THE UNIVERSITY of TULSA

Electrical and Computer Engineering

USB-Enabled 4x3 Capacitive Touch Keypad

Final Project Design Report

Adam Dyer

December 13, 2021

Revision History

Date	Version	Author	Description
November 22, 2021	1.0	Adam Dyer	Created File
November 23, 2021	1.1	Adam Dyer	Added Basic Information
December 11, 2021	1.2	Adam Dyer	Created Graphics
December 12, 2021	1.3	Adam Dyer	Wrote Majority of Report
December 13, 2021	1.4	Adam Dyer	Finalized Report

Customer Approval

Name	**Role**	**Signature of Approval**	**Approval Date**
Adam Dyer	Lead Engineer	*Adam Dyer*	December 13, 2021
Nathan Hutchins	Customer	*Nathan Hutchins*	December 14, 2021

Contents

List of Figures

Listings

Abstract

This report will go over the hardware and software design of the project to create a USB-Enabled 4x3 Capacitive Touch Keypad. This device is a keypad without any physical buttons on it. Instead, the "buttons" are in the form of capacitive touch sensors hidden under the top face of the keypad. Touching the face of the keypad above one of these sensor acts as if you had pressed a physical button located there. This results in a very sleek and novel device that has no obvious buttons on it yet functions exactly as if it did. There are also LED backlights hidden under the top face of the keypad that illuminate when the "button" is pressed, further enhancing its appearance and aesthetic and also functioning as an indicator that the button press was detected. A pair of Arduino Nano's control the capacitive sensors themselves and relay that information to a NUCLEO-STM32F411RE Microcontroller Development Board. It takes the information from the Arduino's and turns them into keystrokes corresponding to which button on the keypad was pressed. When plugged into a computer, the Nucleo presents itself as a keyboard and is able to be detected as a keyboard by the Computer. This allows the keys "pressed" on the keypad to show up on a computer screen as if you had typed them yourself on a regular keyboard.

Nomenclature

Nucleo	The NUCLEO-STM32F411RE Development Board
Arduino's	Referring to the pair of Arduino Nano's
LED	Light Emitting Diode
USB	Universal Serial Bus
SPI	Serial Peripheral Interface Communication Protocol
HID	Human Interface Device

Chapter 1

Introduction

1.1 Introduction

The keypad has quickly become a staple of the modern work life. Especially when doing computer work involving a lot of numbers, the keypad makes it easy and convenient to input numbers and common symbols into a computer with only one hand. With enough practice, using a keypad can become much faster than using the standard number row above the letter keys on a keyboard. While some computer keyboards have a number keypad built in to the right of the regular keyboard, quite often laptops and smaller keyboards do not have this functionality.

Additionally, regular keypads have some limitations to them. For one, they are difficult to clean without damaging the electronic button connections inside the actual physical buttons of the keypad. They can also be loud, since the physical buttons can produce audible noise when pressed. They also require a certain amount of force to press down the buttons, which can lead to fatigue of the hand when using a keypad for a long period of time. My project aims to address some of these pitfalls of the standard keypad while also creating a unique, innovative, and aesthetically pleasing device. The main feature of my project will be the use of capacitive touch sensors instead of physical buttons on the keypad.

Capacitive sensing is a technology that can detect and measure anything that is conductive. Capacitive sensing technology is what makes modern day touchscreens on devices like smartphones, tablets, and e-readers possible. While these types of capacitive sensor touchscreens are quite complicated, the basic principle of a capacitive touch sensor is relatively straightforward. Imagine a piece of metal foil that is connected to a pin of a microcontroller. Because the human body is conductive, when you touch your finger to this piece of metal foil it will change it's capacitance. Detecting this change in capacitance using the microcontroller allows that piece of metal foil to function as a simple push button.

However, you do not necessarily need to make physical contact with the metal foil. With a sensitive enough setup, you are able to detect the presence of a finger above the metal foil without actually having the finger physically touch the foil. The exact range of detection varies depending on many factors and can range from needing to make physical contact with the sensor to being able to detect a finger from over a foot away. This ability to detect a finger without having the finger make physical contact with the sensor is the basic working principle behind this project.

Another technology that will be implemented in this project is USB On-The-Go. This is a communication protocol that allows devices to communicate with each other over USB and function as both a host and a peripheral. The NUCLEO-STM32F411RE Development Board is USB On-The-Go enabled. This means that it can act both as a host to receive programming instructions from a computer and as a peripheral to send information back to the computer.

1.2 Objective

The objective of this project is to combine the capacitive sensing technology and USB-On-The-Go features to create a capacitive touch keypad that will be able to connect to any computer as a keyboard. This

means that the capacitive touch keypad will appear to any computer as a connected keyboard, able to send keystroke information to the computer and having those characters appear on the screen as if you typed them yourself.

1.3 Customer

Dr. Nathan Hutchins is the intended customer of this project. He is a Professor of Electrical and Computer Engineering at the University of Tulsa, and the instructor of ECE-2263 Embedded Systems in C. This project is designed to fulfill the requirements of the Final Project for ECE-2263. This project will be assessed and graded by Dr. Hutchins and other faculty members.

1.4 Design Team

1.4.1 Engineering Manager/Lead Engineer

Adam Dyer is the Engineering Manager and Lead Engineer on this project. He is currently a first-year undergraduate student at the University of Tulsa majoring in Computer Engineering. He is the only team member and thus was responsible for the entire project, including hardware and software design and construction, assembly, testing, documentation, and presentation.

Chapter 2

Project Requirements

The following is the Project Requirements as defined in the Requirements Report submitted November 18, 2021. They have not been modified in any way to what what on the original requirements document to ensure the product meets the intended specifications.

2.1 Hardware Requirements

2.1.1 Keypad Layout

The keypad will be designed as a standard 4x3 keypad, meaning it will have four rows and three columns of buttons. See Figure 2.1 for the layout of the buttons. Each button will have a corresponding touch sensor associated with it.

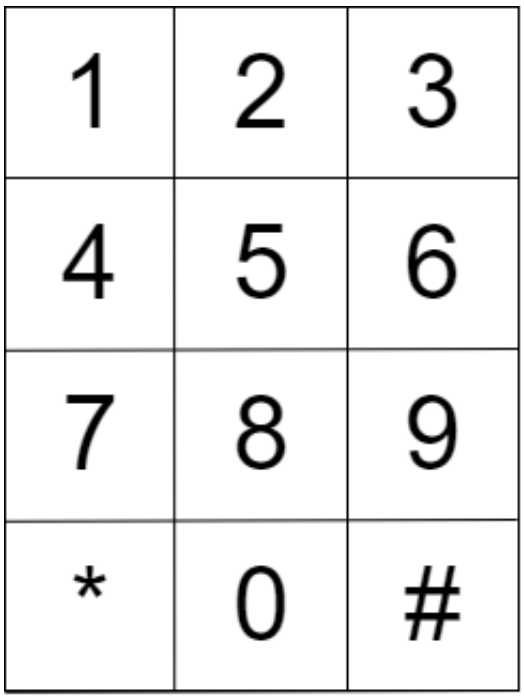

Figure 2.1: Keypad Layout

2.1.2 Enclosure

A key part of what makes this project interesting is the futuristic, sci-fi nature of capacitive touch sensors, since you are essentially pressing a button without actually pressing a button. As such, the overall design of the enclosure will need to be simple and sleek. The keypad itself will be encased in a white 3D-printed enclosure with some type of marking on the top face to indicate the placement of the buttons as shown in Figure 2.1. The top face will be a smooth, solid piece of plastic. The actual metal touch sensors that will detect the button "press" will be located inside this enclosure and will not be visible on the outside. Essentially, the enclosure will look like a standard keypad without having any of the actual, physical buttons present on it.

2.1.3 LED Backlights

Each button will have a small LED mounted to it. This LED will be under the top face of the enclosure, directly above the metal touch sensors. Because the enclosure will be 3D-printed, the light from these LED's will be able to penetrate this top face. The purpose of these LED's is to act as indicators for when a button is pressed. When a button is pressed, the LED directly under it will turn on and illuminate that key of the keypad. This will also enhance the futuristic and sleek appearance of the keypad as described in the first part of section 2.1.2. It should be noted that these LED backlights are purely for aesthetic and ease of use, and will have no impact on the actual functionality of the device.

2.1.4 Microcontollers

Arduino Nano

Do to time constraints and the complexity of capacitive sensing, a pair of Arduino Nano's will act as an intermediary between the keypad and the Nucleo. They will only be responsible for controlling the touch sensors under each button and relying that information to the Nucleo. A pair of Arduino Nano's is used because each touch sensor will require a dedicated analog pin, and two Arduino's are needed to ensure enough analog pins are available.

Nucleo

The NUCLEO-STM32F411RE Development Board will act as the main microcontroller. It will receive the information about the button presses from the Arduino Nano's and relay that information to the connected computer using its built in USB-On-The-Go capabilities.

2.2 Software Requirements

2.2.1 Capacitive Sensing

The software to control and detect the capacitive touch buttons will be written in Arduino C and hosted on the pair of Arduino Nano's. It should be able to reliably detect the presence of a human finger above each button on the keypad. It needs to be sensitive enough to detect the capacitance of a human finger without it actually touching the metal touch sensors (since those sensors will be hidden under the top face of the enclosure) but robust enough to distinguish each individual button. Basically, it should detect a finger when it is directly above the button, but not detect the finger if it is over an adjacent button. The software also needs to be fast enough to handle multiple buttons being pressed in quick succession.

2.2.2 Communication between Arduino's and the Nucleo

The Arduino's must be able to communicate with the Nucleo. This will be implemented using either SPI or I^2C, and the final determination of which protocol to use will be determined later in the development period depending on which protocol presents itself as the better option. The decision on which protocol to use should have no effect on the final functionality of the device.

2.2.3 Nucleo and USB Communication

The software on the Nucleo must be able to receive the SPI or I^2C signals coming from the Arduino's, interpret the signal to determine which button on the keypad was pressed, and relay that information to the Computer using the USB-On-The-Go protocol. Along with this, the Nucleo should present itself to the connected computer as a keyboard so that the computer knows to interpret the information from the Nucleo as keyboard inputs and print the corresponding characters on the computer screen.

Chapter 3

Hardware Design

3.1 Mechanical Design

The mechanical design of this keypad is fairly complex. It consists of four main components: The bottom enclosure, grounding plane, sensor brackets, and the top enclosure.

3.1.1 Bottom Enclosure

The bottom enclosure is a single 3D printed part made from white PLA plastic. A render of the 3D model is shown in Figure 3.1 Its main purpose is to provide the base and specific locations for the sensor brackets to sit on. The purpose of the dotted line protrusions on the front is to provide a reference for the edge of each sensor bracket to ensure the sensor brackets were placed in the correct position.

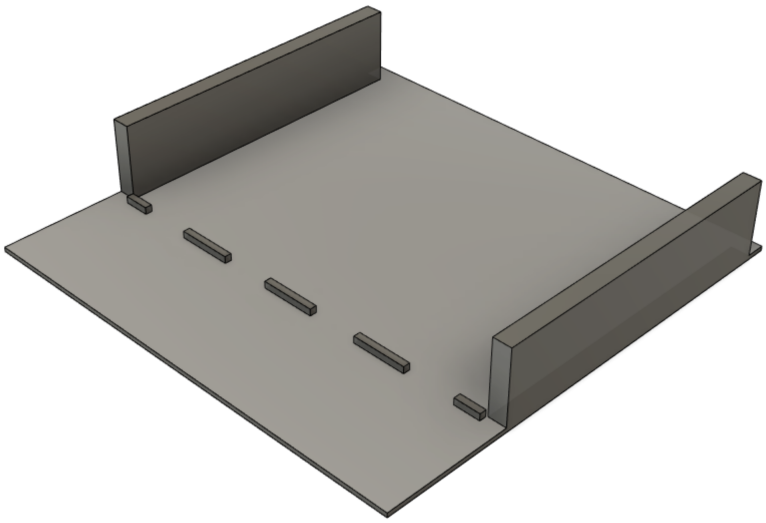

Figure 3.1: Render of Bottom Enclosure

3.1.2 Grounding Plane

Attached to the top of the bottom enclosure is a foil grounding plane. It consists of a large piece of aluminum foil underneath the sensor brackets with four copper foil strips extending out past the dotted line protrusions (Figure 3.2). it main purpose is to provide a solid ground reference to the capacitive sensors. Because the

grounding of the sensors is so important, the ground from the USB cable connected to the computer does not provide a stable enough reference. The ground plane under the sensors does provide a stable ground reference which dramatically increases the sensitivity and consistency of each sensor. The grounding plane was attached using electrical tape, ensuring that the electrical tape is under the parts of the keypad that could short out if they touched the foil. The purpose of the copper strips is to provide an easy place to solder wire to connect the grounding plane to the Arduino's.

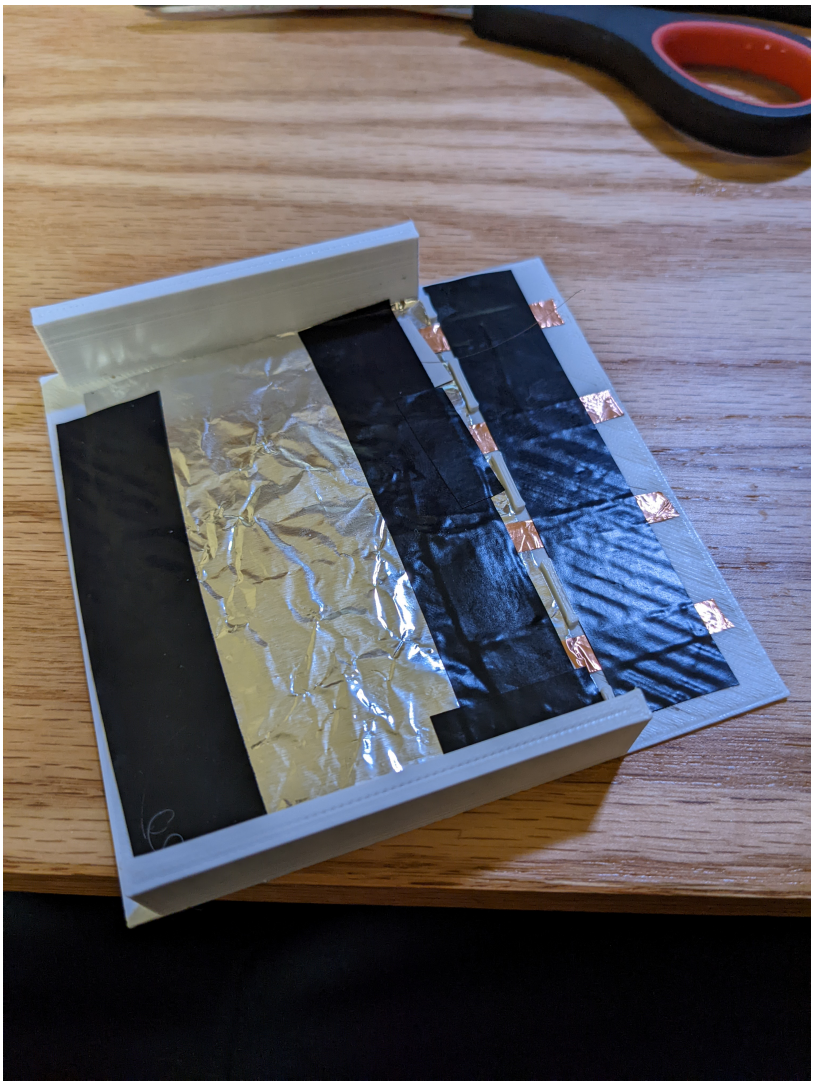

Figure 3.2: Grounding Plane attached to Bottom Enclosure

3.1.3 Sensor Brackets

For ease of assembly and programming, the final sensor bracket assembly actually contains a pair of sensor brackets with six sensors each to make up the 12 sensors total. Each bracket is connected to one Arduino Nano, and the two brackets are identical. The main purpose of the sensor brackets is to elevate the copper foil sensors above the rest of the assembly. Because capacitive sensors measure the capacitance of any metal that touches them, even the wires that connect the sensor to the Arduino's become sensors themselves. As such, it was important to elevate the sensors above the wires so that any interference from the wires was negated due to the much greater distance between your fingers and the connecting wires than between your fingers and the sensors themselves. This also had the added benefit of providing a convenient location to

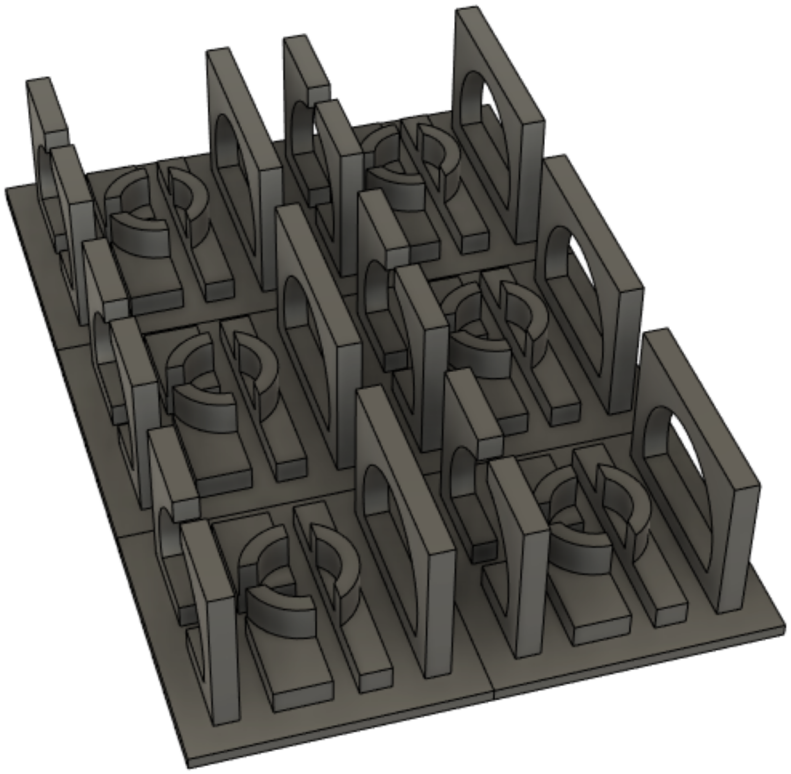

Figure 3.3: Render of One of the Sensor Brackets

mount the LED backlights directly under the sensors. Figure 3.3 is a render of one of the two sensor brackets. The circular mounting hole in the middle of each sensor is where the LED backlights are installed.

Figure 3.4 shows how the LED backlights were installed under each sensor. The anode of each of the six LED's were connected together with the cathode of each LED connected to individual digital pins on the Arduino to allow each LED to be controlled individually.

Figure 3.5 shows the final assembly of one of the sensor brackets. The copper foil sensors each have a wire connected to the bottom of them that is connected to one analog pin of the Arduino. Figure 3.6 shows how the pair of sensor brackets is mounted in the bottom enclosure above the ground plane.

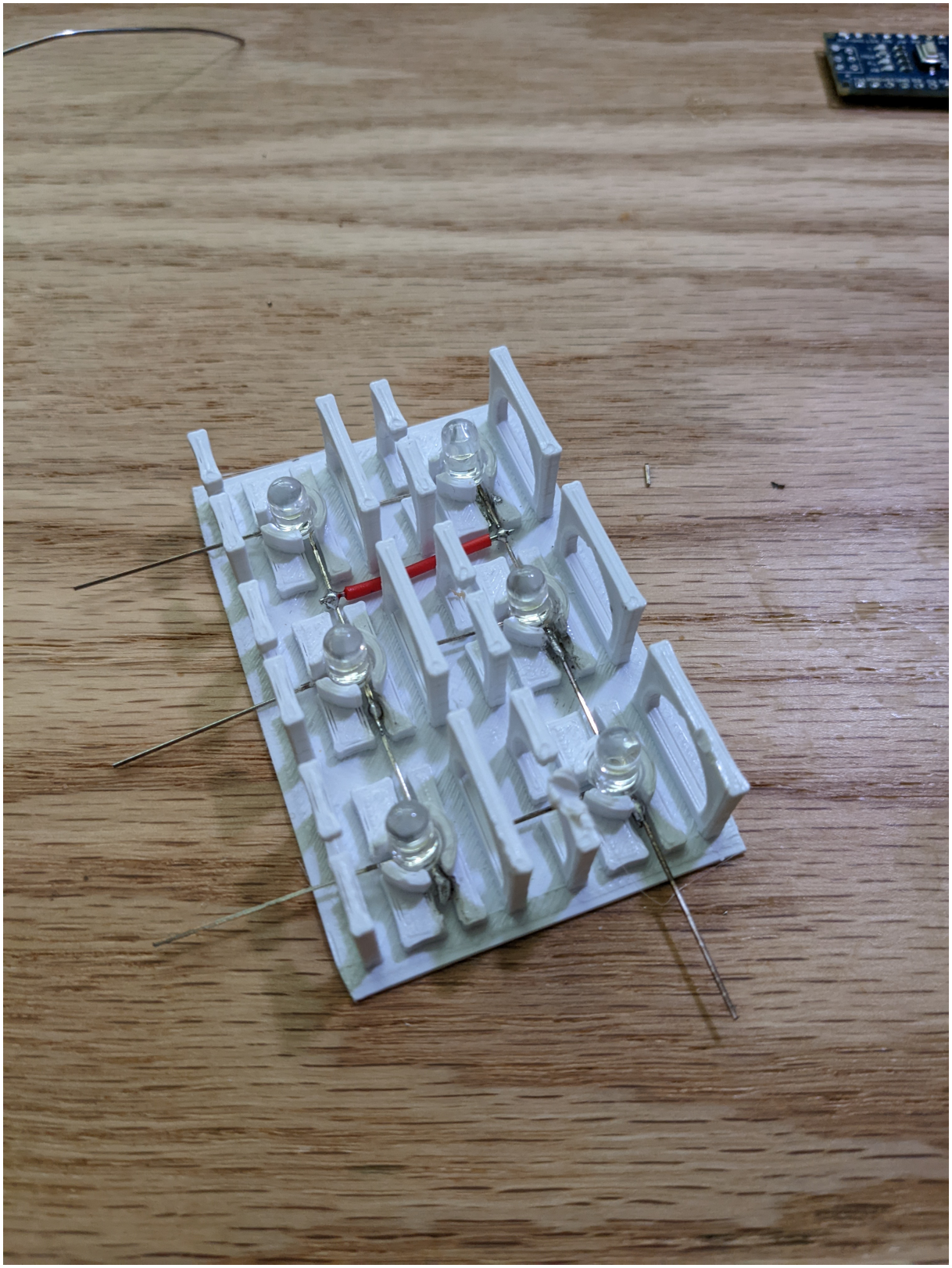

Figure 3.4: Sensor Bracket with LED Backlights Installed

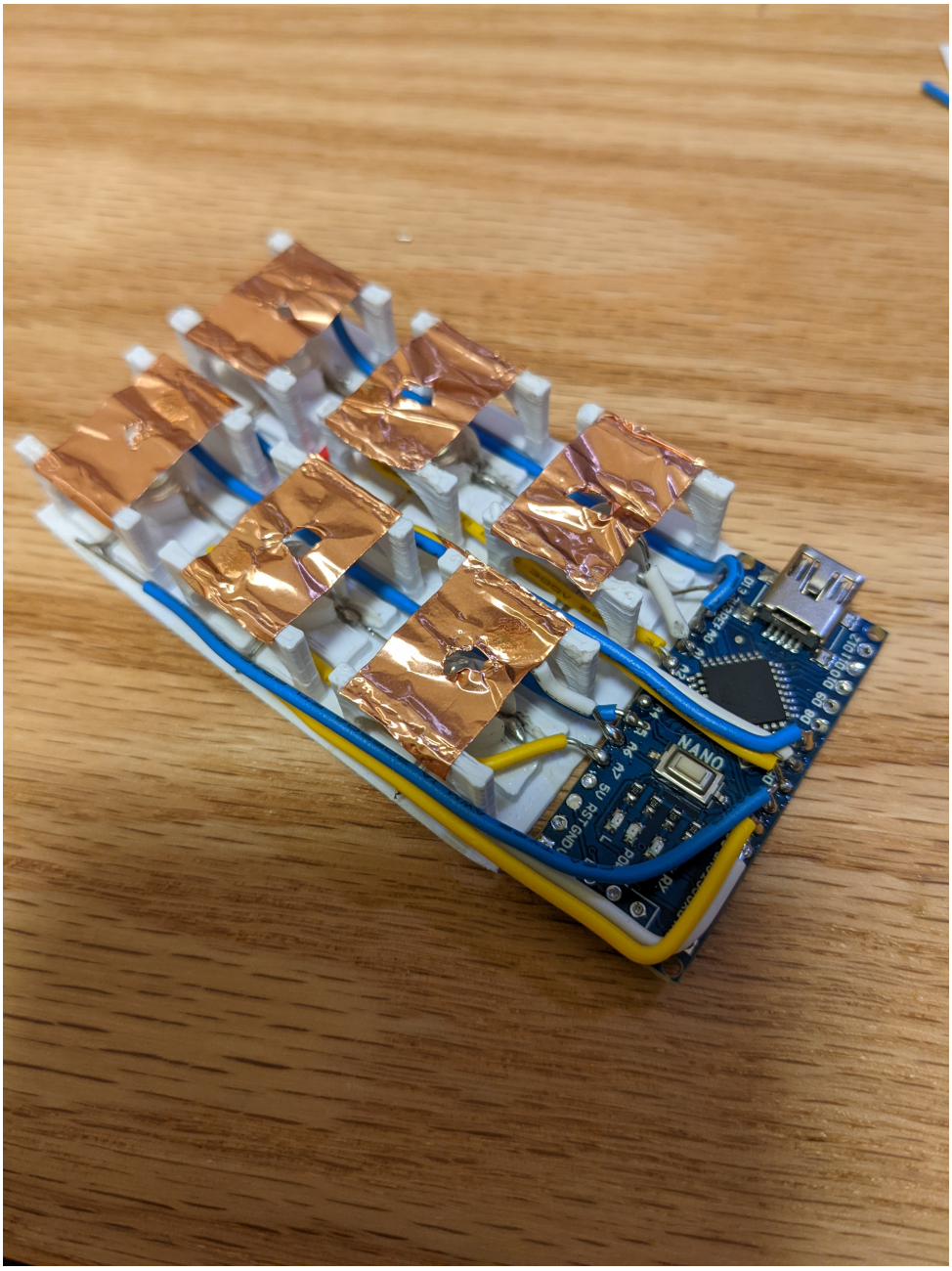

Figure 3.5: A completed Sensor Bracket with Arduino Attached

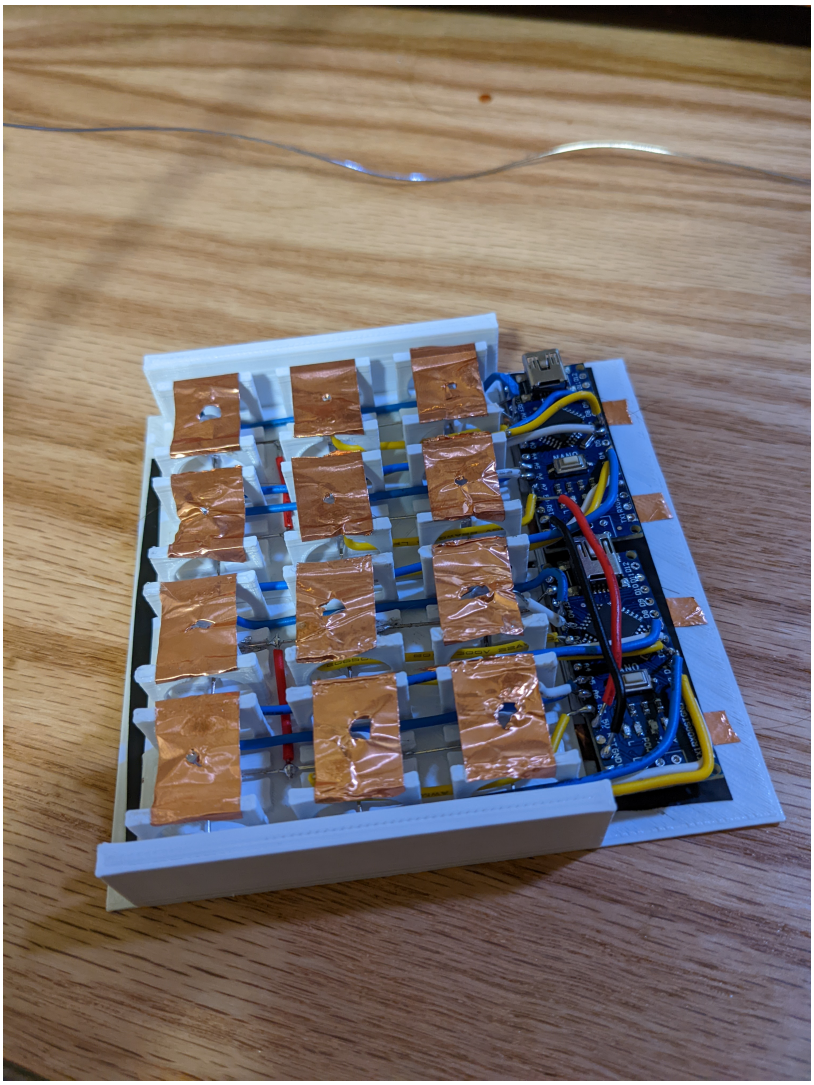

Figure 3.6: Final Sensor Assembly in Bottom Enclosure

3.1.4 Top Enclosure

The top enclosure (Figure 3.7) is what differentiates a regular keypad from this capacitive sense one. It is a solid piece of white PLA plastic with subtle debossed numbers to indicate the placement of each "button". Additionally, if you notice in Figure 3.1, the bottom enclosure only contains the horizontal enclosure walls. The top enclosure contains the the rest of the walls along with a cutout in the front right of the keypad to allow for the wires connecting the Arduino's and the Nucleo to exit the keypad enclosure. Figure 3.8 is an image of what the final keypad looks like. Note that this image is missing the wires coming out of the front right that connect to the Nucleo.

Figure 3.7: Render of the Top Enclosure

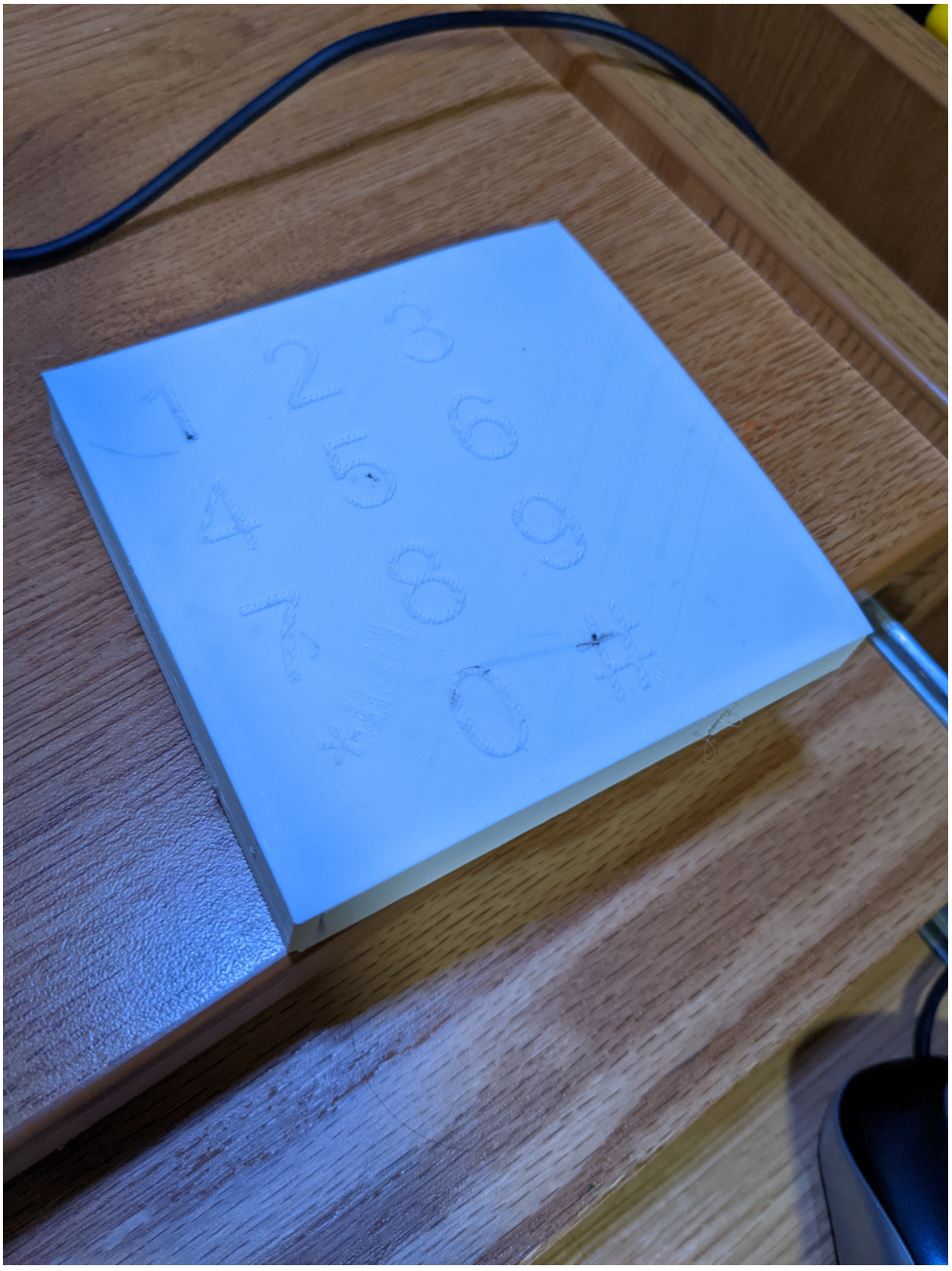

Figure 3.8: Image of the Final Keypad Enclosure

3.2 Electrical System Design

Figure 3.9 is a diagram of the power flow and data connections between each of the major components of this keypad. For simplicity, each of the six sensors and LED's in each bracket has been condensed into one sensor and LED that is representative of how the other five are connected. Once the Arduino detects that a button has been "pressed", it sends a unique code to the Nucleo. The Nucleo then interprets that code and sends the corresponding keystroke to the computer. The entire system is able to be powered directly from the USB connection between the Nucleo and the computer, so no external power source is necessary.

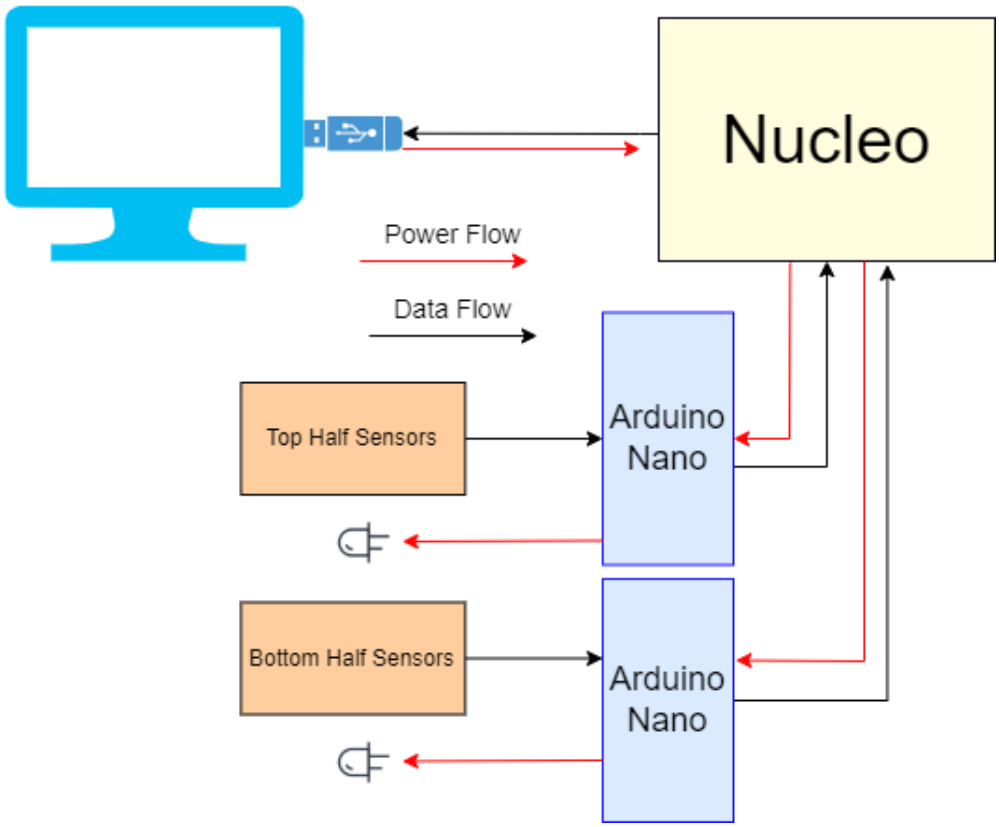

Figure 3.9: Diagram of Power and Data Flow

3.3 Electronic System Design

Figure 3.10 is the full schematic of the entire keypad. Each of the 12 sensors is connected to one analog pin of its corresponding Arduino. The LED's for each sensor are all connected together as a common anode, with the common anode connected to the 3.3V pin of the Arduino. Since the lights shine very infrequently (only when a button press is detected) and are connected to 3.3V instead of 5V there was no absolute need to have current limiting resistors to prevent the LED's from burning out. The cathode of each LED is connected to its own detected digital GPIO pin on the Arduino's.

The Arduino's are connected to the Nucleo using the SPI communication protocol with the Arduino as the master. Since the only data flow is from the Arduino to Nucleo, and each Arduino is using its own dedicated SPI line with the Nucleo (since the Nucleo has multiple SPI connections available) neither the MISO nor chip select pins are needed. Thus, each Arduino only connects to the Nucleo with three pins - MOSI, SCK (clock signal), and a single wire that acts as a signal wire from the Nucleo to Arduino to run the calibration script on the Arduino.

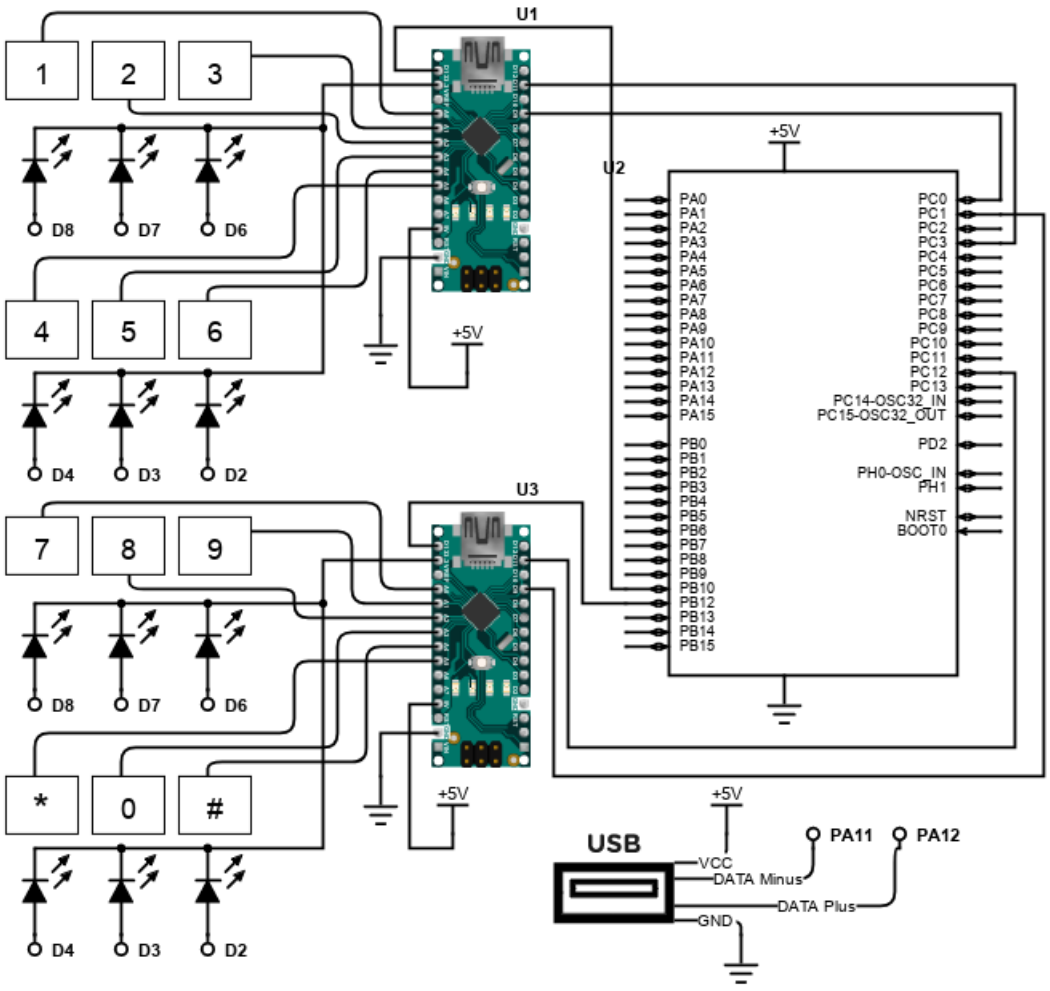

Figure 3.10: Schematic of Project

Chapter 4

Software Design

4.1 Introduction

The software for this project is made up of three distinct programs - The code running on each Arduino and the code running on the Nucleo. Each Arduino is running the exact same program except for the individual codes that it sends to the Nucleo when it detects a button press. For that reason, this section of the report will focus on the code running on only one of the Arduino's and the code running on the Nucleo.

The basic working principle is as follows. The Arduino reads a sensor and determines if its value is greater than a certain threshold value. This means the button has been "pressed". If that happens, the Arduino sends a code over SPI to the Nucleo telling it which button has been pressed. The Nucleo then interprets the code sent over SPI, determines which button was pressed, and then sends the corresponding keystroke to the computer.

The Nucleo also has the ability to calibrate the sensors. If the User button on the Nucleo is pressed it sends a calibration command to the Arduino that runs a calibration sketch to calibrate the threshold values of the sensors.

4.2 Arduino Code

Due to the complexity of capacitive sensing and time constraints a capacitive touch sensing library created by Martin Pittermann (https://github.com/martin2250/ADCTouch) was used to do the heavy lifting of reading the capacitive touch sensors. However, the library only provides functions for reading the raw values from the sensors - everything else must by implemented yourself. The program begins by including the capacitive touch sensing library and the SPI library, followed but some define macros to store the pins of each button and LED.

```
1  #include <ADCTouch.h>
2  #include <SPI.h>
3
4  #define on LOW
5  #define off HIGH
6  #define SAMPLES 50
7  #define CALIBRATE_SAMPLES 100
8  #define DELAYTIME 0
9
10 #define but1 A0
11 #define but2 A2
12 #define but3 A1
13 #define but4 A5
14 #define but5 A3
15 #define but6 A4
```

```
16
17 #define but1LED 8
18 #define but2LED 7
19 #define but3LED 6
20 #define but4LED 4
21 #define but5LED 3
22 #define but6LED 2
23
24 #define calibrateSignalPin 9
```

Listing 4.1: Macros

I then created some arrays to make the logic of the sensors much easier. This allows access to all the functions and values of each sensor by using its index in the array. For example, index 0 of each array holds all the information for the button '1', including its pin, its LED, its current value, reference value, threshold values, and the code to send. The value of each sensor is the raw value at the time of reading the senor. The reference value is the ambient value of the sensor, i.e. when no touch is present. The threshold value of each sensor is the value that the sensor must be greater than to count as a "press".

```
1 int buts[6] = {but1, but2, but3, but4, but5, but6};
2 int refs[6] = {0, 0, 0, 0, 0, 0};
3 int values[6] = {0, 0, 0, 0, 0, 0};
4 int thresholds[6] = {5, 11, 10, 5, 5, 3};
5 int leds[6] = {but1LED, but2LED, but3LED, but4LED, but5LED, but6LED};
6
7 char sendValues[6] = {'1', '2', '3', '4', '5', '6'};
```

Listing 4.2: Arrays

In the setup() function we do some basic setup stuff including starting SPI and Serial (used in debugging the code during development), setting all the LED's as outputs and turning them off, and setting the calibrate signal pin from the Nucleo as an input.

```
1  void setup() {
2
3    Serial.begin(9600);
4    SPI.begin();
5
6    for(int i = 0; i<6; i++)
7    {
8      pinMode(leds[i], OUTPUT);
9      digitalWrite(leds[i], off);
10   }
11   pinMode(calibrateSignalPin, INPUT);
12
13   //calibrate();
14   delay(1000);
15
16 }
```

Listing 4.3: Program Setup

In the main program loop we first begin by resetting the reference values every 1000 program loops using the function getRefs(), which is discussed below. This keeps the sensor from drifting to much if plugged in for an extended period of time. The purpose of the oneTimeCounter variable is to reset the reference values right away after startup. We then check the value of the calibrate signal from the Nucleo. If it is high, we calibrate the sensor. Next we use a for loop to loop through each sensor using its index in the array. For each sensor, we find its value after subtracting its corresponding reference value. This essentially gives us a

number for how close the finger is to the sensor. A high number means the finger is very close or touching, and a value of 0 means the sensor has the same value as the ambient value. If that value is greater than the corresponding threshold value, then we turn the LED backlight under the sensor on and send the code over SPI to the Nucleo to let it know that the button was pressed. Otherwise we simply leave the LED off.

```
void loop() {

  if(counter == 1000 || oneTimeCounter == 10)
  {
    getRefs();
    counter = 0;
  }

  calibrateFlag = digitalRead(calibrateSignalPin);
  if(calibrateFlag == 1)
  {
    calibrate();
  }

  for(int i = 0; i<6; i++)
  {
    values[i] = (ADCTouch.read(buts[i], SAMPLES) - refs[i]);
    Serial.print(values[i]);
    Serial.print("␣␣␣");
    if(values[i] > thresholds[i])
    {
      digitalWrite(leds[i], on);
      SPI.transfer(sendValues[i]);
    }
    else
    {
      digitalWrite(leds[i], off);
    }
    delay(DELAYTIME);
  }

  Serial.println("");
  counter++;
  oneTimeCounter++;
}
```

Listing 4.4: Main Program Loop

The getRefs() function below is used every time we need to get the ambient reference values from the sensors. It simply loops through each sensor with a very high number of samples and sets its corresponding reference. The high number of samples takes some time for the Arduino to process, but ultimately results in a much more accurate result.

```
void getRefs()
{
 delay(100);
 for(int i = 0; i<6; i++)
 {
   refs[i] = ADCTouch.read(buts[i], 200);
   delay(DELAYTIME);
 }
```

```
 delay(100);
}
```

Listing 4.5: getRefs() Function

Finally, we move onto the calibrate function. This function runs whenever the Nucleo sends the calibration signal to the Arduino. First, the program turns all the LED's under all the sensors on to let the user know that the sensor is now calibrating. For each sensor, the program turns on its corresponding LED, waits for two seconds, gets the reference value for that sensor, and then turns the LED off. This signals to the user to put their finger above that sensor. The program waits two seconds, then reads the value of the sensor with a high sample size to get the average value of the sensor with a finger on it. It then sets the threshold values for each sensor by subtracting a certain amount from the sensor value. The amount to subtract was determined empirically and changes heavily depending on the individual details of each sensor. Finally, the program blinks all the LEDs again to inform the user that calibration is now over.

```
void calibrate()
{
  for(int i = 0; i<6; i++)
  {
    digitalWrite(leds[i], on);
  }
  delay(2000);

  for(int i = 0; i<6; i++)
  {
    digitalWrite(leds[i], off);
  }
  delay(1000);

  for(int i = 0; i<6; i++)
  {
    digitalWrite(leds[i], on);
    delay(2000);
    getRefs();
    digitalWrite(leds[i], off);
    delay(2000);

    for(int j = 0; j<CALIBRATE_SAMPLES; j++)
    {
      values[i] += (ADCTouch.read(buts[i], SAMPLES) - refs[i]);
    }
    values[i] = values[i]/CALIBRATE_SAMPLES;
    delay(500);
  }
  thresholds[0] = values[0] - 2;
  thresholds[1] = values[1] - 3;
  thresholds[2] = values[2] - 3;
  thresholds[3] = values[3] - 3;
  thresholds[4] = values[4] - 8;
  thresholds[5] = values[5] - 8;

  Serial.println("Threshold␣Values");
  for(int i = 0; i<6; i++)
  {
    Serial.print(thresholds[i]);
    Serial.print("␣␣␣");
```

```
  }
  Serial.println("");

  for(int i = 0; i<6; i++)
  {
    digitalWrite(leds[i], on);
  }
  delay(2000);

  for(int i = 0; i<6; i++)
  {
    digitalWrite(leds[i], off);
  }
  delay(1000);
  getRefs();
  delay(1000);
}
```

Listing 4.6: Calibrate Function

4.3 Nucleo Code

While the actual program running on the Nucleo is split up into many different files, the only code I wrote was in the main program in the main.c file. As such, it will be the only file included in this report. I started by configuring all the pins in the graphical interface (Figure 4.1) to their intended functions. This included enabling the USB-On-The-Go Protocol, SPI, and setting the USB parameters. The USB parameters allowed me to setup the Nucleo as a HID device, specifically as a keyboard, so that the computer detects the Nucleo as a keyboard. I then created a structure to hold the key press information that is consistent with the USB keyboard standard.

```

typedef struct
{
    uint8_t MODIFIER;
    uint8_t RESERVED;
    uint8_t KEYCODE1;
    uint8_t KEYCODE2;
    uint8_t KEYCODE3;
    uint8_t KEYCODE4;
    uint8_t KEYCODE5;
    uint8_t KEYCODE6;

} keyboardHID;

keyboardHID keyboard = {0,0,0,0,0,0,0,0};
```

Listing 4.7: Keyboard Struct

In the main program loop I first write the calibration signal pins that go to each Arduino low. I then check if the built-in button was pressed. If it was, I set the calibration signal to each Arduino high, wait a second, and then set it back to low. This tells the Arduino to run its calibration function. I also check the see if any SPI communication is happening on either of the two SPI lines. If there is, i.e. the Arduino just sent a code that a button was pressed, then that code is stored in a buffer.

```
  while (1)
  {
      HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0, 0);
      HAL_GPIO_WritePin(GPIOC, GPIO_PIN_1, 0);

      if(buttonState == 1)
```

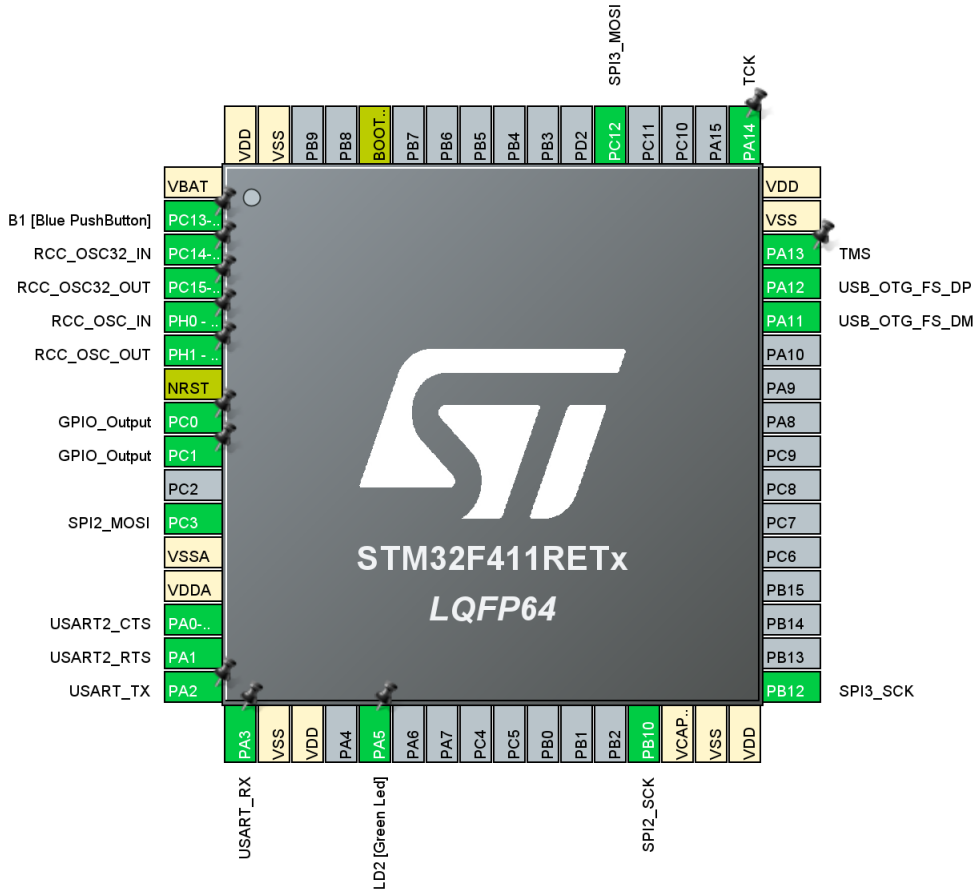

Figure 4.1: Graphical Representation of Pin Configurations

```
7       {
8           HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0, 1);
9           HAL_GPIO_WritePin(GPIOC, GPIO_PIN_1, 1);
10          HAL_Delay(1000);
11          HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0, 0);
12          HAL_GPIO_WritePin(GPIOC, GPIO_PIN_1, 0);
13
14          buttonState = 0;
15      }
16
17      HAL_SPI_Receive(&hspi2, (uint8_t *)buff, 1, TIMEOUT_VALUE);
18      HAL_Delay(50);
19      HAL_SPI_Receive(&hspi3, (uint8_t *)buff2, 1, TIMEOUT_VALUE);
```

Listing 4.8: Start of Program Loop

The rest of the main program is devoted to sending the keystroke information. This was accomplished using a long series of if statements. While this is not the most elegant, it does accomplish the intended function and was easy to implement. Below, you can see that the program simply checks the value of the SPI buffer and compares it to the character it represents. If they match, then the program sends that corresponding keystroke, empties the buffer, and delays for a certain amount of time to prevent multiple keystrokes from sending simultaneously. Note that the excerpt below only shows half of the if statements, but the other half continues the same pattern.

```
1
2       if(buff[0] == '1'){
3           sendKeystroke(0x59);
4           strcpy(buff, "");
5           HAL_Delay(DELAYTIME);
6       }
7       if(buff[0] == '2'){
8           sendKeystroke(0x5A);
9           strcpy(buff, "");
10          HAL_Delay(DELAYTIME);
11      }
12      if(buff[0] == '3'){
13          sendKeystroke(0x5B);
14          strcpy(buff, "");
15          HAL_Delay(DELAYTIME);
16      }
17      if(buff[0] == '4'){
18          sendKeystroke(0x5C);
19          strcpy(buff, "");
20          HAL_Delay(DELAYTIME);
21      }
22      if(buff[0] == '5'){
23          sendKeystroke(0x5D);
24          strcpy(buff, "");
25          HAL_Delay(DELAYTIME);
26      }
27      if(buff[0] == '6'){
28          sendKeystroke(0x5E);
29          strcpy(buff, "");
30          HAL_Delay(DELAYTIME);
31      }
```

Listing 4.9: Main Program Flow

The sendKeystroke() function takes a hex key code as input and then sets that hex value as the key code value in the keyboard structure. The hex values were found by consulting the USB keyboard standard. The program then uses a built in function to send the keystroke report to the computer. Since the computer only receives the report that a button was "pressed" it needs to be sent a report that the button was "unpressed" to simulate an actual button press. Thus, the program immediately sends an empty report to indicate the button was released.

```
1
2 void sendKeystroke(int keycode)
3 {
4     keyboard.KEYCODE1 = keycode;
5     USBD_HID_SendReport(&hUsbDeviceFS, (uint8_t*)&keyboard, sizeof(keyboard));
6     HAL_Delay(50);
7
8     keyboard.KEYCODE1 = 0x00;
9     USBD_HID_SendReport(&hUsbDeviceFS, (uint8_t*)&keyboard, sizeof(keyboard));
10    HAL_Delay(50);
11 }
```

Listing 4.10: sendKeystroke Function

Chapter 5

Testing Requirements

The following is a detailed examination of my project compared to the defined requirements. In general, my keypad satisfies all of the requirements as defined in Chapter 2: Project Requirements. However, it falls a bit short in terms of the sensitivity of the sensors. They work quite well most of the time, but sometimes a button is registered as pressed even though it is not and vice versa. While this technically means it failed a requirement, I believe it still met the spirit of the requirement and as such I consider this keypad to satisfy all of the requirements.

5.1 Hardware Testing Requirements

Requirement	**Meets Requirement?**	**Explanation**
The layout of the keypad will be as shown in Figure 2.1	Yes	See Figure 3.8
Enclosure should be simple and sleek, with a "sci-fi" aesthetic	Yes	Subjective, but I believe it can be classified as simple and sleek. It does have a "sci-fi" aesthetic to it with the white color and buttonless interface
Enclosure should be made out of white PLA with markings on top surface to indicate placement of buttons	Yes	See Figure 3.8
Top surface should be a smooth, solid piece of plastic	Yes	See Figure 3.8
The actual touch sensors should be hidden in the inside of the enclosure and not visible on the outside	Yes	See Figure 3.8
Each button on the keypad should have an LED backlight associated with it	Yes	See Figure 3.4
The light from the LED's should penetrate the top surface and be clearly visible	Yes	The light is able to penetrate the top surface just fine
The LED should turn on when its corresponding button has been pressed	Yes	See Arduino Program Code. The LED is also visually confirmed to turn on
A pair of Arduino Nano's should be used only to read the sensors and relay that information back to the Nucleo	Yes	See Arduino Program Code

5.2 Software Testing Requirements

Requirement	Meets Requirement?	Explanation
The software the control and detect the capacitive sensors should be written in Arduino C and hosted on a pair of Arduino Nano's	Yes	See Arduino Program Code
The software should be sensitive enough to detect the capacitance of a human finger without the finger actually touching the metal sensors	Yes	All the sensors have this ability, its controlling this ability where there are a few problems
The software should be able to reliably detect the presence of a human finger above each keypad	Sometimes	The values that the Arduino reads from the sensors vary considerably between readings, which makes setting consistent threshold values very difficult. While they work well most of the time, occasionally the sensor will simply fail to detect your finger or will detect your finger even if it is not there.
The software should detect a finger when it is directly above the button, but not if it is over an adjacent button	Sometimes	See explanation above
The software should be fast enough to handle multiple buttons being pressed in quick succession	Yes	You can lay your entire hand on the keypad and get all the sensors to trigger at once if you want
The Arduino's should be able to communicate with the Nucleo using I^2C or SPI	Yes	They communicate using SPI
The software on the Nucleo should receive the SPI or I^2C signals coming from the Arduino's and interpret that information to determine which button was pressed	Yes	See Nucleo Code
The Nucleo should present itself to a connected computer as a USB keyboard	Yes	See Nucleo Code, also verified by plugging keypad into a computer and checking connected devices
The Nucleo should send the keystroke information to the Computer and have the keystrokes appear on the computer as if you typed them yourself	Yes	See Nucleo Code, also verified by hitting any key on the keypad while it is connected to a computer

Chapter 6

Users Guide

6.1 Introduction

This keypad is incredibly easy to use. Simply plug in the keypad to any USB port on any computer that supports keyboards, wait a couple seconds to allow the keypad to get setup, and then begin typing away.

6.1.1 Calibrating the Keypad

While it should not be needed, if the keypad sensitivity is too high (buttons being "pressed" without you actually pressing them) or too low (the buttons not being "pressed" even if you have your finger over the sensor) then a calibration may be necessary to return the keypad to a working order. To perform the calibration follow the steps below.

1. Press the Blue User Button on the Nucleo
2. All the LED's will turn on briefly, then turn off. After they turn off, The LED's corresponding to the '1' and '7' keys will turn on
3. As soon as those LED's turn off, place a finger over each key
4. Keep finger there until the next pair of LEDs turn on, then immediately remove the fingers from the keypad. Once that next pair of LEDs turns off, place your two fingers over that next pair of LED's
5. Repeat until all keys have been calibrated
6. After the last pair of keys has been calibrated, all the LED's will flash briefly to indicate calibration is complete
7. Wait a couple seconds to allow the keypad to stabilize its sensitivity

Chapter 7

Lessons Learned

I learned a significant amount from this project. First, I overestimated the complexity of this project. I have messed around with capacitive sensors before, but never this many that are this close together. Getting one sensor to reliably detect your finger above a surface is hard enough - getting 12 sensors that close in proximity to each other to do this without any interference was essentially impossible. I have also wondered why I have not seen any capacitive sense keyboards or even other keypads - now I know why. The technology is cool, but definitely impractical if you need perfect, consistent results each time. In that regard, the physical buttons are much better.

Second, and somewhat related, is that I chose quite an ambitious project. While I very much enjoyed making this, because I enjoy these kinds of projects, it did take up a very large amount of time especially the two weeks before I submitted this. I am lucky that I was able to devote as much time to it as I did because otherwise I might have run out of time.

On a more positive note, I learned a lot more about capacitive sensing and how it works on a fundamental level. While it may not be the best technology to use in a situation like a keypad or keyboard, I still think it is very cool and useful in other circumstances. I also learned a lot about how USB keyboards and the keyboard communication standard works. It was actually a lot simpler than I thought, and is something that I might want to incorporate in my future projects.

Appendix A

Source Code

```
#include <ADCTouch.h>
#include <SPI.h>

#define on LOW
#define off HIGH
#define SAMPLES 50
#define CALIBRATE_SAMPLES 100
#define DELAYTIME 0

#define but1 A0
#define but2 A2
#define but3 A1
#define but4 A5
#define but5 A3
#define but6 A4

#define but1LED 8
#define but2LED 7
#define but3LED 6
#define but4LED 4
#define but5LED 3
#define but6LED 2

#define calibrateSignalPin 9

int buts[6] = {but1, but2, but3, but4, but5, but6};
int refs[6] = {0, 0, 0, 0, 0, 0};
int values[6] = {0, 0, 0, 0, 0, 0};
int thresholds[6] = {5, 11, 10, 5, 5, 3};
int leds[6] = {but1LED, but2LED, but3LED, but4LED, but5LED, but6LED};

char sendValues[6] = {'1', '2', '3', '4', '5', '6'};

int counter = 0;
int oneTimeCounter = 0;
int calibrateFlag = 0;

void setup() {

```

```
  Serial.begin(9600);
  SPI.begin();

  for(int i = 0; i<6; i++)
  {
    pinMode(leds[i], OUTPUT);
    digitalWrite(leds[i], off);
  }
  pinMode(calibrateSignalPin, INPUT);

  //calibrate();
  delay(1000);

}

void loop() {

  if(counter == 1000 || oneTimeCounter == 10)
  {
    getRefs();
    counter = 0;
  }

  calibrateFlag = digitalRead(calibrateSignalPin);
  if(calibrateFlag == 1)
  {
    calibrate();
  }

  for(int i = 0; i<6; i++)
  {
    values[i] = (ADCTouch.read(buts[i], SAMPLES) - refs[i]);
    Serial.print(values[i]);
    Serial.print("␣␣␣");
    if(values[i] > thresholds[i])
    {
      digitalWrite(leds[i], on);
      SPI.transfer(sendValues[i]);
    }
    else
    {
      digitalWrite(leds[i], off);
    }
    delay(DELAYTIME);
  }

  Serial.println("");
  counter++;
  oneTimeCounter++;
}

void getRefs()
{
 delay(100);
```

```
 for(int i = 0; i<6; i++)
 {
   refs[i] = ADCTouch.read(buts[i], 200);
   delay(DELAYTIME);
 }
 delay(100);
}

void calibrate()
{
  for(int i = 0; i<6; i++)
  {
    digitalWrite(leds[i], on);
  }
  delay(2000);

  for(int i = 0; i<6; i++)
  {
    digitalWrite(leds[i], off);
  }
  delay(1000);

  for(int i = 0; i<6; i++)
  {
    digitalWrite(leds[i], on);
    delay(2000);
    getRefs();
    digitalWrite(leds[i], off);
    delay(2000);

    for(int j = 0; j<CALIBRATE_SAMPLES; j++)
    {
      values[i] += (ADCTouch.read(buts[i], SAMPLES) - refs[i]);
    }
    values[i] = values[i]/CALIBRATE_SAMPLES;
    delay(500);
  }
  thresholds[0] = values[0] - 2;
  thresholds[1] = values[1] - 3;
  thresholds[2] = values[2] - 3;
  thresholds[3] = values[3] - 3;
  thresholds[4] = values[4] - 8;
  thresholds[5] = values[5] - 8;

  Serial.println("Threshold Values");
  for(int i = 0; i<6; i++)
  {
    Serial.print(thresholds[i]);
    Serial.print("   ");
  }
  Serial.println("");

  for(int i = 0; i<6; i++)
  {
```

```
    digitalWrite(leds[i], on);
  }
  delay(2000);

  for(int i = 0; i<6; i++)
  {
    digitalWrite(leds[i], off);
  }
  delay(1000);
  getRefs();
  delay(1000);
}
```

Listing A.1: Arduino Source Code

```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * <h2><center>&copy; Copyright (c) 2021 STMicroelectronics.
  * All rights reserved.</center></h2>
  *
  * This software component is licensed by ST under BSD 3-Clause license,
  * the "License"; You may not use this file except in compliance with the
  * License. You may obtain a copy of the License at:
  *                        opensource.org/licenses/BSD-3-Clause
  *
  ******************************************************************************
  */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "usb_device.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */

#include "string.h"
#include "stdlib.h"
#include "stdio.h"
#include "stdint.h"
#include "usbd_hid.h"

/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

typedef struct
{
    uint8_t MODIFIER;
    uint8_t RESERVED;
    uint8_t KEYCODE1;
    uint8_t KEYCODE2;
    uint8_t KEYCODE3;
    uint8_t KEYCODE4;
    uint8_t KEYCODE5;
    uint8_t KEYCODE6;

} keyboardHID;

```

```
keyboardHID keyboard = {0,0,0,0,0,0,0,0};


extern USBD_HandleTypeDef hUsbDeviceFS;

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

#define TIMEOUT_VALUE 50
#define DELAYTIME 250


/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
SPI_HandleTypeDef hspi2;
SPI_HandleTypeDef hspi3;

UART_HandleTypeDef huart2;

/* USER CODE BEGIN PV */

int buttonState = 0;
char buff[20];
char buff2[20];
char newline[] = "\r\n";

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_SPI2_Init(void);
static void MX_SPI3_Init(void);
/* USER CODE BEGIN PFP */

void sendKeystroke(int keycode);

/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */


  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();
```

```
119
120 /* USER CODE BEGIN Init */
121
122 /* USER CODE END Init */
123
124 /* Configure the system clock */
125 SystemClock_Config();
126
127 /* USER CODE BEGIN SysInit */
128
129 /* USER CODE END SysInit */
130
131 /* Initialize all configured peripherals */
132 MX_GPIO_Init();
133 MX_USART2_UART_Init();
134 MX_USB_DEVICE_Init();
135 MX_SPI2_Init();
136 MX_SPI3_Init();
137 /* USER CODE BEGIN 2 */
138
139
140 HAL_Delay(1000);
141
142 /* USER CODE END 2 */
143
144 /* Infinite loop */
145 /* USER CODE BEGIN WHILE */
146 while (1)
147 {
148     HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0, 0);
149     HAL_GPIO_WritePin(GPIOC, GPIO_PIN_1, 0);
150
151     if(buttonState == 1)
152     {
153         HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0, 1);
154         HAL_GPIO_WritePin(GPIOC, GPIO_PIN_1, 1);
155         HAL_Delay(1000);
156         HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0, 0);
157         HAL_GPIO_WritePin(GPIOC, GPIO_PIN_1, 0);
158
159         buttonState = 0;
160     }
161
162     HAL_SPI_Receive(&hspi2, (uint8_t *)buff, 1, TIMEOUT_VALUE);
163     HAL_Delay(50);
164     HAL_SPI_Receive(&hspi3, (uint8_t *)buff2, 1, TIMEOUT_VALUE);
165
166     //HAL_UART_Transmit(&huart2, (uint8_t *)buff2, strlen(buff2), HAL_MAX_DELAY);
167     //HAL_UART_Transmit(&huart2, (uint8_t *)newline, strlen(newline), HAL_MAX_DELAY);
168
169     //TOP HALF OF KEYBOARD CHECKS
170
171     if(buff[0] == '1'){
172         sendKeystroke(0x59);
173         strcpy(buff, "");
174         HAL_Delay(DELAYTIME);
175     }
176     if(buff[0] == '2'){
177         sendKeystroke(0x5A);
178         strcpy(buff, "");
179         HAL_Delay(DELAYTIME);
180     }
181     if(buff[0] == '3'){
182         sendKeystroke(0x5B);
183         strcpy(buff, "");
184         HAL_Delay(DELAYTIME);
185     }
186     if(buff[0] == '4'){
```

```
187             sendKeystroke(0x5C);
188             strcpy(buff, "");
189             HAL_Delay(DELAYTIME);
190         }
191         if(buff[0] == '5'){
192             sendKeystroke(0x5D);
193             strcpy(buff, "");
194             HAL_Delay(DELAYTIME);
195         }
196         if(buff[0] == '6'){
197             sendKeystroke(0x5E);
198             strcpy(buff, "");
199             HAL_Delay(DELAYTIME);
200         }
201 
202         //BOTTOM HALF OF KEYPAD CHECKS
203 
204         if(buff2[0] == '7'){
205             sendKeystroke(0x5F);
206             strcpy(buff2, "");
207             HAL_Delay(DELAYTIME);
208         }
209         if(buff2[0] == '8'){
210             sendKeystroke(0x60);
211             strcpy(buff2, "");
212             HAL_Delay(DELAYTIME);
213         }
214         if(buff2[0] == '9'){
215             sendKeystroke(0x61);
216             strcpy(buff2, "");
217             HAL_Delay(DELAYTIME);
218         }
219         if(buff2[0] == '0'){
220             sendKeystroke(0x62);
221             strcpy(buff2, "");
222             HAL_Delay(DELAYTIME);
223         }
224         if(buff2[0] == '*'){
225             sendKeystroke(0x55);
226             strcpy(buff2, "");
227             HAL_Delay(DELAYTIME);
228         }
229         if(buff2[0] == '#'){      //this one is different because some computers do not support the
        keypad # code so I had to use left shift and 3
230 
231             keyboard.MODIFIER = 0x02;
232             keyboard.KEYCODE1 = 0x20;
233             USBD_HID_SendReport(&hUsbDeviceFS, (uint8_t*)&keyboard, sizeof(keyboard));
234             HAL_Delay(50);
235 
236             keyboard.MODIFIER = 0x00;
237             keyboard.KEYCODE1 = 0x00;
238             USBD_HID_SendReport(&hUsbDeviceFS, (uint8_t*)&keyboard, sizeof(keyboard));
239             HAL_Delay(50);
240 
241             strcpy(buff2, "");
242             HAL_Delay(DELAYTIME);
243         }
244         else{
245             strcpy(buff, "");
246             strcpy(buff2, "");
247         }
248 
249 
250     /* USER CODE END WHILE */
251 
252     /* USER CODE BEGIN 3 */
253   }
```

```
254   /* USER CODE END 3 */
255 }
256
257 /**
258   * @brief System Clock Configuration
259   * @retval None
260   */
261 void SystemClock_Config(void)
262 {
263   RCC_OscInitTypeDef RCC_OscInitStruct = {0};
264   RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
265
266   /** Configure the main internal regulator output voltage
267   */
268   __HAL_RCC_PWR_CLK_ENABLE();
269   __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);
270   /** Initializes the RCC Oscillators according to the specified parameters
271   * in the RCC_OscInitTypeDef structure.
272   */
273   RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
274   RCC_OscInitStruct.HSEState = RCC_HSE_BYPASS;
275   RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
276   RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
277   RCC_OscInitStruct.PLL.PLLM = 4;
278   RCC_OscInitStruct.PLL.PLLN = 72;
279   RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
280   RCC_OscInitStruct.PLL.PLLQ = 3;
281   if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
282   {
283     Error_Handler();
284   }
285   /** Initializes the CPU, AHB and APB buses clocks
286   */
287   RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
288                                 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
289   RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
290   RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
291   RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
292   RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
293
294   if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) != HAL_OK)
295   {
296     Error_Handler();
297   }
298 }
299
300 /**
301   * @brief SPI2 Initialization Function
302   * @param None
303   * @retval None
304   */
305 static void MX_SPI2_Init(void)
306 {
307
308   /* USER CODE BEGIN SPI2_Init 0 */
309
310   /* USER CODE END SPI2_Init 0 */
311
312   /* USER CODE BEGIN SPI2_Init 1 */
313
314   /* USER CODE END SPI2_Init 1 */
315   /* SPI2 parameter configuration*/
316   hspi2.Instance = SPI2;
317   hspi2.Init.Mode = SPI_MODE_SLAVE;
318   hspi2.Init.Direction = SPI_DIRECTION_2LINES_RXONLY;
319   hspi2.Init.DataSize = SPI_DATASIZE_8BIT;
320   hspi2.Init.CLKPolarity = SPI_POLARITY_LOW;
321   hspi2.Init.CLKPhase = SPI_PHASE_1EDGE;
```

```
322   hspi2.Init.NSS = SPI_NSS_SOFT;
323   hspi2.Init.FirstBit = SPI_FIRSTBIT_MSB;
324   hspi2.Init.TIMode = SPI_TIMODE_DISABLE;
325   hspi2.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
326   hspi2.Init.CRCPolynomial = 10;
327   if (HAL_SPI_Init(&hspi2) != HAL_OK)
328   {
329     Error_Handler();
330   }
331   /* USER CODE BEGIN SPI2_Init 2 */
332
333   /* USER CODE END SPI2_Init 2 */
334
335 }
336
337 /**
338   * @brief SPI3 Initialization Function
339   * @param None
340   * @retval None
341   */
342 static void MX_SPI3_Init(void)
343 {
344
345   /* USER CODE BEGIN SPI3_Init 0 */
346
347   /* USER CODE END SPI3_Init 0 */
348
349   /* USER CODE BEGIN SPI3_Init 1 */
350
351   /* USER CODE END SPI3_Init 1 */
352   /* SPI3 parameter configuration*/
353   hspi3.Instance = SPI3;
354   hspi3.Init.Mode = SPI_MODE_SLAVE;
355   hspi3.Init.Direction = SPI_DIRECTION_2LINES_RXONLY;
356   hspi3.Init.DataSize = SPI_DATASIZE_8BIT;
357   hspi3.Init.CLKPolarity = SPI_POLARITY_LOW;
358   hspi3.Init.CLKPhase = SPI_PHASE_1EDGE;
359   hspi3.Init.NSS = SPI_NSS_SOFT;
360   hspi3.Init.FirstBit = SPI_FIRSTBIT_MSB;
361   hspi3.Init.TIMode = SPI_TIMODE_DISABLE;
362   hspi3.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
363   hspi3.Init.CRCPolynomial = 10;
364   if (HAL_SPI_Init(&hspi3) != HAL_OK)
365   {
366     Error_Handler();
367   }
368   /* USER CODE BEGIN SPI3_Init 2 */
369
370   /* USER CODE END SPI3_Init 2 */
371
372 }
373
374 /**
375   * @brief USART2 Initialization Function
376   * @param None
377   * @retval None
378   */
379 static void MX_USART2_UART_Init(void)
380 {
381
382   /* USER CODE BEGIN USART2_Init 0 */
383
384   /* USER CODE END USART2_Init 0 */
385
386   /* USER CODE BEGIN USART2_Init 1 */
387
388   /* USER CODE END USART2_Init 1 */
389   huart2.Instance = USART2;
```

```
  huart2.Init.BaudRate = 9600;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_RTS_CTS;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART2_Init 2 */

  /* USER CODE END USART2_Init 2 */

}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0|GPIO_PIN_1, GPIO_PIN_RESET);

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pin : B1_Pin */
  GPIO_InitStruct.Pin = B1_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);

  /*Configure GPIO pins : PC0 PC1 */
  GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

  /*Configure GPIO pin : LD2_Pin */
  GPIO_InitStruct.Pin = LD2_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);

  /* EXTI interrupt init*/
  HAL_NVIC_SetPriority(EXTI15_10_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);

}

/* USER CODE BEGIN 4 */

void sendKeystroke(int keycode)
{
```

```
458     keyboard.KEYCODE1 = keycode;
459     USBD_HID_SendReport(&hUsbDeviceFS, (uint8_t*)&keyboard, sizeof(keyboard));
460     HAL_Delay(50);
461 
462     keyboard.KEYCODE1 = 0x00;
463     USBD_HID_SendReport(&hUsbDeviceFS, (uint8_t*)&keyboard, sizeof(keyboard));
464     HAL_Delay(50);
465 }
466 
467 
468 
469 
470 /* USER CODE END 4 */
471 
472 /**
473   * @brief  This function is executed in case of error occurrence.
474   * @retval None
475   */
476 void Error_Handler(void)
477 {
478   /* USER CODE BEGIN Error_Handler_Debug */
479   /* User can add his own implementation to report the HAL error return state */
480   __disable_irq();
481   while (1)
482   {
483   }
484   /* USER CODE END Error_Handler_Debug */
485 }
486 
487 #ifdef  USE_FULL_ASSERT
488 /**
489   * @brief  Reports the name of the source file and the source line number
490   *         where the assert_param error has occurred.
491   * @param  file: pointer to the source file name
492   * @param  line: assert_param error line source number
493   * @retval None
494   */
495 void assert_failed(uint8_t *file, uint32_t line)
496 {
497   /* USER CODE BEGIN 6 */
498   /* User can add his own implementation to report the file name and line number,
499      ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
500   /* USER CODE END 6 */
501 }
502 #endif /* USE_FULL_ASSERT */
503 
504 /************************ (C) COPYRIGHT STMicroelectronics *****END OF FILE****/
```

Listing A.2: Nucleo Source Code

Appendix B

Bill of Materials

Item	**Quantity**
NUCLEO-STM32F411RE	1
Arduino Nano	2
USB Cable	1
White LED's	12
Aluminum Foil	About 10in^2
1-inch Wide Copper Foil Strip	About 1 Foot
Electrical Tape	About 2 Feet
Hookup Wire	A Lot