Remote Wah Guitar Pedal

ECE 445 Final Report - Fall 2023

Julian Brookfield, Luna Rathod, Chris Read
Team 14

TA: Stasiu Chyczewski
Professor: Victoria Shao

Table Of Contents

Introduction

Guitar and bass players have a wealth of effects pedals to choose from in order to modify the sound of their instrument, such as adding distortion, echo, reverb, etc. In most cases, the controls for effects pedals are set by the player beforehand and turned on and off with a footswitch, or controlled by a foot treadle to modify a single parameter, such as the sweep of a high-Q bandpass filter in a wah pedal. However, this requires the player to remain fixed in place while using the effect, which can get in the way of the performance aspect of playing live music. It would be very convenient & expressive to have a way of controlling the parameters of certain effects while maintaining the ability to move around a stage unimpeded.

Our idea is that rather than using a foot treadle to control the filter sweep of a wah pedal, the range of the filter sweep is controlled by a sensor mounted to the headstock of a guitar/bass. This allows achieving the characteristic sweep sound of a wah by swinging the guitar up and down rather than using a foot treadle, which allows the use of the effect anywhere on stage and makes for an interesting visual accompaniment that is suited for live performance (it would look pretty cool). To further aid in freedom of movement, the effect will have the ability to be remotely activated via a button mounted to the body of the guitar within convenient reach of the player.

Design

Block Diagram

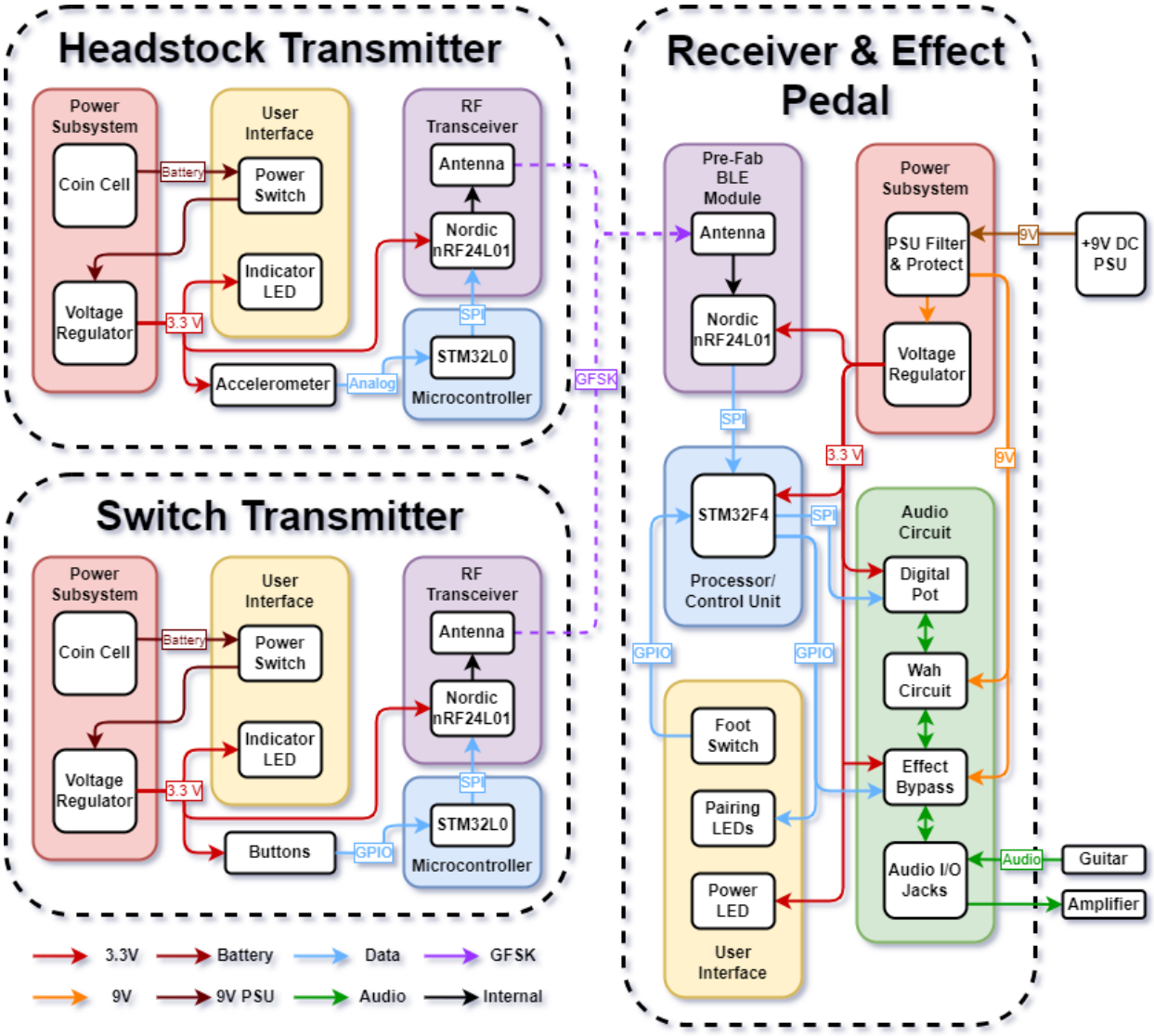


Figure 1: Block Diagram

Our solution involves the creation of three components: A headstock transmitter to wirelessly relay accelerometer data, a button interface that allows the effect to be turned on and off remotely, and the main floor pedal receiver which processes the accelerometer data into the appropriate control signals for sweeping the digital pot and routing the audio path, as well as implementing the actual wah effect.

Subsystem Designs

Headstock Transmitter

Schematic:

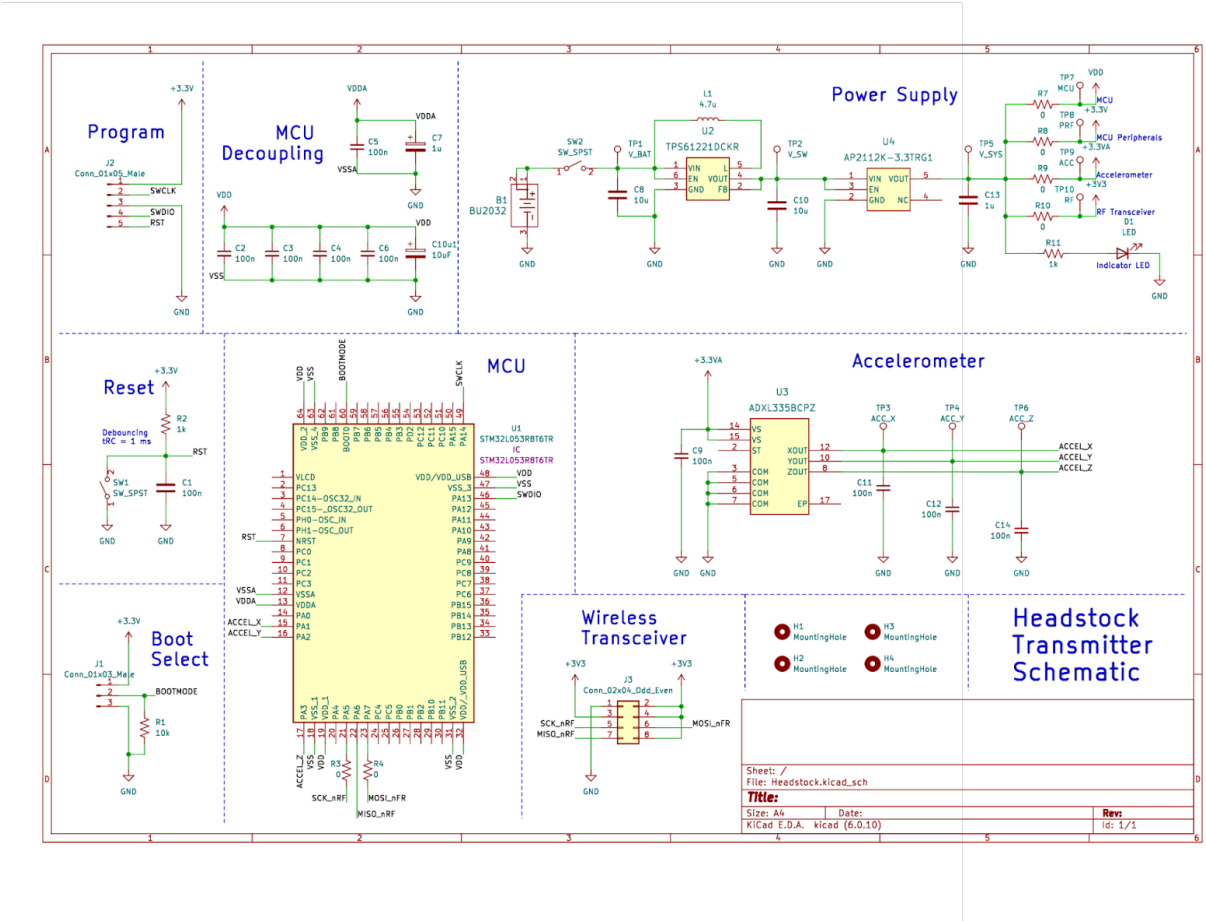


Figure 2: Headstock Transmitter Schematic

Layout:

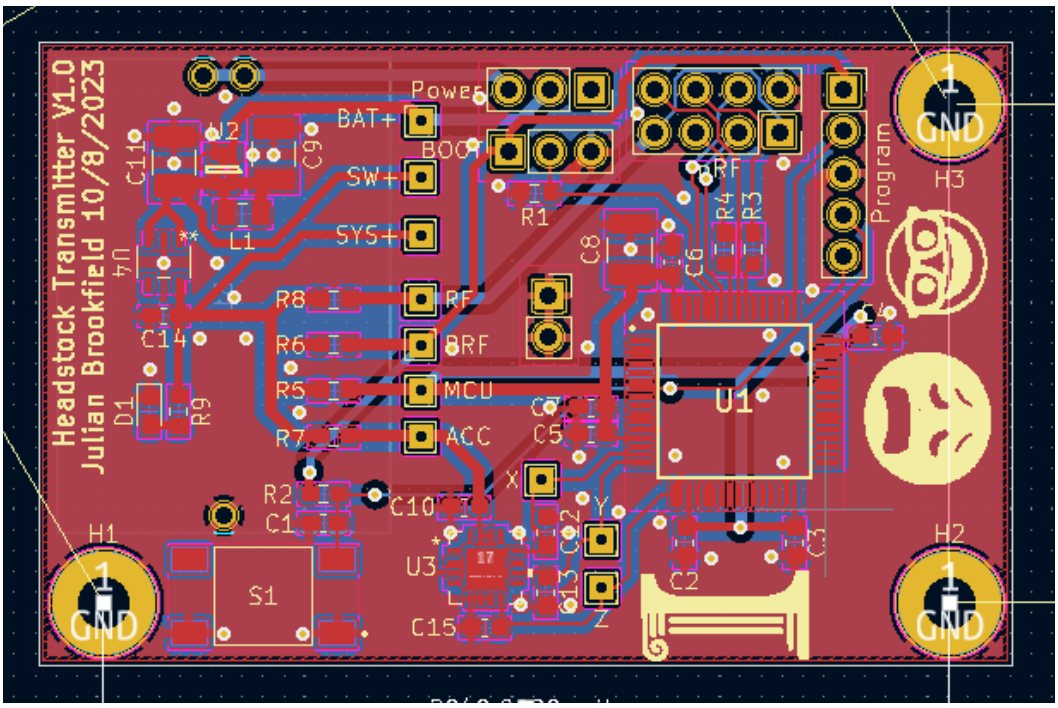


Figure 3: Headstock Transmitter PCB Layout

Final Implementation:

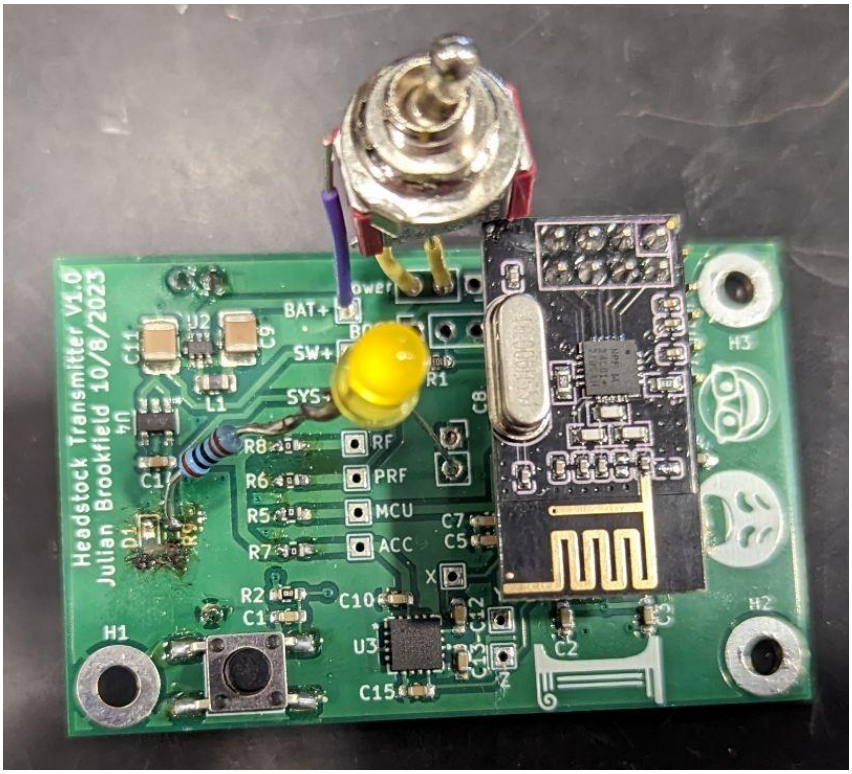


Figure 4: Final Assembled Headstock PCB

Block Description:

The headstock transmitter's primary purpose is to send the sensor data read from the on-board accelerometer back to the floor pedal receiver to be processed. Internally there will be an accelerometer, the Analog Devices ADXL355, which will output an analog voltage proportional to the acceleration of the device. An analog accelerometer was chosen for ease of use, as it keeps the number of serial interfaces that need to be programmed to a minimum. This same chip was also used successfully in previous projects [5], lending us confidence to use it. This voltage is then converted into digital data using the integrated 12-bit ADC included in the STM32L053 microcontroller. Finally, the accelerometer data is sent to a wireless transceiver, a Nordic nRF24L01 based module, and transmitted to the floor pedal receiver.

The STM and nRF chips were chosen for their extensive individual documentation [7] [8] [9] [10] [11], as well as resources that aid in their operation together [6]. The microcontroller will communicate to the transceiver using SPI, and the transceiver will send the data using GFSK in the 2.4 GHz frequency band. Nordic devices are certified by the FCC to operate in this frequency range [12], which gives us confidence to use it in our design.

The headstock transmitter is powered using a standard 3V CR2032 coin cell battery, which is then up-converted using a boost converter [14] to achieve 3.3V

operation for the rest of the circuit and further regulated with an LDO [15]. Boost converters operate at high frequencies which can leak into the supply rails [16], so the additional LDO is included to reduce any ripples should they occur. A power switch is also included to turn the effect off when not in use, as well as a power indicator LED.

Button Transmitter

Schematic:

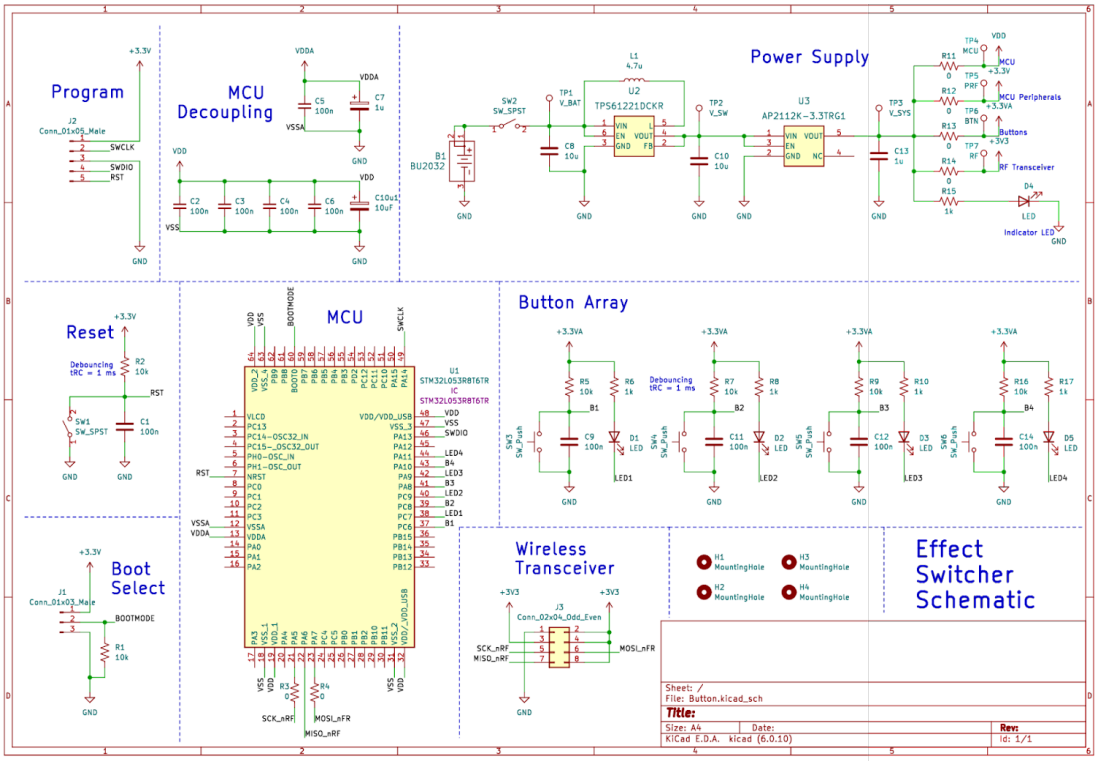


Figure 5: Button Transmitter Schematic

Layout:

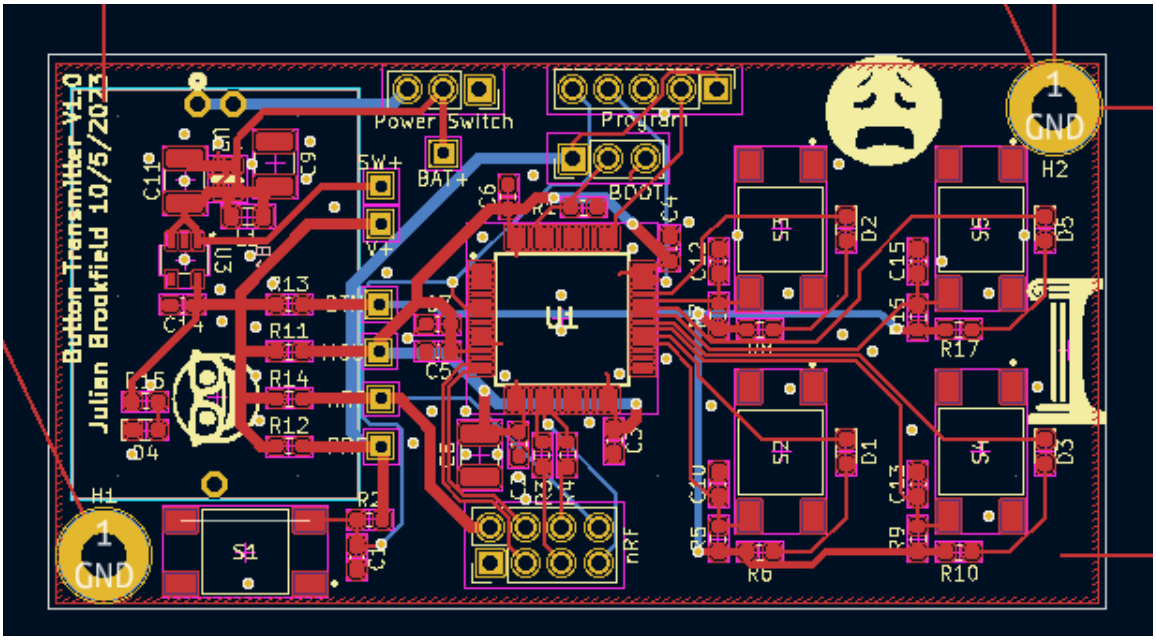


Figure 6: Button Transmitter PCB Layout

Final Implementation:

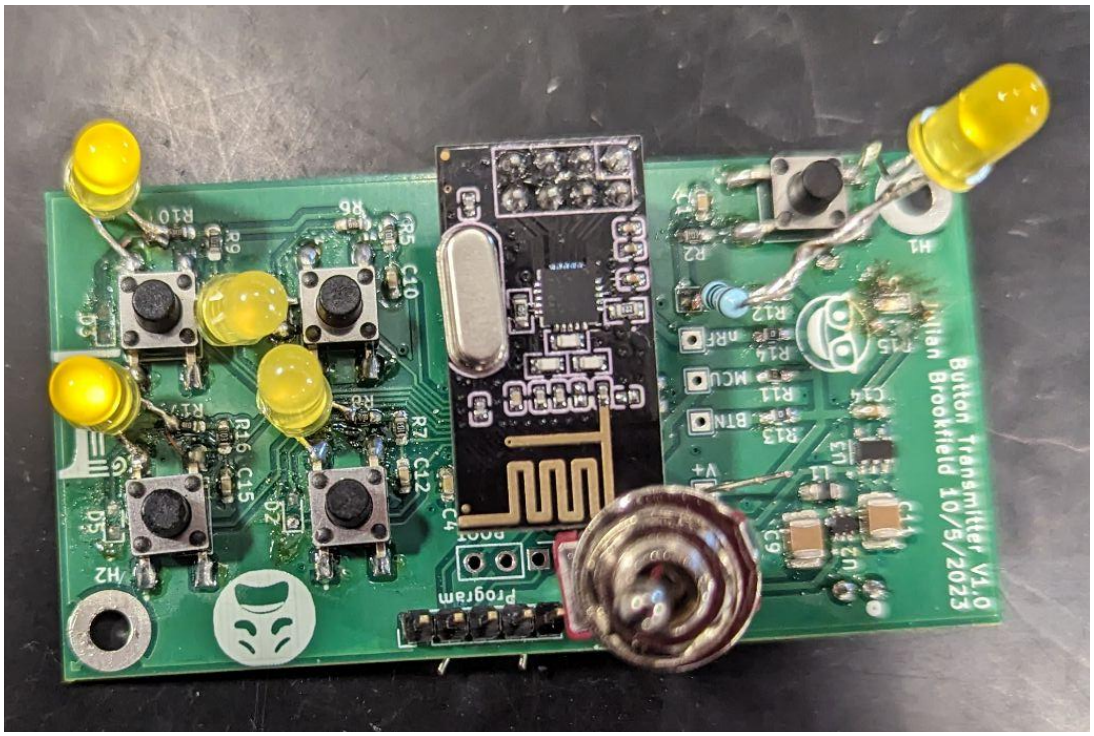


Figure 7: Final Assembled Button Transmitter PCB

Block Description:

The button transmitter is designed to remotely turn the wah effect on and off from the pickguard of the user's guitar. On the exterior of the package will be four buttons, each with a corresponding indicator light, which can control the routing of the audio signals in the floor pedal receiver. Our initial design only necessitates a single button to turn the effect on and off, but as a reach goal we plan on having two effects embedded within our pedal with increased routing flexibility. The buttons then can serve both as on/off switches, and as user-configurable "macros", which can turn on multiple effects with the same button, reverse effect order, etc.

Buttons that are mapped to be On/Off switches have their corresponding LEDs turn on and off to indicate their position. Buttons that are mapped to have a macro function will override the settings of the On/Off buttons and their LED status, turning them off and turning on the corresponding macro LED. This behavior is outlined in greater detail in the Software section. Buttons and their corresponding LEDs are connected to the microcontroller through GPIO pins.

The microcontroller and wireless transceiver used are the same STM32L053 and Nordic nRF24L01 parts from the headstock transmitter, which allows us to cut down on design complexity by reusing design elements. In the same manner as before, the button data is sent from the microcontroller to the wireless transceiver via SPI, and then wirelessly using GFSK. The power supply scheme is also the same, and thus will have the same testing requirements.

Pedal Receiver

Schematic:

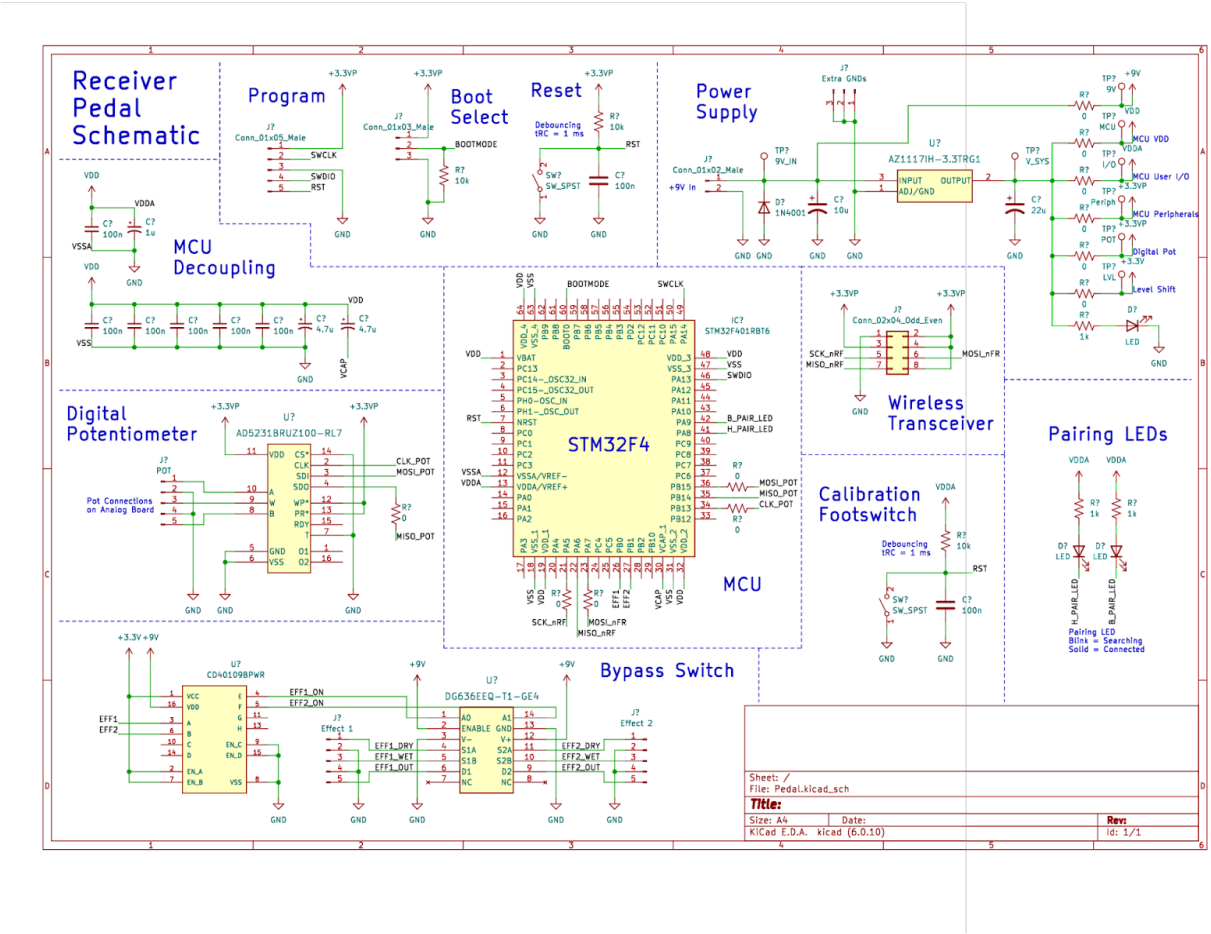


Figure 8: Pedal Receiver Schematic

Layout:

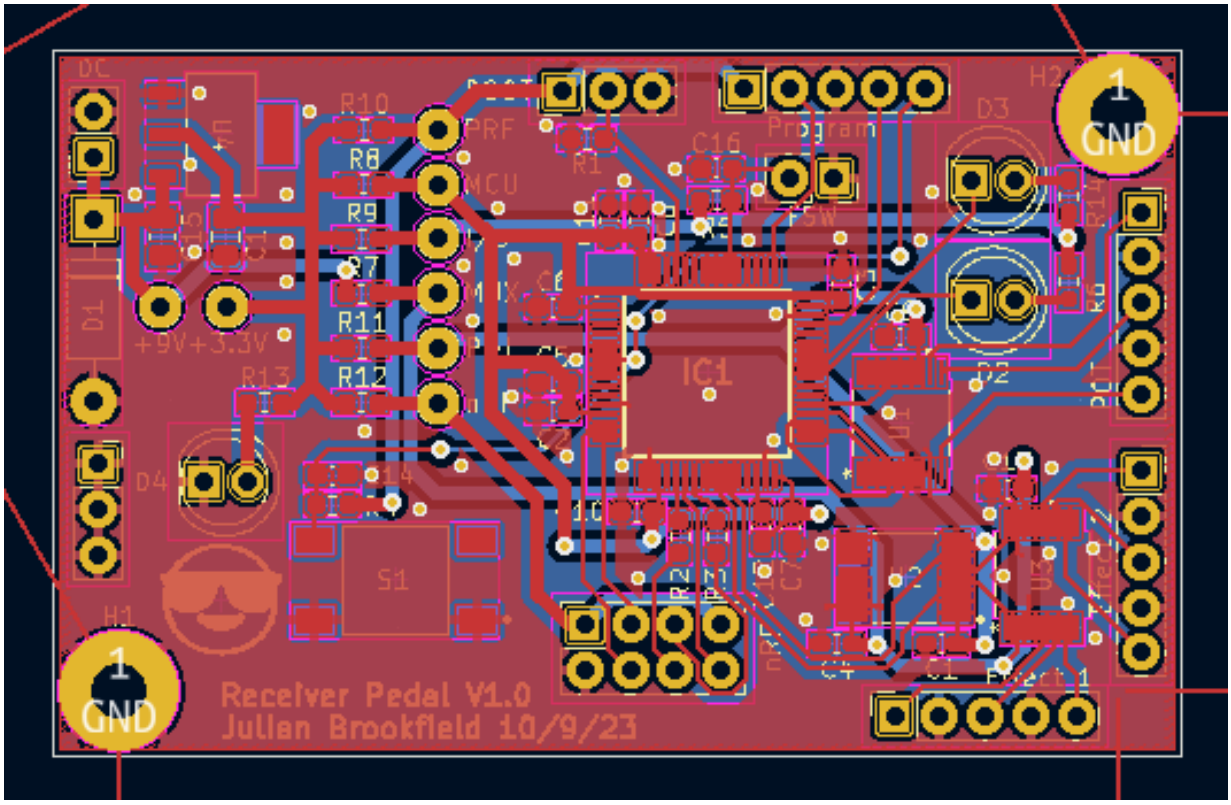


Figure 9: Receiver Board PCB Layout

Final Implementation:

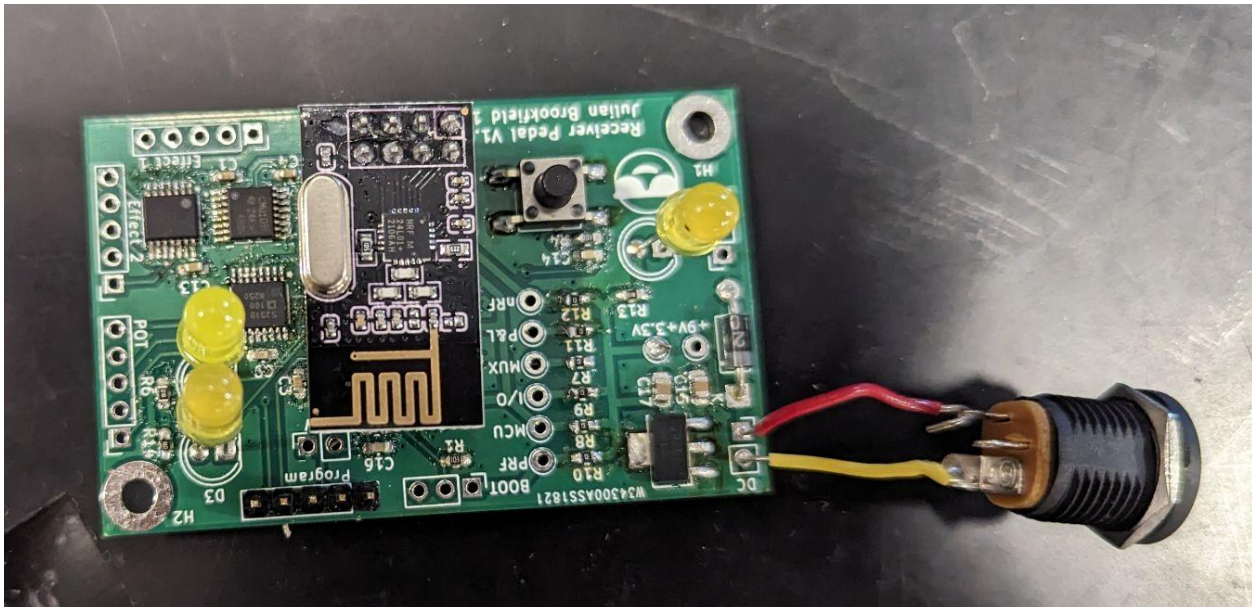


Figure 10: Final Assembled Receiver PCB

Block Description:

The control unit board for the floor pedal receiver is designed to control both the sweep of the filter in the audio board and the routing of audio signals from it. This starts with the wireless transceiver, another Nordic nRF24L01 module, which receives data from both of the transmitters and sends it to the microcontroller via SPI. A useful feature of the nRF module is that it can communicate with up to 6 devices concurrently [11], which is mostly designed for mesh network applications but allows us to have multiple transmitters connected to one receiver. Two pairing LEDs are included to indicate the connection status of the transmitters, with blinking indicating that connection is being established, and a continuous light indicating that connection has been achieved.

The microcontroller is an STM32F4 [17], which is conveniently available in the ECE Service Shop Inventory, and also has more computing power than the STM32L0s in the transmitters. This extra processing power is needed to convert the raw accelerometer data into positional data that can be used to generate a sweep of the digital pot, the algorithm for which is detailed further in the Software section. This also offloads most of the processing away from the transmitter MCUs, which allows them to run at much lower clock rates and thus conserve power, which is ideal for battery powered devices. To aid with the processing of the accelerometer data, a footswitch is included to set a calibration point for the headstock transmitter that corresponds to an initial position. This will also allow the user to choose the range of the sweep they wish to use, as well as the spatial position that range moves through, which is useful for accommodating the various playing positions that a user may prefer to use.

Once the accelerometer data is processed into a 10 bit position value, it is sent to the digital potentiometer [18] via SPI. We chose to have the digital pot located on the control board, despite taking the place of a pot located within the circuit of the audio board, in order to ensure signal integrity of the digital communication interface. Low frequency audio signals are much more tolerant of longer signal paths than high frequency serial interfaces, whose signal paths start to behave as transmission lines. Thus in order to keep the serial interface traces short and avoid the undesired effects that arise from this behavior, it was chosen to keep the digital pot close to the microcontroller on the control board. This will lead to slightly worse noise performance of the audio signal compared to the alternative, but this is much more desirable than a faulty digital interface, which can cease to function entirely.

Signal routing for the audio board is included on the control board for roughly the same reasons. Audio signals are routed through an analog mux that behaves as an SPDT switch [19], allowing either an unaffected “dry” signal or effected “wet” signal to

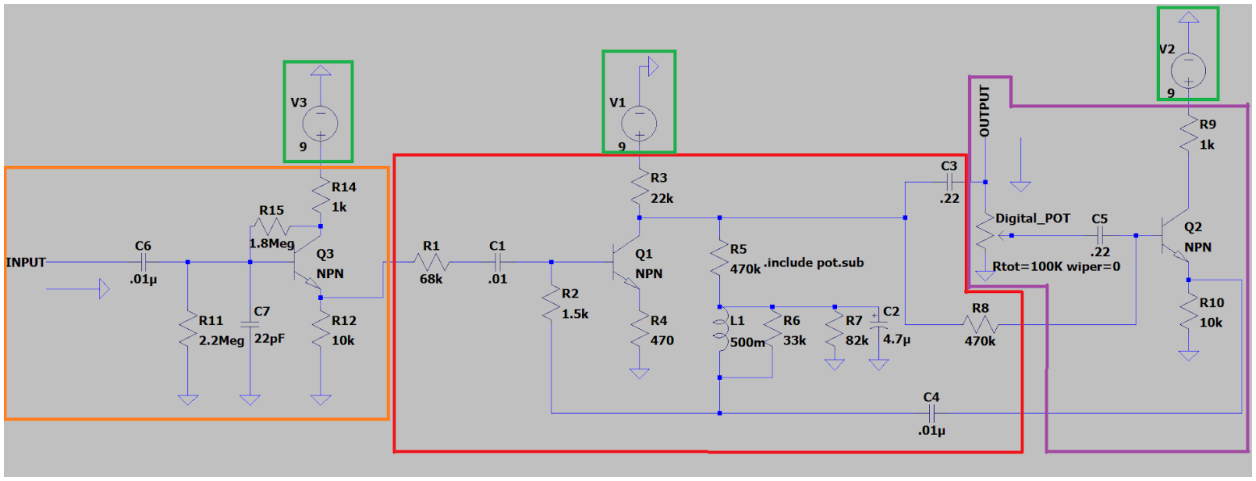
pass through. In the same manner as the buttons, switching for two audio effects is possible through the use of a dual analog mux, in the event we have time to implement a second effect. In order to interface between the 3.3 V logic of the microcontroller and the 9 V operation of the analog circuitry, a level shifter [20] is needed to interface with the analog mux.

The power for this board will be supplied externally from a 9 V DC wall adapter that is commonly used for guitar effects [13], which is already filtered and certified for sale as a commercial product. A reverse bias protection diode is included in parallel to the rest of the circuit to prevent damage from a user plugging in an incorrect supply. In such a scenario, this arrangement will destroy the diode, but using this arrangement also avoids the voltage drop associated with a protection diode in series during normal operation, allowing us to use the 9 V directly for the analog board. For the digital hardware, an LDO steps down the voltage from 9V to 3.3V.

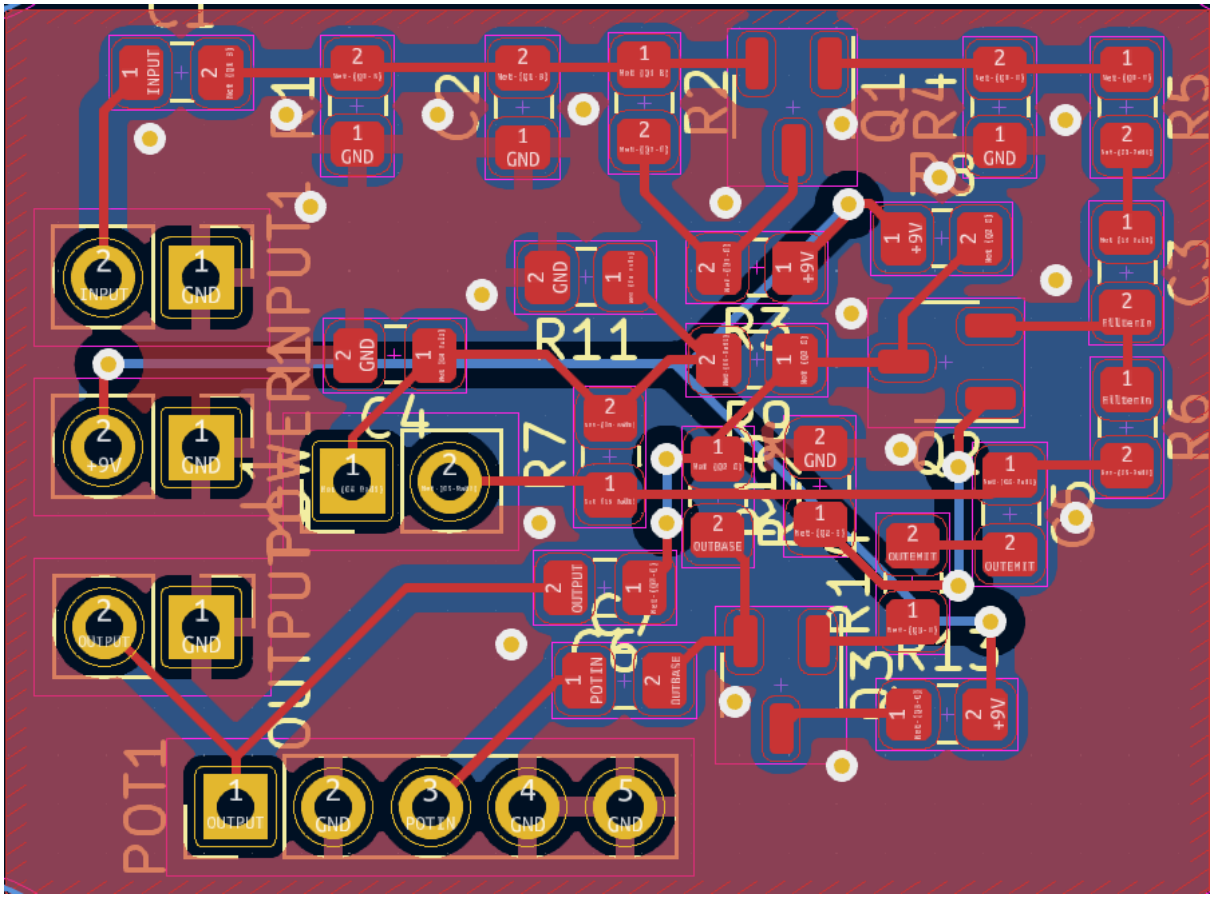
Audio/Wah Circuit

Schematic:

(a)



(b)



Blocks: Input Filter Output Power

Figure 11: Wah Effect Schematics
(a) LTSpice (b) KiCad PCB Layout

Final Implementation:

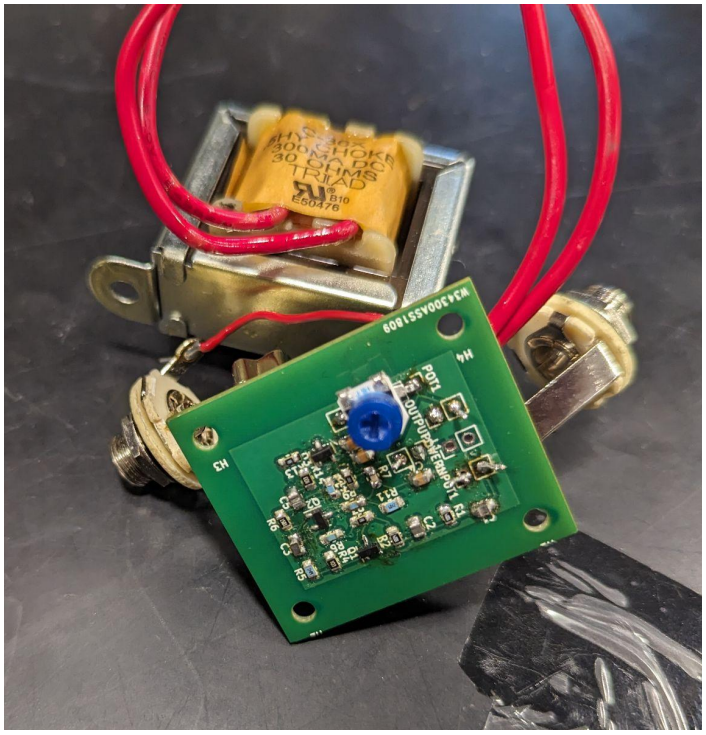


Figure 12: Final Assembled Audio PCB

Block Description:

The Power blocks represent the power to be supplied externally by the previously mentioned One Spot 9V DC wall adapter [13], as this adapter is already filtered and includes reverse-bias protection. Resistors connecting the power blocks to the other blocks aid in suppression oscillation.

The Input block contains a common-collector NPN buffer. To avoid higher-frequency signal loss, the input impedance consisting of R11 and R15 is orders of magnitude higher than the output impedance. The need for an input buffer is only a consideration if we do not implement a true bypass of the signal for the “off” mode, as is the case with the implementation of an analog mux for signal routing [4]. This prevents the wah circuit from loading the clean signal and cutting its tone.

The Filter block shows the layout for an active bandpass filter with a common-emitter amplifier. This filter is designed to boost a frequency range centering from roughly 450 Hz to 2 kHz, covering an acceptable range for a wah effect. The center of this range is to be controlled by a digital potentiometer (Digital_POT), which will connect to the Filter block at capacitor C3. The bandwidth of the boosted frequency range will also increase with frequency, meaning that the bandwidth will widen as the center frequency of the bandpass increases. This is to prevent intense resonance from the higher frequencies. Resistor R1 serves as the input impedance of

68kOhms. Capacitor C1 serves as a bypass between the input stage and the filter stage. The biasing of both the NPN BJT and the inductor will be handled by R3 and R7, which form a voltage divider. The most essential component of this block is the 500 mH inductor, as altering this value significantly will certainly have an effect on the boosted frequency range. The tolerance of this value will be explored in the “Tolerance Analysis” section.

The Output block features a low output impedance common collector NPN BJT, similar to the input buffer. The BJT is biased to buffer the signal from the digital potentiometer and control the effect of the potentiometer position on the volume level of the output.

Software

Accelerometer Pot Voltage:

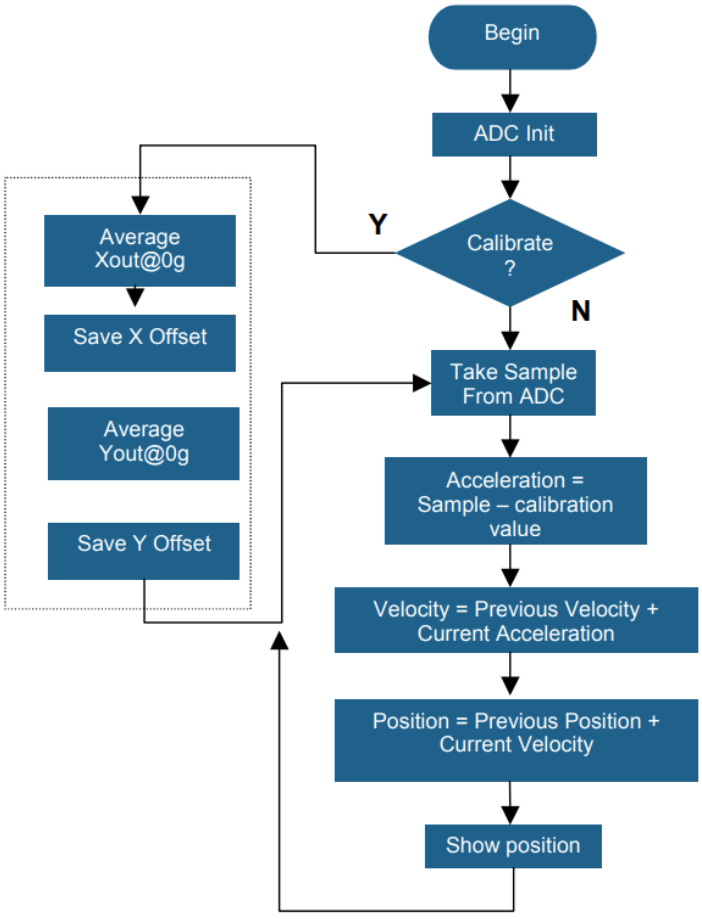


Figure 13: Sample accelerometer manipulation algorithm taken from NXP Semiconductor application note [22]

Our goal with using an accelerometer is to convert the accelerometer data into positional data which can then be normalized to fit within a 10 bit range to control the digital potentiometer. We planned on using this using an algorithm and code from an NXP semiconductor application note on the subject [22]. Our implementation will look much like the flowchart seen above, however we will have two calibration points to set a start and end position, which will allow the generation of a finite range of values. This range can then be normalized so that any distance covered by the headstock can be used to control the full sweep of the digital potentiometer.

Wireless Transmission:

In order to wirelessly transmit the sensor and button data we chose to use off-the-shelf nRF24L01 modules which are controlled with internal registers via SPI. Since these modules are primarily used with Arduinos, it was necessary for us to program our own drivers to interface with our STM32 microcontrollers. We achieved this using the native STM32 SPI functions and performing register manipulation according to the datasheet [10] and available online resources [6].

Results

Throughout the semester, we were able to achieve some modest results whilst facing a myriad of challenges, both technical and institutional. This primarily manifested in our struggle to get parts ordered through the business office, which ended up taking a month longer than expected and left us with one complete week to build and test our PCBs with all of our parts. However, despite the time crunch, we were able to make solid progress on our first revisions, as seen below:

Successes

We were able to fully assemble and program both of our transmitters successfully. While we weren't able to implement the receiver (discussed in the next section), we were able to program the nRF modules on the transmitters to successfully transmit data out while generating a log of its results, as seen in Figures 16 and 17.

We also achieved the correct behavior of the button LEDs on our button transmitter board, with one LED lighting up corresponding to the button pressed without overlap. Of note is that we intended to use SMD LEDs which turned out to be

defective, thus we had to bodge in through-hole LEDs to demonstrate their functionality.

```
// write a single byte to the particular register
void nrf24_WriteReg (uint8_t Reg, uint8_t Data)
{
    uint8_t buf[2];
    buf[0] = Reg|1<<5;
    buf[1] = Data;

    // Pull the CS Pin LOW to select the device
    CS_Select();

    HAL_SPI_Transmit(NRF24_SPI, buf, 2, 1000);

    // Pull the CS HIGH to release the device
    CS_UnSelect();
}

//write multiple bytes starting from a particular register
void nrf24_WriteRegMulti (uint8_t Reg, uint8_t *data, int size)
{
    uint8_t buf[2];
    buf[0] = Reg|1<<5;
//  buf[1] = Data;

    // Pull the CS Pin LOW to select the device
    CS_Select();

    HAL_SPI_Transmit(NRF24_SPI, buf, 1, 100);
    HAL_SPI_Transmit(NRF24_SPI, data, size, 1000);

    // Pull the CS HIGH to release the device
    CS_UnSelect();
}
```

```
uint8_t nrf24_ReadReg (uint8_t Reg)
{
    uint8_t data=0xff;

    // Pull the CS Pin LOW to select the device
    CS_Select();

    HAL_SPI_Transmit(NRF24_SPI, &Reg, 1, 100);
    HAL_SPI_Receive(NRF24_SPI, &data, 1, 100);

    // Pull the CS HIGH to release the device
    CS_UnSelect();

    return data;
}


/* Read multiple bytes from the register */
void nrf24_ReadReg_Multi (uint8_t Reg, uint8_t *data, int size)
{
    // Pull the CS Pin LOW to select the device
    CS_Select();

    HAL_SPI_Transmit(NRF24_SPI, &Reg, 1, 100);
    HAL_SPI_Receive(NRF24_SPI, data, size, 1000);

    // Pull the CS HIGH to release the device
    CS_UnSelect();
}
```

Figure 14: Snapshot of Homemade SPI Drivers

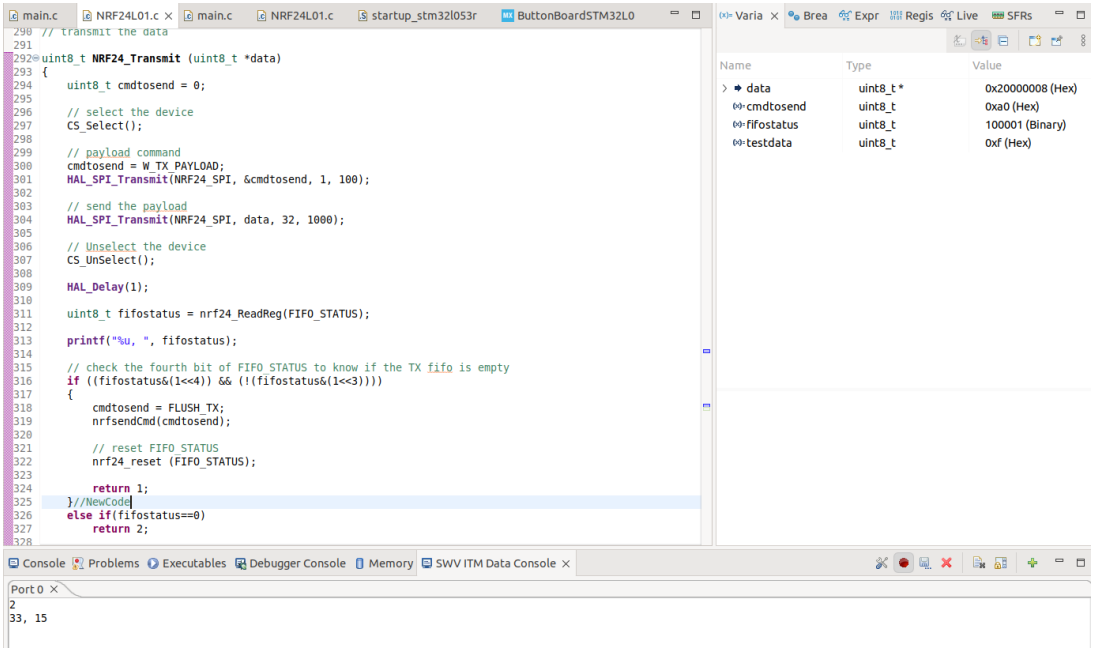


Figure 15: nRF Console Log

Our boost converters are able to successfully step up the 3 V from our coin cell batteries to the 3.3V required for digital logic without any spurious emissions or switching frequencies bleeding into the power rails.

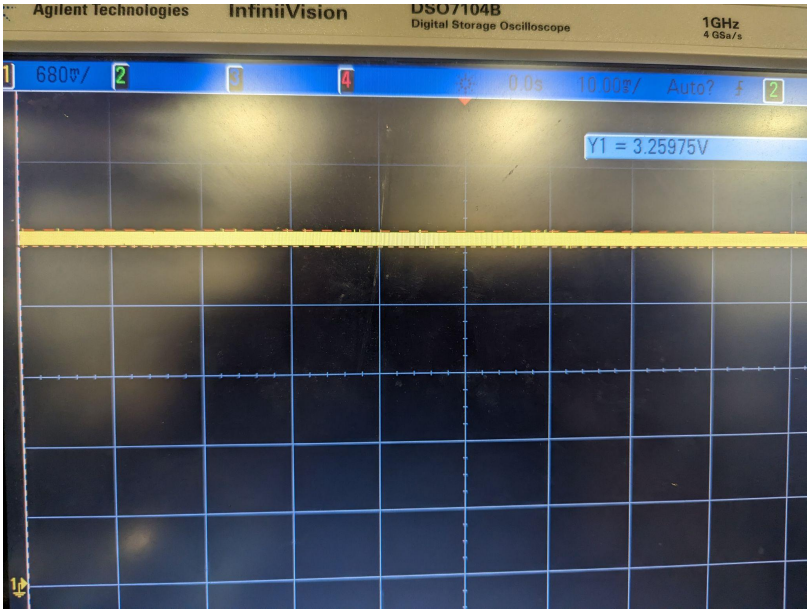


Figure 19: Steady 3.3V Output From Boost Converter

Our accelerometer also correctly outputs a voltage swing of 1.2 V, which is greater than the minimum 990 mV required to achieve proper fidelity across the sweep of our digital pot.

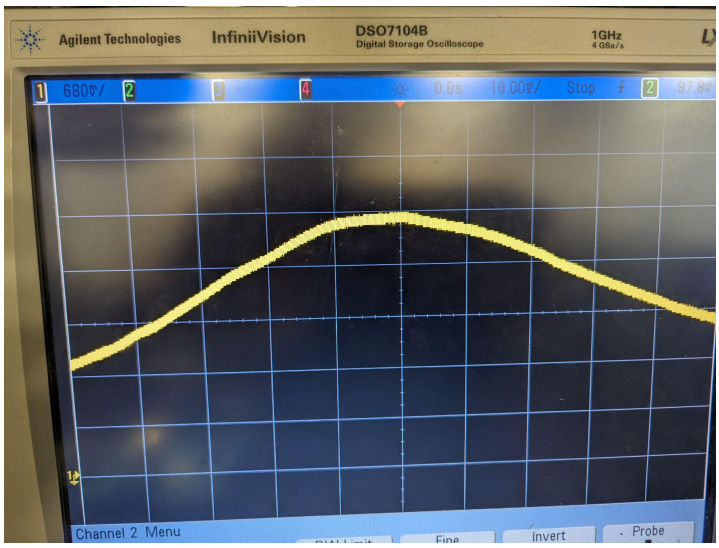
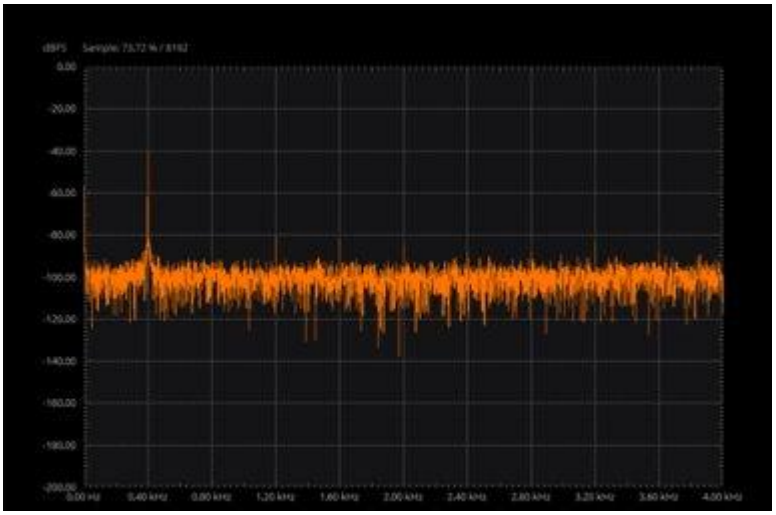


Figure 20: Maximum 1.2V Voltage Swing From Accelerometer

The analog audio circuit passed a signal from input to output, and connecting a traditional potentiometer changes the highlighted frequency of the bandpass filter.

(a) 

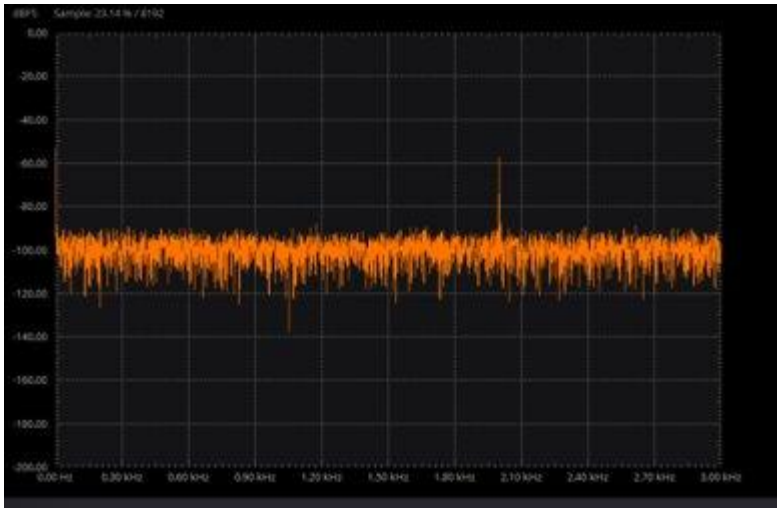
(b) 

Figure 21: Analog Filter Boosted Frequency
(a) Potentiometer at maximum value (b) Potentiometer at minimum value

Challenges

Despite our progress, we were unable to fully integrate our PCBs due to a number of issues, mostly stemming from the time constraints we faced. The largest setback we encountered was the inability for our receiver microcontroller to be programmed. We attempted numerous fixes to remedy the issue, such as replacing the MCU three times and repeatedly checking for signal continuity, but were ultimately unable to get any programs to load onto it. Our best hypothesis for this has to do with the BOOTMODE pins that control the start-up behavior on STM32 microcontrollers. Most STM32s only require a single boot pin to be configured, but can have two BOOTMODE pins on higher- performance chips such as the one used in our receiver. We neglected to connect the second boot pin assuming it wasn't necessary, but given that our lower-performance MCUs on the transmitters without the need for a second boot pin could be programmed without issue, this is perhaps a misguided assumption to make. As a result, we were unable to verify the functionality of the other components on the receiver PCB such as the signal routing and the sweep of the digital potentiometer.

Another consequence of our limited time was being unable to program the nRF module to work as a receiver. Writing and debugging the drivers for the nRF module from scratch took a larger amount of time to implement than we anticipated, and thus we simply couldn't work fast enough amidst the other debugging work we performed.

Given another week of time, we most likely could have written the receiver drivers and been able to establish a wireless connection between two transceivers.

Finally, we faced issues with our audio PCB. When the audio board is powered, the guitar signal is defeated and no sound gets through. However, when unpowered, the guitar signal can pass through and does exhibit some of the bandpass filtering effect from the circuit, albeit with significant attenuation. After probing for continuity and tracing the signal path with an oscilloscope, the most likely explanation comes down to an error in the PCB layout which results in two nets being connected which shouldn't, and thus a board revision is likely necessary to remedy this problem.

Cost

Labor:

Using the model provided to us from the course website, we estimate the cost of labor at $40/hr, 8 hours per week, for 16 weeks of class to be

$40/hr/person * 8 hr/wk * 16 wk * 3 people * 2.5 overhead = $38,400

Parts:

The bill of materials and its associated cost is detailed in Table 5 in the appendix, the total of which adds to $86.69. We can round this number up to $100 to account for reordering broken parts and other minor unaccounted expenses.

Total Cost = **$38,500**

Conclusion

While we weren't successful in creating a fully functional final demo, we are nevertheless proud of what we were able to accomplish given our circumstances. All three of us came into this project with relatively little background experience in the various disciplines required to create a fully integrated final project from the ground up, and we were still able to create two functional boards. All of our progress can be thought of as an initial prototype, the likes of which very rarely work on their first attempt without a revision or two to work out the issues. In this context, we're happy with what we could get done and are optimistic about having a fully functioning prototype with a few more weeks of effort.

In keeping with ethical guidelines outlined by IEEE [1], consideration for the safety of the end user is our highest priority. We have made design choices when possible to minimize any risk involved in the operation of our project. For powering our subsystems, we chose to use standardized and regulated power supplies, be it our voice of 9V wall supply [13] or our choice to use coin cells instead of dealing with the risk inherent with a rechargeable Li-Ion battery supply. Ethically, as far as the environment is concerned, a rechargeable power supply solution would be advisable in the future, but given our very limited development time we felt it safest to use a power source that is readily available and minimizes risk. Our choice of RF modules are also in compliance with FCC regulations on ISM band devices [12]. Finally, we will ensure the quality of our designs through seeking criticism and advice not only through mandatory design reviews, but by seeking the guidance and advice of others with experience in the scope of our project.

Citations

[1] IEEE, “IEEE Code of Ethics,” *ieee.org*, 2020. https://www.ieee.org/about/corporate/governance/p7-8.html

[2] “RANGE OF EXPRESSION: FIND YOUR CRY BABY® WAH - Lifestyle - Dunlop,” Oct. 08, 2021. https://lifestyle.jimdunlop.com/find-your-cry-baby-wah/ (accessed Sep. 26, 2023).

[3] “Critical Bandwidths and Just-Noticeable Differences,” *www.phys.uconn.edu*. https://www.phys.uconn.edu/~gibson/Notes/Section7_2/Sec7_2.htm#:~:text=Between%20these%20two%20mechanisms%2C%20the (accessed Sep. 29, 2023).

[4] “The Technology of Wah Pedals,” *www.geofex.com*. http://www.geofex.com/article_folders/wahpedl/wahped.htm

[5] R. Foote, M. Zhang, T. Macdonald, and H. Shao, “ECE 445 SENIOR DESIGN LABORATORY Musical Hand Team #24,” 2022. Accessed: Sep. 29, 2023. [Online]. Available: https://courses.engr.illinois.edu/ece445/getfile.asp?id=20496

[6] “How to Write Basic Library for NRF24L01 PART 1 || Common configuration || STM32 SPI,” *www.youtube.com*. https://www.youtube.com/watch?v=mB7LsiscM78 (accessed Sep. 29, 2023).

[7] “UM1724 User manual,” *ST,* 2020. Accessed: Sep. 29, 2023. [Online]. Available: https://www.st.com/resource/en/user_manual/um1724-stm32-nucleo64-boards-mb1136-stmicroelectronics.pdf

[8] “STM32L053C6 STM32L053C8 STM32L053R6 STM32L053R8.” *ST*. 2020. Accessed: Sep. 29, 2023. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32l053c6.pdf

[9] "nRF24 Series - Nordic Semiconductor," *www.nordicsemi.com*. https://www.nordicsemi.com/products/nrf24-series

[10] "nRF24L01 Single Chip 2.4GHz Transceiver Product Specification," *Mouser*, 2007. https://www.mouser.com/datasheet/2/297/nRF24L01_Product_Specification_v2_0-9199.pdf (accessed Sep. 28, 2023).

[11] "nRF24L01 Wireless RF Module," *Components101*. https://components101.com/wireless/nrf24l01-pinout-features-datasheet

[12] "Regulatory and Compliance Standards for RF Devices White Paper," 2007. Accessed: Sep. 29, 2023. [Online]. Available: https://infocenter.nordicsemi.com/pdf/nwp_010.pdf

[13] "1 SPOT® – Truetone." https://truetone.com/1-spot/ (accessed Sep. 29, 2023).

[14] "TPS6112x Synchronous Boost Converter With 1.1-A Switch and Integrated LDO," *Texas Instruments*, 2015. Accessed: Sep. 29, 2023. [Online]. Available: https://www.ti.com/lit/ds/symlink/tps61121.pdf?ts=1695846105933&ref_url=https%253A%252F%252Fgoogle.com

[15] "LOW DROPOUT LINEAR REGULATOR WITH INDUSTRIAL TEMPERATURE RANGE," *Diodes Incorporated*, 2002. https://www.diodes.com/assets/Datasheets/AZ1117I.pdf (accessed Sep. 28, 2023).

[16] " Five Steps to a Good PCB Layout of a Boost Converter User's Guide Five Steps to a Good PCB Layout of a Boost Converter," *Texas Instruments*, 2016. Accessed: Sep. 29, 2023. [Online]. Available: https://www.ti.com/lit/an/slva773/slva773.pdf?ts=1695894579968

[17] "STM32F401xB STM32F401xC," 2019. Accessed: Sep. 29, 2023. [Online]. Available: https://www.st.com/content/ccc/resource/technical/document/datasheet/9e/50/b1/5a/5f/ae/4d/c1/DM00086815.pdf/files/DM00086815.pdf/jcr:content/translations/en.DM00086815.pdf

[18] "Nonvolatile Memory, 1024-Position Digital Potentiometer." Accessed: Sep. 29, 2023. [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/AD5231.pdf

[19] "Vishay Siliconix." Accessed: Sep. 29, 2023. [Online]. Available: https://www.vishay.com/docs/75621/dg636e.pdf

[20]"CMOS Quad Low-to-High Voltage Level Shifter," *Texas Instruments*, 2003. Accessed: Sep. 29, 2023. [Online]. Available: https://www.ti.com/lit/ds/symlink/cd40109b.pdf

[21] "ADXL335," *Analog Devices* , 2009. https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL335.pdf (accessed Sep. 28, 2023).

[22] K. Seifert and O. Camacho, "Implementing Positioning Algorithms Using Accelerometers," 2007. Available: https://www.nxp.com/docs/en/application-note/AN3397.pdf

[23] "Axial Shielded Power Chokes," *Allied Components International*, Nov. 12, 2020. https://www.alliedcomponents.com/storage/through_hole_inductors_chokes/pdfs/aspc80.pdf (accessed Sep. 28, 2023).

Appendix

• Accelerometer axes output minumum ± 990 mV from 1.5 V bias point through range of motion	1. Connect accelerometer supply rail and ground to + 3.3 V external supply 2. Connect accelerometer axes test points to oscilloscope 3. Record output voltage reading from the accelerometer while moving it at the maximum reasonable speed the device will be used, and verify reading is within acceptable boundaries.
• Output of power supply maintains 3.3V across maximum load of 150 mA within ± 5%, including both voltage sag and power supply ripple	1. Plug in coin cell battery, or attach 3V supply to the input test point of boost converter 2. With all internal power rails disconnected, attach 47 Ohm resistor to LDO output test point 3. Measure output voltage across load across resistor and verify it remains within ± 5% of 3.3 V

Table 1: Headstock Transmitter Requirements & Verifications

• The corresponding LEDs for each button behave according to our specification, as highlighted above and in the Software section	1. Plug in coin cell battery, or attach 3.3V supply to the V_SYS test point 2. Press buttons in various sequences and verify correct LED & signal behavior is being observed

• Output of power supply maintains 3.3V across maximum load of 70 mA within ± 5%, including both voltage sag and power supply ripple	1. Plug in coin cell battery, or attach 3V supply to the input test point of boost converter 2. With all internal power rails disconnected, attach 47 Ohm resistor to LDO output test point 3. Measure output voltage across load across resistor and verify it remains within ± 5% of 3.3 V

Table 2: Button Transmitter Requirements & Verifications

• Receiver pedal is able to receive data from both transmitters simultaneously	1. Power on all transmitters and receiver 2. Pair both transmitters to receiver, verify that indicator lights display correct status 3. Test that button data and accelerometer data is correct using procedures below
• Digital potentiometer implements a voltage divider which is able to output a voltage from 0 V to 9 V, ± 5%	1. Power on board with 9V supply 2. Connect outer pins of pot output connector to +9V and GND, and center pin to a multimeter/ oscilloscope 3. Create test program for microcontroller which increases the number of taps slowly between 0 - 1023 4. Measure output voltage and verify that it sweeps from rail to rail, 0 V to 9 V.

• Microcontroller maps accelerometer data to a 10 bit position value which is able to sweep the digital potentiometer from 0V to 9V, ΔV ± 5%	1. Power on board with 9V supply 2. Connect outer pins of pot output connector to +9V and GND, and center pin to a multimeter/ oscilloscope 3. Pair headstock transmitter to receiver pedal and load accelerometer data processor program 4. Move headstock transmitter up and down and record potentiometer output voltage, verifying that it sweeps between 0 V and 9V rail to rail.
• Button is able to select between dry and wet audio signals, with attenuation of less than 3 dB	1. Turn on and pair button transmitter and pedal receiver 2. Connect inputs of analog mux pins to two voltage sources outputting sine tones of different frequencies 3. Load button control program to microcontroller 4. Verify that the frequency of dry input is present at the output with button off, and wet input frequency is present at the output with button on 5. Connect analog mux output to oscilloscope and verify that it is attenuated less than 3 dB from the input signal amplitude
• Footswitch calibrates	1. Turn on and pair headstock transmitter and pedal receiver, connect digital pot leads to their

	respective outputs 2. Run accelerometer data processor program and verify that it works as previously stated 3. Move headstock transmitter to different starting location, press footswitch, and again move transmitter up and down and verify that the digital pot outputs a voltage sweep between 0 V and 9 V
• Power supply can accurately supply 9 V output up to 50 mA with ΔV ± 1% and 3.3 V at 100 mA with ΔV ± 1% simultaneously	1. Disconnect all internal PCB power rails 2. Connect 9V external power supply 3. Connect ~180 Ohm resistor to 9 V output test point, and measure voltage across resistor, verifying it remains within 9 V ± 1% 4. Connect ~33 Ohm resistor to 3.3 V output test point, and measure voltage across resistor, verifying it remains within 3.3V ± 1% 5. With both loads connected, measure voltages across each load and verify they both still operate within specified range

Table 3: Pedal Receiver Requirements & Verifications

• Circuit sweeps frequencies from lower-mid to upper-mid range (400Hz to 2.5kHz) with 15-20 dB amplification.	1. Connect input of wah circuit to external AC power supply 2. Connect output to oscilloscope

	3. Create test program for microcontroller which sets the number of taps for the digital potentiometer to 0 or 1023 4. Perform a spectral analysis on the oscilloscope with the number of taps at maximum. Note the peak and -3dB range for the minimum frequency boost. 5. Perform a spectral analysis on the oscilloscope with the number of taps at zero. Note the peak and -3dB range.
● Filter voltage gain must be between 1dB to 19dB	1. Connect input of wah circuit to external AC power supply 2. Connect voltmeter to filter block input and ground 3. Connect output to oscilloscope 4. Create test program for microcontroller which sets the number of taps for the digital potentiometer to 0 or 1023 5. Measure output voltage for each potentiometer setting

Table 4: Audio Board Requirements & Verifications

Name/Disc.	Manufacturer	Digikey Part #	Quantity	Cost
Transmitter MCUs	ST Microelectronics	497-19667-1-ND	2	$11.24
Receiver MCU	ST Microelectronics	497-17428-ND	1	$6.40
Accelerometer	Analog Devices	505-ADXL335BCPZ-ND	1	$8.36
Analog Mux	Vishay Siliconix	DG636EEQ-T1-GE4CT-ND	1	$1.54
Digital Pot	Analog Devices	505-AD5231BRUZ100-RL7CT-ND	1	$6.61
Level Shifter	Texas Instruments	296-12163-1-ND	1	$0.61
Boost Converter	Texas Instruments	296-41854-1-ND	2	$2.44
3.3V-3.3V LDO	Diodes Incorporated	AP2112K-3.3TRG1DITR-ND	2	$0.70
9V-3.3V LDO	Diodes Incorporated	AZ1117IH-3.3TRG1DICT-ND	1	$0.38
Battery	Panasonic	P189-ND	2	$0.88
Battery Holder	MPD	BU2032-1-HD-G-ND	2	$2.46
Transistors	OnSemi	BC846BLT1GOSCT-ND	3	$0.42
Large Inductor	Allied Components	3475-ASPC80-564K-RC-ND	1	$4.25
Mechanical Pot (Testing)	TT Electronics	987-1723-ND	1	$1.55
Assorted SMD passives	Various	Various	~100	~$20

resistors/ capacitors/ inductors/ LEDs				
5 Pin Connector	Sullins Connector Solutions	S6103-ND	7	$3.36
3 Pin Connector	Sullins Connector Solutions	S7036-ND	9	$3.33
Buttons	CW Industries	2223-TS04-66-70-BK-100-SMT-ND	10	$1.75
Audio Jacks	Neutrik	Not On Digikey	4	$6.36
Footswitch	N/A	Not On Digikey	1	$2.55
Enclosures	3D Print	N/A	3	~$5
				$86.69

Table 5: Bill of Materials

NRF24L01.c

```
#include "stm32l0xx.h"
#include "stm32l0xx_hal_conf.h"
#include "NRF24L01.h"
extern SPI_HandleTypeDef hspi1;
#define NRF24_SPI &hspi1
#define NRF24_CE_PORT   GPIOA
#define NRF24_CE_PIN    GPIO_PIN_1
#define NRF24_CSN_PORT   GPIOA
#define NRF24_CSN_PIN    GPIO_PIN_2
void CS_Select (void)
{
      HAL_GPIO_WritePin(NRF24_CSN_PORT, NRF24_CSN_PIN, GPIO_PIN_RESET);
}
void CS_UnSelect (void)
{
      HAL_GPIO_WritePin(NRF24_CSN_PORT, NRF24_CSN_PIN, GPIO_PIN_SET);
}
void CE_Enable (void)
{
```

```
	HAL_GPIO_WritePin(NRF24_CE_PORT, NRF24_CE_PIN, GPIO_PIN_SET);
}
void CE_Disable (void)
{
	HAL_GPIO_WritePin(NRF24_CE_PORT, NRF24_CE_PIN, GPIO_PIN_RESET);
}
// write a single byte to the particular register
void nrf24_WriteReg (uint8_t Reg, uint8_t Data)
{
	uint8_t buf[2];
	buf[0] = Reg|1<<5;
	buf[1] = Data;
	// Pull the CS Pin LOW to select the device
	CS_Select();
	HAL_SPI_Transmit(NRF24_SPI, buf, 2, 1000);
	// Pull the CS HIGH to release the device
	CS_UnSelect();
}
//write multiple bytes starting from a particular register
void nrf24_WriteRegMulti (uint8_t Reg, uint8_t *data, int size)
{
	uint8_t buf[2];
	buf[0] = Reg|1<<5;
//	buf[1] = Data;
	// Pull the CS Pin LOW to select the device
	CS_Select();
	HAL_SPI_Transmit(NRF24_SPI, buf, 1, 100);
	HAL_SPI_Transmit(NRF24_SPI, data, size, 1000);
	// Pull the CS HIGH to release the device
	CS_UnSelect();
}
uint8_t nrf24_ReadReg (uint8_t Reg)
{
	uint8_t data=0xff;
	// Pull the CS Pin LOW to select the device
	CS_Select();
	HAL_SPI_Transmit(NRF24_SPI, &Reg, 1, 100);
	HAL_SPI_Receive(NRF24_SPI, &data, 1, 100);
	// Pull the CS HIGH to release the device
	CS_UnSelect();
	return data;
}
/* Read multiple bytes from the register */
void nrf24_ReadReg_Multi (uint8_t Reg, uint8_t *data, int size)
{
	// Pull the CS Pin LOW to select the device
	CS_Select();
	HAL_SPI_Transmit(NRF24_SPI, &Reg, 1, 100);
	HAL_SPI_Receive(NRF24_SPI, data, size, 1000);
```

```
	// Pull the CS HIGH to release the device
	CS_UnSelect();
}
// send the command to the NRF
void nrfsendCmd (uint8_t cmd)
{
	// Pull the CS Pin LOW to select the device
	CS_Select();
	HAL_SPI_Transmit(NRF24_SPI, &cmd, 1, 100);
	// Pull the CS HIGH to release the device
	CS_UnSelect();
}
void nrf24_reset(uint8_t REG)
{
	if (REG == STATUS)
	{
		nrf24_WriteReg(STATUS, 0x00);
	}
	else if (REG == FIFO_STATUS)
	{
		nrf24_WriteReg(FIFO_STATUS, 0x11);
	}
	else {
	nrf24_WriteReg(CONFIG, 0x08);
	nrf24_WriteReg(EN_AA, 0x3F);
	nrf24_WriteReg(EN_RXADDR, 0x03);
	nrf24_WriteReg(SETUP_AW, 0x03);
	nrf24_WriteReg(SETUP_RETR, 0x03);
	nrf24_WriteReg(RF_CH, 0x02);
	nrf24_WriteReg(RF_SETUP, 0x0E);
	nrf24_WriteReg(STATUS, 0x00);
	nrf24_WriteReg(OBSERVE_TX, 0x00);
	nrf24_WriteReg(CD, 0x00);
	uint8_t rx_addr_p0_def[5] = {0xE7, 0xE7, 0xE7, 0xE7, 0xE7};
	nrf24_WriteRegMulti(RX_ADDR_P0, rx_addr_p0_def, 5);
	uint8_t rx_addr_p1_def[5] = {0xC2, 0xC2, 0xC2, 0xC2, 0xC2};
	nrf24_WriteRegMulti(RX_ADDR_P1, rx_addr_p1_def, 5);
	nrf24_WriteReg(RX_ADDR_P2, 0xC3);
	nrf24_WriteReg(RX_ADDR_P3, 0xC4);
	nrf24_WriteReg(RX_ADDR_P4, 0xC5);
	nrf24_WriteReg(RX_ADDR_P5, 0xC6);
	uint8_t tx_addr_def[5] = {0xE7, 0xE7, 0xE7, 0xE7, 0xE7};
	nrf24_WriteRegMulti(TX_ADDR, tx_addr_def, 5);
	nrf24_WriteReg(RX_PW_P0, 0);
	nrf24_WriteReg(RX_PW_P1, 0);
	nrf24_WriteReg(RX_PW_P2, 0);
	nrf24_WriteReg(RX_PW_P3, 0);
	nrf24_WriteReg(RX_PW_P4, 0);
	nrf24_WriteReg(RX_PW_P5, 0);
```

```
		nrf24_WriteReg(FIFO_STATUS, 0x11);
		nrf24_WriteReg(DYNPD, 0);
		nrf24_WriteReg(FEATURE, 0);
		}
}
void NRF24_Init (void)
{
		// disable the chip before configuring the device
		CE_Disable();
		// reset everything
		nrf24_reset (0);
		nrf24_WriteReg(CONFIG, 0);  // will be configured later
		nrf24_WriteReg(EN_AA, 0);  // No Auto ACK
		nrf24_WriteReg (EN_RXADDR, 0);  // Not Enabling any data pipe right now
		nrf24_WriteReg (SETUP_AW, 0x03);  // 5 Bytes for the TX/RX address
		nrf24_WriteReg (SETUP_RETR, 0);   // No retransmission
		nrf24_WriteReg (RF_CH, 0);  // will be setup during Tx or RX
		nrf24_WriteReg (RF_SETUP, 0x0E);   // Power= 0db, data rate = 2Mbps
		// Enable the chip after configuring the device
		CE_Enable();
}
// set up the Tx mode
void NRF24_TxMode (uint8_t *Address, uint8_t channel)
{
		// disable the chip before configuring the device
		CE_Disable();
		nrf24_WriteReg (RF_CH, channel);  // select the channel
		nrf24_WriteRegMulti(TX_ADDR, Address, 5);  // Write the TX address
		// power up the device
		uint8_t config = nrf24_ReadReg(CONFIG);
		config = config | (1<<1);   // write 1 in the PWR_UP bit
//		config = config & (0xF2);    // write 0 in the PRIM_RX, and 1 in the
PWR_UP, and all other bits are masked
		nrf24_WriteReg (CONFIG, config);
		// Enable the chip after configuring the device
		CE_Enable();
}
// transmit the data
uint8_t NRF24_Transmit (uint8_t *data)
{
		uint8_t cmdtosend = 0;
		// select the device
		CS_Select();
		// payload command
		cmdtosend = W_TX_PAYLOAD;
		HAL_SPI_Transmit(NRF24_SPI, &cmdtosend, 1, 100);
		// send the payload
		HAL_SPI_Transmit(NRF24_SPI, data, 32, 1000);
		// Unselect the device
```

```
	CS_UnSelect();
	HAL_Delay(1);
	uint8_t fifostatus = nrf24_ReadReg(FIFO_STATUS);
	// check the fourth bit of FIFO_STATUS to know if the TX fifo is empty
	if ((fifostatus&(1<<4)) && (!(fifostatus&(1<<3))))
	{
		cmdtosend = FLUSH_TX;
		nrfsendCmd(cmdtosend);
		// reset FIFO_STATUS
		nrf24_reset (FIFO_STATUS);
		return 1;
	}
	else if(fifostatus==0) //if status is all zeros then there must be an
error
		return 2;
	return 0;
}
void NRF24_RxMode (uint8_t *Address, uint8_t channel)
{
	// disable the chip before configuring the device
	CE_Disable();
	nrf24_reset (STATUS);
	nrf24_WriteReg (RF_CH, channel);  // select the channel
	// select data pipe 2
	uint8_t en_rxaddr = nrf24_ReadReg(EN_RXADDR);
	en_rxaddr = en_rxaddr | (1<<2);
	nrf24_WriteReg (EN_RXADDR, en_rxaddr);
	/* We must write the address for Data Pipe 1, if we want to use any pipe
from 2 to 5
	 * The Address from DATA Pipe 2 to Data Pipe 5 differs only in the LSB
	 * Their 4 MSB Bytes will still be same as Data Pipe 1
	 *
	 * For Eg->
	 * Pipe 1 ADDR = 0xAABBCCDD11
	 * Pipe 2 ADDR = 0xAABBCCDD22
	 * Pipe 3 ADDR = 0xAABBCCDD33
	 *
	 */
	nrf24_WriteRegMulti(RX_ADDR_P1, Address, 5);  // Write the Pipe1 address
	nrf24_WriteReg(RX_ADDR_P2, 0xEE);  // Write the Pipe2 LSB address
	nrf24_WriteReg (RX_PW_P2, 32);   // 32 byte payload size for pipe 2
	// power up the device in Rx mode
	uint8_t config = nrf24_ReadReg(CONFIG);
	config = config | (1<<1) | (1<<0);
	nrf24_WriteReg (CONFIG, config);
	// Enable the chip after configuring the device
	CE_Enable();
}
uint8_t isDataAvailable (int pipenum)
```

```
{
	CS_Select();
	uint8_t status = nrf24_ReadReg(STATUS);
	if ((status&(1<<6))&&(status&(pipenum<<1)))
	{
		nrf24_WriteReg(STATUS, (1<<6));
		CS_UnSelect();
		return 1;
	}
	CS_UnSelect();
	return 0;
}
void NRF24_Receive (uint8_t *data)
{
	uint8_t cmdtosend = 0;
	// select the device
	CS_Select();
	// payload command
	cmdtosend = R_RX_PAYLOAD;
	HAL_SPI_Transmit(NRF24_SPI, &cmdtosend, 1, 100);
	// Receive the payload
	HAL_SPI_Receive(NRF24_SPI, data, 32, 1000);
	// Unselect the device
	CS_UnSelect();
	HAL_Delay(1);
	cmdtosend = FLUSH_RX;
	nrfsendCmd(cmdtosend);
}
// Read all the Register data
void NRF24_ReadAll (uint8_t *data)
{
	for (int i=0; i<10; i++)
	{
		*(data+i) = nrf24_ReadReg(i);
	}
	nrf24_ReadReg_Multi(RX_ADDR_P0, (data+10), 5);
	nrf24_ReadReg_Multi(RX_ADDR_P1, (data+15), 5);
	*(data+20) = nrf24_ReadReg(RX_ADDR_P2);
	*(data+21) = nrf24_ReadReg(RX_ADDR_P3);
	*(data+22) = nrf24_ReadReg(RX_ADDR_P4);
	*(data+23) = nrf24_ReadReg(RX_ADDR_P5);
	nrf24_ReadReg_Multi(RX_ADDR_P0, (data+24), 5);
	for (int i=29; i<38; i++)
	{
		*(data+i) = nrf24_ReadReg(i-12);
	}
}
```

NRF24L01.h

```
#ifndef INC_NRF24L01_H_
#define INC_NRF24L01_H_
void NRF24_Init (void);
void NRF24_TxMode (uint8_t *Address, uint8_t channel);
uint8_t NRF24_Transmit (uint8_t *data);
void NRF24_RxMode (uint8_t *Address, uint8_t channel);
uint8_t isDataAvailable (int pipenum);
void NRF24_Receive (uint8_t *data);
void NRF24_ReadAll (uint8_t *data);
/* Memory Map */
#define CONFIG      0x00
#define EN_AA       0x01
#define EN_RXADDR   0x02
#define SETUP_AW    0x03
#define SETUP_RETR  0x04
#define RF_CH       0x05
#define RF_SETUP    0x06
#define STATUS      0x07
#define OBSERVE_TX  0x08
#define CD          0x09
#define RX_ADDR_P0  0x0A
#define RX_ADDR_P1  0x0B
#define RX_ADDR_P2  0x0C
#define RX_ADDR_P3  0x0D
#define RX_ADDR_P4  0x0E
#define RX_ADDR_P5  0x0F
#define TX_ADDR     0x10
#define RX_PW_P0    0x11
#define RX_PW_P1    0x12
#define RX_PW_P2    0x13
#define RX_PW_P3    0x14
#define RX_PW_P4    0x15
#define RX_PW_P5    0x16
#define FIFO_STATUS 0x17
#define DYNPD         0x1C
#define FEATURE       0x1D
/* Instruction Mnemonics */
#define R_REGISTER    0x00
#define W_REGISTER    0x20
#define REGISTER_MASK 0x1F
#define ACTIVATE      0x50
#define R_RX_PL_WID   0x60
#define R_RX_PAYLOAD  0x61
#define W_TX_PAYLOAD  0xA0
#define W_ACK_PAYLOAD 0xA8
#define FLUSH_TX      0xE1
#define FLUSH_RX      0xE2
```

```
#define REUSE_TX_PL   0xE3
#define NOP           0xFF
#endif /* INC_NRF24L01_H_ */
```

Main.c (ButtonBoard)

```
/* Includes
------------------------------------------------------------------*/
#include "main.h"
/* Private includes
----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "NRF24L01.h"
#include "string.h"
/* USER CODE END Includes */
/* Private typedef
-----------------------------------------------------------*/
/* USER CODE BEGIN PTD */
/* USER CODE END PTD */
/* Private define
------------------------------------------------------------*/
/* USER CODE BEGIN PD */
/* USER CODE END PD */
/* Private macro
-------------------------------------------------------------*/
/* USER CODE BEGIN PM */
/* USER CODE END PM */
/* Private variables
---------------------------------------------------------*/
SPI_HandleTypeDef hspi1;
/* USER CODE BEGIN PV */
/* USER CODE END PV */
/* Private function prototypes
-----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_SPI1_Init(void);
/* USER CODE BEGIN PFP */
/* USER CODE END PFP */
/* Private user code
---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
uint8_t RxAddress[] = {0x00,0xDD,0xCC,0xBB,0xAA};
uint8_t RxData[32];
uint8_t data[50];
//uint8_t TxAddress[] = {0xEE,0xDD,0xCC,0xBB,0xAA};
//uint8_t TxData[] = "Hello World\n";
/* 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();
 /* USER CODE BEGIN Init */
 /* USER CODE END Init */
 /* Configure the system clock */
 SystemClock_Config();
 /* USER CODE BEGIN SysInit */
 /* USER CODE END SysInit */
 /* Initialize all configured peripherals */
 MX_GPIO_Init();
 MX_SPI1_Init();
 /* USER CODE BEGIN 2 */
 NRF24_Init();
 NRF24_RxMode(RxAddress, 10);
//  NRF24_TxMode(TxAddress, 10);
 NRF24_ReadAll(data);
 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_7, 1);
 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_9, 1);
 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_9, 1);
 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_11, 1);
 /* USER CODE END 2 */
 /* Infinite loop */
 /* USER CODE BEGIN WHILE */
 while (1)
 {
   /* USER CODE END WHILE */
   /* USER CODE BEGIN 3 */
//      HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_7);  //Controls LED: D1 (UL)
//      HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_9);  //Controls LED: D3 (LL)
//      HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_9);  //Controls LED: D5 (LR)
//      HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_11); //Controls LED: D2 (UR)
//------------------------FINAL
CODE-----------------------------------------
        if(HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_8) == 0){ //Pushbutton LL
              HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_9);
              HAL_GPIO_WritePin(GPIOC, GPIO_PIN_7, 1);
              HAL_GPIO_WritePin(GPIOA, GPIO_PIN_9, 1);
              HAL_GPIO_WritePin(GPIOA, GPIO_PIN_11, 1);
```

```
        }
        if(HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_8) == 0){ //Pushbutton LR
              HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_9);
              HAL_GPIO_WritePin(GPIOC, GPIO_PIN_7, 1);
              HAL_GPIO_WritePin(GPIOC, GPIO_PIN_9, 1);
              HAL_GPIO_WritePin(GPIOA, GPIO_PIN_11, 1);
        }
        if(HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_6) == 0){ //Pushbutton UL
              HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_7);
              HAL_GPIO_WritePin(GPIOC, GPIO_PIN_9, 1);
              HAL_GPIO_WritePin(GPIOA, GPIO_PIN_9, 1);
              HAL_GPIO_WritePin(GPIOA, GPIO_PIN_11, 1);
        }
        if(HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_10) == 0){ //Pushbutton UR
              HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_11);
              HAL_GPIO_WritePin(GPIOC, GPIO_PIN_7, 1);
              HAL_GPIO_WritePin(GPIOC, GPIO_PIN_9, 1);
              HAL_GPIO_WritePin(GPIOA, GPIO_PIN_9, 1);
        }
//-------------------------------------------------------------------------
-----
//      HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_11);
              if (isDataAvailable(2) == 1)
              {
                    NRF24_Receive(RxData);
//                  HAL_UART_Transmit(&huart2, RxData, strlen((char *)RxData),
1000);
//                  if(*RxData != 0x0)
//                        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_7, 0);
              }
//      uint8_t tx = NRF24_Transmit(TxData);
//      if (tx == 1)
//      {
//            HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_11);
//      }
//      else if(tx == 2)
//      {
//            HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_7);
//      }
        HAL_Delay(500);
 }
 /* USER CODE END 3 */
}
/**
 * @brief System Clock Configuration
 * @retval None
 */
void SystemClock_Config(void)
{
```

```
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
 /** Configure the main internal regulator output voltage
 */
 __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);
 /** Initializes the RCC Oscillators according to the specified parameters
 * in the RCC_OscInitTypeDef structure.
 */
 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 RCC_OscInitStruct.HSIState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
 RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
 RCC_OscInitStruct.PLL.PLLMUL = RCC_PLLMUL_4;
 RCC_OscInitStruct.PLL.PLLDIV = RCC_PLLDIV_2;
 if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
 {
   Error_Handler();
 }
 /** Initializes the CPU, AHB and APB buses clocks
 */
 RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
 RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
 RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
 if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1) != HAL_OK)
 {
   Error_Handler();
 }
}
/**
 * @brief SPI1 Initialization Function
 * @param None
 * @retval None
 */
static void MX_SPI1_Init(void)
{
 /* USER CODE BEGIN SPI1_Init 0 */
 /* USER CODE END SPI1_Init 0 */
 /* USER CODE BEGIN SPI1_Init 1 */
 /* USER CODE END SPI1_Init 1 */
 /* SPI1 parameter configuration*/
 hspi1.Instance = SPI1;
 hspi1.Init.Mode = SPI_MODE_MASTER;
 hspi1.Init.Direction = SPI_DIRECTION_2LINES;
 hspi1.Init.DataSize = SPI_DATASIZE_8BIT;
 hspi1.Init.CLKPolarity = SPI_POLARITY_LOW;
```

```
 hspi1.Init.CLKPhase = SPI_PHASE_1EDGE;
 hspi1.Init.NSS = SPI_NSS_SOFT;
 hspi1.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_8;
 hspi1.Init.FirstBit = SPI_FIRSTBIT_MSB;
 hspi1.Init.TIMode = SPI_TIMODE_DISABLE;
 hspi1.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
 hspi1.Init.CRCPolynomial = 7;
 if (HAL_SPI_Init(&hspi1) != HAL_OK)
 {
   Error_Handler();
 }
 /* USER CODE BEGIN SPI1_Init 2 */
 /* USER CODE END SPI1_Init 2 */
}
/**
 * @brief GPIO Initialization Function
 * @param None
 * @retval None
 */
static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};
/* USER CODE BEGIN MX_GPIO_Init_1 */
/* USER CODE END MX_GPIO_Init_1 */
 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOH_CLK_ENABLE();
 __HAL_RCC_GPIOA_CLK_ENABLE();
 __HAL_RCC_GPIOC_CLK_ENABLE();
 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOA, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_9|GPIO_PIN_11,
GPIO_PIN_RESET);
 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_7|GPIO_PIN_9, GPIO_PIN_RESET);
 /*Configure GPIO pins : PA1 PA2 PA9 PA11 */
 GPIO_InitStruct.Pin = GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_9|GPIO_PIN_11;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
 /*Configure GPIO pins : PC6 PC8 */
 GPIO_InitStruct.Pin = GPIO_PIN_6|GPIO_PIN_8;
 GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
 /*Configure GPIO pins : PC7 PC9 */
 GPIO_InitStruct.Pin = GPIO_PIN_7|GPIO_PIN_9;
 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 pins : PA8 PA10 */
 GPIO_InitStruct.Pin = GPIO_PIN_8|GPIO_PIN_10;
 GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
/* USER CODE BEGIN MX_GPIO_Init_2 */
/* USER CODE END MX_GPIO_Init_2 */
}
/* USER CODE BEGIN 4 */
/* USER CODE END 4 */
/**
 * @brief  This function is executed in case of error occurrence.
 * @retval None
 */
void Error_Handler(void)
{
 /* USER CODE BEGIN Error_Handler_Debug */
 /* User can add his own implementation to report the HAL error return state
*/
 __disable_irq();
 while (1)
 {
//      HAL_GPIO_WritePin(GPIOA, GPIO_PIN_9, 1);
 }
 /* USER CODE END Error_Handler_Debug */
}
#ifdef  USE_FULL_ASSERT
/**
 * @brief  Reports the name of the source file and the source line number
 *         where the assert_param error has occurred.
 * @param  file: pointer to the source file name
 * @param  line: assert_param error line source number
 * @retval None
 */
void assert_failed(uint8_t *file, uint32_t line)
{
 /* USER CODE BEGIN 6 */
 /* User can add his own implementation to report the file name and line
number,
    ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line)
*/
 /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```