed. This would be ideal in an realistic product market environment where we shouldn't expect the user to install all of these developer tools where they only need the intended result of the software. This is also more secure and helps protect trademark and copyright in the real world where you wouldn't want somebody to see the source code and then potentially copy it and sell it as their own. If this project were expanded upon in a real market environment, these tests and goals of converting the source code into executables is essential.

Figure 114: Advantages Found Converting Python Code into Executable File


Advantages
User running code via executable (.exe)
User doesn't need to install any tools (such as Python or AHK)
Avoid interpreting and/or compiling time since it's already machine code
User cannot see source code

Hence, running .ahk and .py files require the installation to experience successful results where .exe do not require installation. Since executables have translated the code into system commands and instructions for the machine to understand, the last test was done on a system that did not have anything installed except what comes default with Windows 10 (this test was done on a virtual machine). Converting the python code into an executable was done in the open source library Auto Py to Exe that uses PyInstaller. At first, the final executable was failing and showing errors and/or the AutoHotKey script wouldn't run. Upon further investigation and documentation reading, an absolute file path needs to be used rather than a relative path. This was essential for this test to be successful because in order for the AutoHotKey script to be executed, the filepath must be specified. Therefore, a Program Files folder was created to put AutoHotKey compiled scripts inside in effort. During the process of converting the .py file into an executable, these AHK program files need to be specified and added along with the conversion process. After this, a number of files and a directory was generated and of course had the final executable file to test with. The executable ran successfully and did not install any software where it simply ran the AutoHotKey script in the background.

*Figure 115: Configuration Settings to Convert Python Code into Executable - "Auto Py to Exe"*



Language: English

Script Location

C:/Users/bvan5/Desktop/AHK-Python-integration-testing/py-to-exe/run-script.py

Onefile (--onedir / --onefile)

Console Window (--console / --windowed)

Icon (--icon)

Additional Files (--add-data)

C:/Users/bvan5/Desktop/AHK-Python-integration-testing/py-tc | .

*If you want to put files in the root directory, put a period (.) in the destination.*

Advanced

Settings

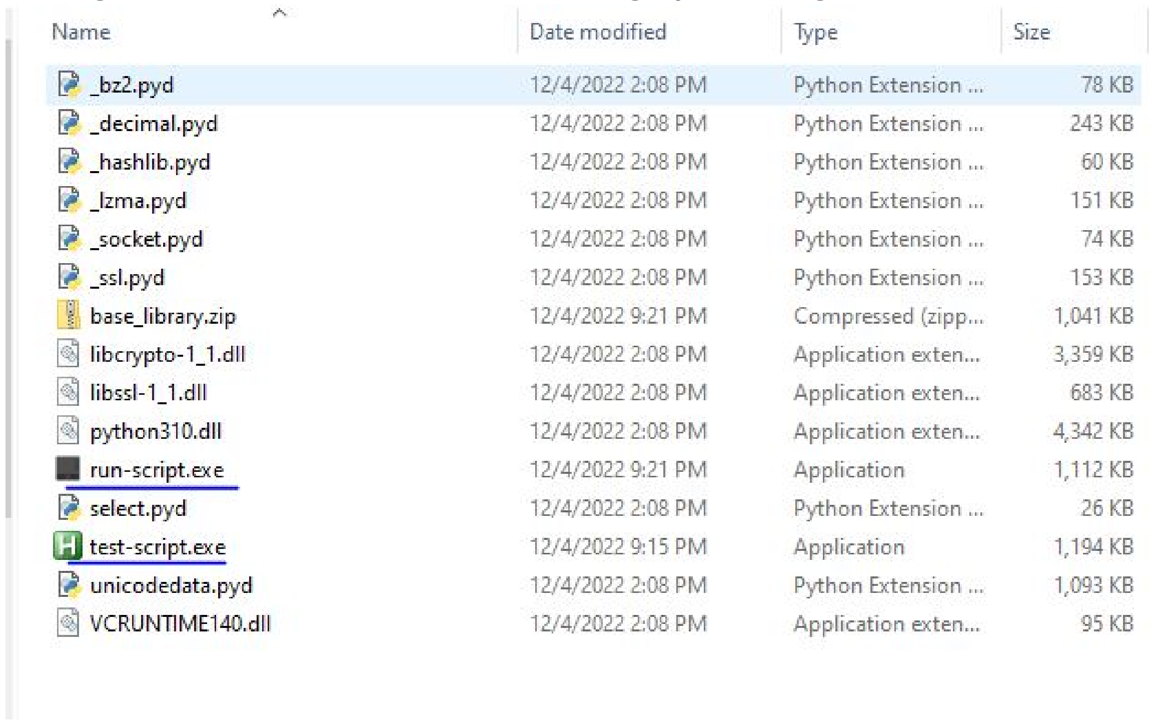
Current Command

```
pyinstaller --noconfirm --onedir --console --add-data "C:/Users/bvan5/Desktop/AHK-Python-integration-
testing/py-to-exe/test-script.exe;."  "C:/Users/bvan5/Desktop/AHK-Python-integration-testing/py-to-
```

69°F 5:02 PM 12/4/2022

It can also be noted that during all of these tests whether it was the .py file or the .exe file executing the prototype application program, if it was terminated the AutoHotKey script would still continue to run unless it was manually stopped. This is beneficial for this project because the entire purpose of this application software is to customize your own macros and upload them to the device. After hitting the button to upload these macros to the device, naturally the user would close the program and then proceed to use the physical device.

*Figure 116: Generated Files - Converting Python Program to Executable*



| Name | Date modified | Type | Size |
|---|---|---|---|
| _bz2.pyd | 12/4/2022 2:08 PM | Python Extension ... | 78 KB |
| _decimal.pyd | 12/4/2022 2:08 PM | Python Extension ... | 243 KB |
| _hashlib.pyd | 12/4/2022 2:08 PM | Python Extension ... | 60 KB |
| _lzma.pyd | 12/4/2022 2:08 PM | Python Extension ... | 151 KB |
| _socket.pyd | 12/4/2022 2:08 PM | Python Extension ... | 74 KB |
| _ssl.pyd | 12/4/2022 2:08 PM | Python Extension ... | 153 KB |
| base_library.zip | 12/4/2022 9:21 PM | Compressed (zipp... | 1,041 KB |
| libcrypto-1_1.dll | 12/4/2022 2:08 PM | Application exten... | 3,359 KB |
| libssl-1_1.dll | 12/4/2022 2:08 PM | Application exten... | 683 KB |
| python310.dll | 12/4/2022 2:08 PM | Application exten... | 4,342 KB |
| run-script.exe | 12/4/2022 9:21 PM | Application | 1,112 KB |
| select.pyd | 12/4/2022 2:08 PM | Python Extension ... | 26 KB |
| test-script.exe | 12/4/2022 9:15 PM | Application | 1,194 KB |
| unicodedata.pyd | 12/4/2022 2:08 PM | Python Extension ... | 1,093 KB |
| VCRUNTIME140.dll | 12/4/2022 2:08 PM | Application exten... | 95 KB |

Overall, for this second phase of early prototyping for the application software, a large majority of goals were achieved in the terms of the code running “behind the curtains” so to speak where the user isn’t necessarily seeing or interacting with directly. Having the ability to run AutoHotkey via Python was an absolutely necessary and a high priority goal where this would act as one of the main functionalities of the application. However, this prototyping phase also achieved an advanced goal of being able to convert all code into executable files where we don’t have to expect the user to download and install the latest version of Python and/or AutoHotKey.

# 6 Administrative Materials

## 6.1 Budget and Costs

This project is not sponsored and is not receiving any other source of outside funding. All funding for this project will be put forth by the 4 members of this team equally. Our goal early in the project’s development was to keep the final amount of money spent under $200. In the end, that budget was overshot, as is shown in the figure below. This table represents a list of parts that we used in the prototyping process separate from the actual components used in the final design.

*Figure 117: Project Development Budget*

| Development Costs | | |
|---|---|---|
| Item | Price | Purpose |
| 3.7" Touchpad | $11.45 | Prototyping touch functions |
| Touchpad to USB breakout board | $17.59 | |
| 500 mAh LiPo battery | $6.99 | Prototyping power system |
| 1700 mAh LiPo battery | $10.99 | |
| 18650 battery charger | $5.99 | |
| USB-C Battery Management board | $8.99 | |
| USB-C Battery Charging board | $12.99 | |
| TPS61023 Development Board | $3.56 | |
| LP3671 Buck Converter | $11.22 | |
| Rotary Encoders | $9.00 | Prototyping HID commands |
| Mech Switch Breakout Board | $3.25 | |
| MDBT50Q | $12.95 | Prototyping microcontroller |
| Itsbitsy nRF52840 Express | $27.20 | |
| SWD Programmer | $6.40 | Prototyping SWD programming |
| nrf52840 DK | $57.18 | |
| ESP32-WROOM-32D | $9.09 | Exploring nRF alternatives |
| PCBs and Stencils | $102.96 | Prototyping system integration |
| | | |
| **Total** | $317.80 | |

The total development cost was $317.80: higher than our initial estimate, but still within an acceptable range when distributed between the four team members. The figure below shows the bill of materials for the final device. This table illustrates the approximate cost of building one Programmable Trackpad.

*Figure 118: Bill of Materials*

| **Bill of Materials** | | |
|---|---|---|
| Item | Quantity | Price |
| MDBT50Q | 1 | $12.95 |
| 500 mAh LiPo battery | 1 | $6.99 |
| LM3671 3.3V Converter | 1 | $1.61 |
| MCP73831 Battery Management System | 1 | $0.77 |
| 3.7" Touchpad | 1 | $11.45 |
| Rotary Encoder | 3 | $2.90 |
| Kailh Socket Pack | 1 | $2.50 |
| Boba U4T Key Switch Pack | 1 | $11.50 |
| PCB | 1 | $2.00 |
| Ribbon Cable Receptacle | 1 | $0.36 |
| Micro USB Receptacle | 1 | $1.36 |
| **Total Cost** | | **$60.19** |

The total cost of the device by a parts breakdown is $60.19. Even without considering bulk discounts or profit margins, this cost would seem quite reasonable in comparison to the comparable devices discussed in our market research.

## 6.2 Milestones

At the beginning of the project, our team set a general timeline specifying our goals for the project with a completion deadline of April 2023. The stages of the development process along the way include Project Definition, Technology Investigation, Parts Acquisition, Prototyping, Design Refinement, and Verification. The following timetables are used as guidelines for the various stages of project completion. Each table represents a new phase of development, each column represents a week of time, and each entry represents the work done during that week.

*Figure 119: Project Timeline*

| **Project Definition** | | | |
|---|---|---|---|
| *Aug. 21st - Aug. 27th* | *Aug. 28th - Sept. 3rd* | *Sept. 4th - Sept. 10th* | *Sept. 11th - Sept. 17th* |
| Form team. | Name project. | Draw initial sketches to explore the project's overall vision. | Create block diagrams to communicate device’s requirements and methods of operation. |
| Initial brainstorming process: explore possible project ideas. | Create a rough outline of the project requirements. | Research existing technology to determine what technologies may be useful to our project. | Research market items similar to our device to inform the design process. |
| | | Finalize project requirements. | |

| Technology Investigation | | | | | |
|---|---|---|---|---|---|
| *Sept. 4th - Sept. 10th* | *Sept. 11th - Sept. 17th* | *Sept. 18th - Sept. 24th* | *Sept. 25th - Oct. 1st* | *Oct. 2nd - Oct. 8th* | *Oct. 9th - Oct. 15th* |
| Research parts that will be necessary for device operation. | Make early decisions on what technologies to use for major components. | Divide responsibilities between group members. | Begin tracking a Bill of Materials based on parts discovered in early research. | Order materials for initial prototyping (pre-built circuit boards). | Finalize project requirements. |
| | | Taylor’s responsibility: Power system. | Flesh out project requirements as we learn about the necessary technology. | Select software programs to use in the final product. | |
| | | Jonah’s responsibility: Bluetooth and USB connectivity. | | | |
| | | Brian’s responsibility: Touchpad hardware. | | | |
| | | Bradley’s responsibility: Application software. | | | |

| **Parts Acquisition** | | | | |
|---|---|---|---|---|
| *Oct. 9th - Oct. 15th* | *Oct. 16th - Oct. 22nd* | *Oct. 23rd - Oct. 29th* | *Oct. 30th - Nov. 5th* | *Nov. 6th - Nov. 12th* |
| Acquire initial prototype materials. | Flesh out budget as the Bill of Materials is fleshed out. | Finalize budget. | | Acquire all parts necessary to prototype every subsystem in the project. |

| **Prototyping Phase 1: Separate Device Testing** | | | |
|---|---|---|---|
| *Oct. 30th - Nov. 5th* | *Nov. 6th - Nov. 12th* | *Nov. 13th - Nov. 19th* | *Nov. 20th - Nov. 26th* |
| Conduct demonstration of USB connection with PC. | Conduct demonstration of battery charging. | Conduct demonstration of macros running through PC software. | Conduct demonstration of application with navigable GUI. |
| Conduct demonstration of touchpad in operation. | Conduct demonstration of sending mouse/keyboard commands to PC. | | |

| **Prototyping Phase 2: Integrated System Testing** | | | | |
|---|---|---|---|---|
| *Jan. 8th - Jan. 14th* | *Jan. 15th - Jan. 21st* | *Jan. 22nd - Jan. 28th* | *Jan. 29th - Feb. 4th* | *Feb. 5th - Feb. 11th* |
| Conduct demonstration of MCU connecting to PC with Bluetooth and USB. | Conduct demonstration of MCU functioning as a mouse/keyboard. | Conduct demonstration of MCU being powered by a charging battery. | Conduct demonstration of software application running macros from user input. | Conduct demonstration of application software running macros from MCU input. |

| Refine Design | | | | |
|---|---|---|---|---|
| *Jan. 29th - Feb. 4th* | *Feb. 5th - Feb. 11th* | *Feb. 12th - Feb. 18th* | *Feb. 19th - Feb. 25th* | *Feb. 26th - Mar. 4th* |
| Complete full electrical schematic of device. | Identify any parts that need to be changed based on prototyping data. | Prototype Testing with custom development board | Order PCB and associated components. | Complete chassis design to house PCB. |
| Begin drafting a PCB layout for the device. | Order prototyping boards for any new parts that have been added to the design. | Finalize advanced features of Application Software | Finalize firmware for final PCB | |

| Prototyping Phase 3: Final Assembly | | | | |
|---|---|---|---|---|
| *Feb. 5th - Feb. 11th* | *Feb. 12th - Feb. 18th* | *Feb. 19th - Feb. 25th* | *Feb. 26th - Mar. 4th* | *Mar. 5th - Mar. 11th* |
| Conduct demonstratio n of all hardware components working together. | Conduct demonstratio ns with new parts ordered to replace old systems. | Continue working with the prototypes from previous weeks as needed. | Continue last second testing | Solder Final PCB |
| | | | | Check PCB for electrical faults. |

| **Verification** | | | |
|---|---|---|---|
| *Mar. 12th - Mar. 18th* | *Mar. 19th - Mar. 25th* | *Mar. 26th - Apr. 1st* | *Apr. 2nd - Apr. 8th* |
| Test firmware uploading to device. | Test device in ordinary use cases. | Stress test device. | Correct problems as needed. |

| **Final Preparation** | |
|---|---|
| *April. 9th - April. 15th* | *April. 16th - April. 18th* |
| Correct problems as needed | Presentation and Demo Preparation |

# 7 Conclusion

In conclusion, this document provides a thorough and in-depth look at our thought process in the design phase of the Programmable Trackpad. It contains detailed metrics about what niche our device aims to fill, why we feel it's necessary and all of the steps we took to make this device a reality.

We believe that there is a hole in the computer peripheral market that is a programmable trackpad. With it, users would be able to map a series of macro-keys and rotary encoders to perform unique custom actions. We believe this will serve as a great convenience for those who prefer a trackpad over a mouse, and will lead to increased productivity for its user.

Once we laid out the problem and its solution, we began discussing all of our goals/objectives for the Programmable Trackpad, and the engineering requirements that we will base our design choices on. These requirements constituted core functions of the Programmable Trackpad such as the inclusion of macro-keys, rotary encoders, wireless functionality, ambidextrous capabilities, etc. Once we had a solid understanding of these requirements, we began research on individual components.

Using the engineering requirements, we conducted intense research into many different electrical hardware components as well as different software choices. We compared the pros and cons of each component to determine which would be the best fit for our requirements. For example, we looked at a few different choices for touchpads. Each one was vastly different from one another and had their own unique features, but we were ultimately able to choose a single one citing that the features it had were better suited for our device.

Upon choosing a list of part numbers, we were able to create a more concrete design plan. We created block diagrams for all of the major interworking systems in the Programmable Trackpad and visualized how they will work together. We also went more in-depth as to how these systems will work, and how each part that we chose in the technology research section will work as a part of a larger system.

After making a decision on components, software and hammering down the design details, we were able to start prototyping these systems. Prototypes included crude circuits of different components and early builds of code bases. For the software side of things, coming up with a set design beforehand is extremely important in this stage of the project. Whether it’s determining what software tools are being used, pinpointing the main functionality of the different software components, and having an overall frontend design before actual coding begins.

Finally, we drew up a bill of materials as well as a timeline. The bill of materials contains an estimation of what we spent on the entire project.  Similarly, the timeline provides a rough estimate as to when we reached certain milestones in the Programmable Trackpad’s development. Overall, this document should allow someone to replicate this project very closely if they read it carefully.

# 8 Bibliography

[1] “MX Master 3S Wireless Performance Mouse.” Logitech, https://www.logitech.com/en-us/products/mice/mx-master-3s.910-006556.html

[2] “Wired Ergonomic Gaming Mouse-Razer™ Deathadder v2.” *Razer*, https://www.razer.com/gaming-mice/razer-deathadder-v2

[3] “Microsoft Surface Precision Mouse.” Microsoft Store, https://www.microsoft.com/en-us/d/surface-precision-mouse/8qc5p0d8ddjt?activetab=pivot%3Aoverviewtab

[4] “Magic Trackpad - White Multi-Touch Surface.” *Apple*, https://apple.co/3daX9E9

[5] “Mousetrapper Advance 2.0: Pain in Neck? Try a Centered Mouse.” Mousetrapper, 30 Aug. 2022, https://us.mousetrapper.com/product/mousetrapper-advance-2-0/

[6] “Keymecher Mano-703ub Wireless Trackpad.” *Keymecher*, https://www.keymecher.com/product-page/keymecher-mano-703ub-wireless-trackpad

[7] “Li-Ion & LiPoly Batteries.” Adafruit, https://learn.adafruit.com/li-ion-and-lipoly-batteries

[8] “Datasheet.” TP4056, http://www.tp4056.com/datasheet/

[9] “MCP73831.” Microchip, https://www.microchip.com/en-us/product/MCP73831

[10] “TPS62203.” Texas Instruments, https://www.ti.com/product/TPS62203

[11] “AP63203.” Diodes, https://www.diodes.com/part/view/AP63203/

[12] “LM3671.” Texas Instruments, https://www.ti.com/product/LM3671

[13] Industries, Adafruit. “LM3671 3.3V Buck Converter Breakout - 3.3V Output 600 mA Max.” *Adafruit Industries Blog RSS*, https://www.adafruit.com/product/2745

[14] “24.27US $: RGB TFT Touch LCD Display Module Driver SPI 320*240 480*320 3.3V/5V IPS with SD Card Socket ILI9341 ILI9488 ST7796S Driver - LCD Modules - Aliexpress.” *Aliexpress.com*, https://tinyurl.com/ycxxnu7y.

[15] “Capacitive Touch Kit for Arduino.” DFRobot, https://www.dfrobot.com/product-463.html.

[16] Industries, Adafruit. “Resistive Touch Screen - 3.7’ Diagonal.” *Adafruit Industries Blog RSS*, https://www.adafruit.com/product/333.

[17] “AR1100-I/So.” JLCPCB,
https://jlcpcb.com/partdetail/MicrochipTech-AR1100_ISO/C220640

[18] Industries, Adafruit. “Resistive Touch Screen Controller - STMPE610.” Adafruit Industries Blog RSS, https://www.adafruit.com/product/1571

[19] “USB Type-C Chargers VS Micro USB Chargers.” Haredata Electronics,
https://www.haredataelectronics.co.uk/usb-type-c-chargers-vs-micro-usb-chargers

[20] “GPIO Ports and Registers in AVR ATmega16/ATmega32.” ElectronicWings,
https://www.electronicwings.com/avr-atmega/atmega1632-gpio-ports-and-registers

[21] “Infocenter.” Nordic Semiconductor,
https://infocenter.nordicsemi.com/index.jsp

[22] “ESP32.” Espressif, https://www.espressif.com/en/products/socs/esp32

[23] Raytac. “MDBT50Q-1MV2.” Raytac,
https://www.raytac.com/product/ins.php?index_id=24

[24] Raytac. “MDBT40-256RV3.” Raytac,
https://www.raytac.com/product/ins.php?index_id=74

[25] Raytac. “MDBT42Q-512KV2.” Raytac,
https://www.raytac.com/product/ins.php?index_id=31

[26] “Seeed Studio XIAO SAMD21(Seeeduino XIAO) - Arduino Microcontroller - SAMD21 Cortex M0+ with Free Course.” Seeed Studio,
https://www.seeedstudio.com/Seeeduino-XIAO-Arduino-Microcontroller-SAMD21-Cortex-M0+-p-4426.html

[27] “HID Usage Tables 1.3.” USB,
https://www.usb.org/document-library/hid-usage-tables-13