Final Project Report: Honk of the Fates

ECE 167

Kevin Jiang, Ishan Madan, Indy Spott, Eliot Wachtel

March 17, 2024

Final Commit ID: `972bd64df37bca6c0f536d6a02c0fcc3af6ab4b6`

1 Introduction

The goal of this project was to make a wireless communication system utilizing a wand with an IR emitter, three buttons, and an RGB LED strip leading back to connecting to the Nucleo-64 board incorporated into a chest piece. This chest piece also contains a mounted IR receiver, a quadrature encoder, and a Bluetooth module. Each component plays a key roll in this project, along with the quantization taken to ensure working modules. The ultimate game play will be explained with further detail in Section 3.6.1, where the state machine is described. However, we will do an overview here.

To begin, the chest module is utilized for player color selection, IR receiving, and packet sending. In order to start the game, a python script is run on the laptop which locates all powered Bluetooth modules waiting to connect (specifically of the same brand, Adafruit Bluefruit). This will find the modules and send a connection acknowledgement in order to move on to the next portion of the game. Once the STM32 has received this acknowledgement, the player will be able to choose the color of the device. This is split into eight color options and will decide the color of the wand and their chest component. Once this color has been selected, the player can press any button on the wand which will trigger a message telling the laptop that a color has been chosen and that the player is ready to start. The computer, once all players have been recognized, will proceed to assign unique player ID's to each of the STM32's and trigger the start of the game. During gameplay the player STM will watch for button, IR, and Bluetooth packet events and act upon them to further the game. The IR receiver will wait for a signal from any of the opposing player's wands and then send a packet using the Bluetooth module if an attack from an opposing ID is detected.

To follow, if one of the buttons on the wand is pressed, a blasting animation will occur on the RGB strip integrated into the wand. The color of this will match the color that the player chose at the beginning of the game. Alongside this, the button will trigger an interrupt that sets off the IR emitter and sends a series of pulses reliant on the player ID assigned. In the same instance, a packet will be sent to the Bluetooth module that contains the button that was pressed and that a hit was sent. The computer will receive this and then do an on-board calculation based on when the packet was received and what spell was sent to determine which player won. Therefore, the players will be able to continuously blast magic, but the computer and latency will determine the "winner" of that point.

The computer will have its own state machine that determines which player receives the point and information handling. The handling of information will be hits, blasts, player ID's, acknowledgements, color, etc. It will be the referee of the game. This will be further explained in Section 3.7.

2 Background

Given the overarching goal of our project, a variety of sensors and techniques had to be implemented, many of which were time sensitive. Button and encoder input were controlled similarly to the labs without much issue. Controlling the RGB LEDs was done by leveraging the SPI transmission hardware to offload most of the bit-banging logic that would otherwise be necessary to simulate the signal (elaborated in section 3.4). Controlling the Bluetooth required leveraging the UART library with a decent amount of support code added to properly manage the signal and mitigate packet loss (elaborated in section 3.2). For IR reception we made use of a component with a built in band-pass filter to reduce the risk of noise (elaborated in section 3.1). For the IR transmission the PWM library was leveraged to achieve the proper frequency (also elaborated in section 3.1). For all aspect of the project, we made heavy use of the timers library to ensure no blocking code was used anywhere (you should be able to search our C files and find no HAL_delay functions in use) (elaborated in multiple sections, but especially in 3.1 and 3.4).

We thought through the selection for each of these components carefully.
Buttons and encoders make convenient and effective simple user input devices so using the course encoder and simple mom-off push-buttons was selected for user input on the wand.
For Bluetooth communication (the protocol suggested by Professor Josephson) we found a development board from Adafruit which looked promising with Python library support on laptops and an integrated UART bus.
For IR transmission we originally hope to use BELS stock IR Light Emitting Diodes, but these proved to be far too weak to use individually across a room. Thus we pulled stronger diodes out of low-cost IR remotes.
For IR reception we decided to use a more complex IR digital receiver which incorporated an internal amplification and band-pass stage to ensure a more stable and reliable signal reception. This part did require us to send somewhat more complex pulses of high frequency IR than a regular photodiode or phototransistor would have, but the benefit was that we saw almost no signal noise from any other source (except for other IR remotes using the same common frequency band).
For game status indication on the wand, we chose WS2812 style addressable RGB LEDs in a LED strip form factor. These provide a convenient to wire 1-wire serial control interface which can be simulated using a properly configured SPI bus.

3 Implementation

3.1 Infrared Emitter + Receiver

The first thing we did, before involving the microcontroller, was hook up the sensor circuits to an oscilloscope and waveform generator to ensure that they worked as expected.

Since the receiver is designed to reject environment noise, it has a very tight bandwidth centered on 30kHz. As such, we decided to send signal using a 30kHz PWM signal going into the IR LED. By turning the PWM channel on and off, we could pulse this 30kHz frequency to get high and low signals on the receiver, allowing us to transmit binary data. Though testing, the 30kHz is absolutely necessary and changing it by even 1kHz drops the ability to receive a signal by enough that we suspect a passband of around 1kHz.

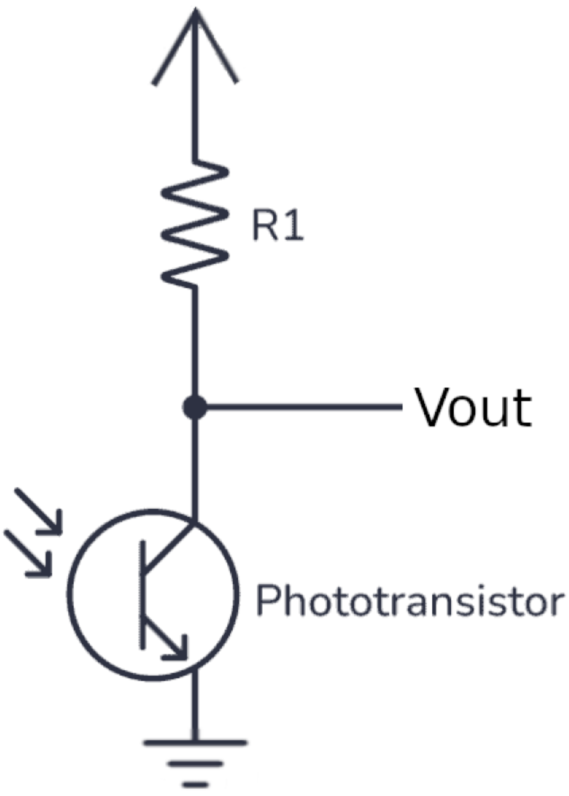


Figure 1: Simple phototransistor low side divider sensor circuit

We then hooked the receiver up to the same interrupt pin (pin 40) used in the capacitive touch lab. We then sent pulses from the first STM32 to the second over IR.

Starting with 3 pulses of length 100ms with pauses of 100ms in between with a 3 second delay between sets, we found the signal of the receiver slightly unstable, so I shortened the on pulse to 50ms (maintaining the 100ms pause in between) and that made it far more consistent.

To count the pulses per set, we used the following interrupt logic:

```
// Define variables externally:
unsigned int temp_time_storage = 0;
unsigned int last_time = 0;
unsigned int counter = 0;


// Interrupt handler function
```

```
function onInterrupt() {
    clearInterruptFlag(); // clear interrupt signal
    temp_time_storage = getMilliseconds(); // get current time in milliseconds

    // check if the elapsed time since last_time is 300ms or more
    if (temp_time_storage >= last_time + 300) {
        counter = 1; // reset counter
        last_time = temp_time_storage; // update last_time
    }
    // check if the elapsed time since last_time is 100ms or more
    else if (temp_time_storage >= last_time + 100) {
        counter += 1; // increment counter
        last_time = temp_time_storage; // update last_time
    }
    setInterruptProcessedFlag(); // indicate interrupt has been processed
}
```

The above code has safety for figuring out if it's the start of a new message or if the pulse is the next in a currently incrementing series.

In order to check that a signal had been received and to check if it was a fully finished count, we exposed two functions. One checked and reset the flag showing the interrupt had occurred. The second exposed the last_time variable so that the main loop could wait for a post reception delay.

By sending bits at an interval of 100ms, we could simply wait for a safe time of 300ms to pass and then declare the message fully received and ready to process.

In the main loop of our STM32 program, we first set up a char variable flag. Building with our functions, `IR_Count()`, `IR_timecheck()`, and `IR_Detect()` from our IR program file, we set our `char` flag high if `IR_Detect()` returns `TRUE`, meaning that we've received an IR interrupt. If the flag is true and `IR_timecheck() + 300` is less than `TIMERS_GetMilliSeconds()`, indicating that 300 ms has elapsed, then we check whether the `IR_count` matches the player ID. First, we check if it is our own player ID, calculated as the absolute value of `IR_Count() - playerID * 3`, where the player ID is a fixed integer from 1 to 8 assigned to the player. If this is true, we simply ignore it, as we don't want players to be able to shoot themselves. Otherwise, we simply calculate the modulo of the IR count by 3. If the modulo is 0 or 1, then we know the hit detection is from the player whose ID is the integer division of the number of pulses received by 3. If the modulo is 2, then we know it is from the player whose ID is one more than the integer division of the number of pulses received by 3. We take this information and put it in a packet to send to the computer, then lower our flag back to `FALSE`.

During testing we did end up damaging one of the receivers by plugging it in backwards. It does not have reverse polarity protection.

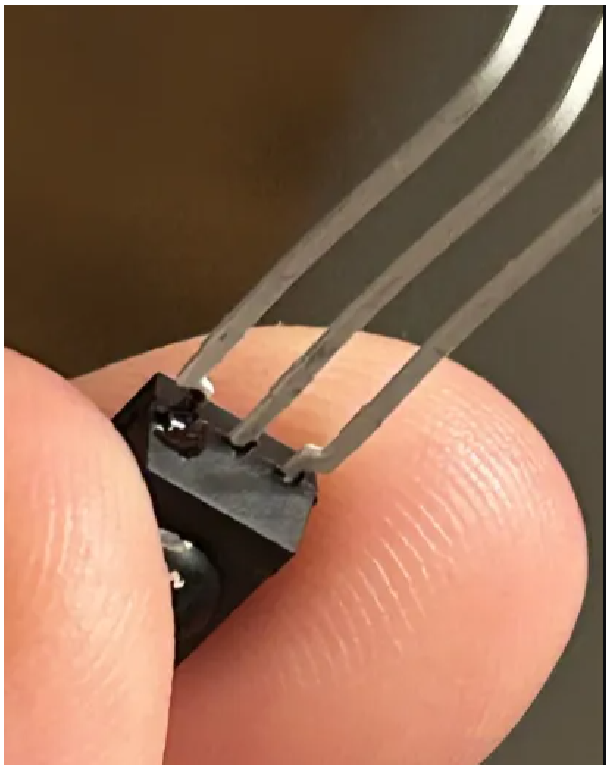


Figure 2: The damaged IR receiver. Notice the shiny spot by the leftmost leg where the internals melted the plastic casing.

We decided to set transmitted messages to three times the sending player's ID number, thus if the received count was off by 1 we could still round to the correct value.

Because we have a lot of time sensitive processes running in parallel, it is necessary to make all of our code reasonably efficient and completely non-blocking wherever possible. As such, while the detection could use interrupts, the final transmission code needed a different method to trigger the periodic enable and disable of the 30kHz PWM signal.

To avoid needing a complex timer interrupt with periodic period changes and periodic activation and deactivation, we settled on a global-clock associate relative-schedule method. This works by writing a handler function which is run every main loop iteration, but only actually causes an action if a certain amount of global time has elapsed since it took the last action.

Below is the function which serviced the IR signal transmission:

```
int sendIRsignal(int playerID){
    if(playerID == 0) return 0; // if input is 0, do not do anything

    // check current time
    check current time and save to temp varaible tempIRTime
    If currently running a signal and ready to do the next item because
    tempIRTime >= (lastIRUpdate + currDelay):
        if (signal_propegation >= (playerID * 3))
        {
            runningIRpulses = FALSE;
            PWM_Stop(PWM_5); // double check that it is stopped
            signal_propegation = 0;
            currDelay = MIN_WAIT_BETWEEN_PULSES;
            return TRUE;
        }

        if(currDelay == ON_TIME){
            // turns off the pwm to give a pause time
            PWM_Stop(PWM_5);
            currDelay = OFF_TIME;
            signal_propegation ++; // increment to track the newly completed pulse.
        } else if(currDelay == OFF_TIME){
            // Turns the pwm on to generate a pulse.
            PWM_Start(PWM_5);
            currDelay = ON_TIME;
        }
        // update last update time store
        lastIRUpdate = TIMERS_GetMilliSeconds();
    } else if (!runningIRpulses && tempIRTime > (lastIRUpdate +
            MIN_WAIT_BETWEEN_PULSES)){
        // start a new pulse
        runningIRpulses = TRUE;
        PWM_Start(PWM_5);
        currDelay = ON_TIME;
        signal_propegation = 0;
        lastIRUpdate = TIMERS_GetMilliSeconds();
    }
    return FALSE;
}
```

The above code does the equivalent of the following six lines, but in a non-blocking manor.

```
for(int i = 0; i < playerID*3; i++){
    PWM_Start(PWM_4);
    HAL_Delay(50);
    PWM_Stop(PWM_4);
    HAL_Delay(100);
}
```

When called, the non-blocking code will either start a new IR signal or service the active one (depending on if there is an active one and what it's inputs are). Starting a signal simply involves turning on the PWM to start the first bit transmission, zeroing the relevant counters, and recording the transmission start time. To service an existing signal, the function checks if the appropriate amount of global time has passed since the last update and will then turn the PWM on/off resetting the recorded last-update time and returning FALSE if the signal is still in progress or TRUE if it has just performed the last PWM deactivation to complete signal transmission.

In the main code, starting a signal requires calling the function once with the player ID. It can then be safely called each while loop iteration with the same ID until it returns a TRUE. We made this happen with a flag that sets when a spell is case and opens an if to run the service until it is done at which point the flag is reset to FALSE.

3.2 Bluetooth

"You're your own worst enemy" (Petersen, 2024).

"It's so fucking fast anyway" (Petersen, 2024).

One of the largest hurdles of this project was the Bluetooth implementation. When planning this project, we set out to source a Bluetooth module and designing a library to provide an interface for wireless transmission of data between the STM32 and a laptop. The idea behind this goal was initially provided by Professor Josephson and Adam, who envisioned our work as a stepping-off point to potentially add a Bluetooth interface to a future I/O shield. Thus, we (and possibly future classes) could wirelessly send data between the STM32 and a student laptop, enabling projects and labs to be conducted without the STM32 tethered directly to a computer with a serial monitor.

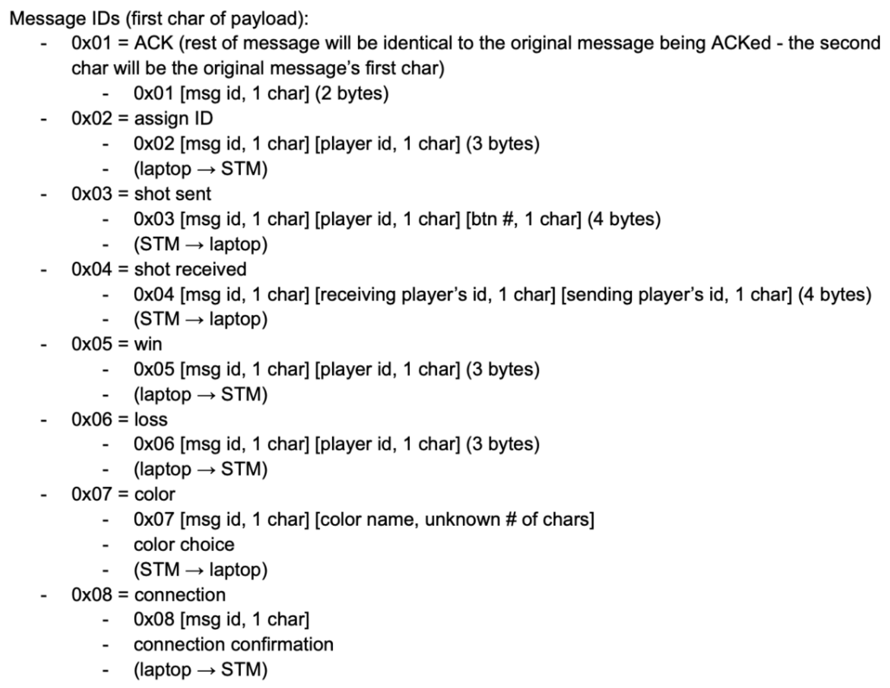
We started by sourcing a Bluetooth board to connect to the STM32. We eventually settled on the Adafruit Bluetooth LE UART Friend, a fairly low-cost option that was effectively a development board for a smaller, more standardized Bluetooth module (the nRF51822), providing a UART interface to the Bluetooth protocol. We chose this board for a few reasons, including the ease of use of UART (unlike the SPI version, we already had fully-functional libraries for UART provided by Adam). We also appreciated that Adafruit sold just the nRF51822 separately and a schematic for the Friend board, as that seemed to enable a future redesign of the I/O shield that could include this Bluetooth chip. Furthermore, our cursory overview of the Friend's documentation indicated that Adafruit provided source code for both sides of the Bluetooth connection: code to interface with the board from an Arduino and code for their mobile and desktop apps to connect to the board via Bluetooth.

Once the Bluetooth boards arrived, we began to test with them. Connecting to the board from an STM32 was simple enough; we used the I/O shield's UART1 pins and Adam's UART library, and with a little debugging, we were able to echo packets over Bluetooth between the iOS app and the STM32 + development board. We did notice a strange issue where the bytes we read from UART on the STM32 seemed to loop through whatever size buffer we read into. This meant that each time we read a series of bytes, the STM32's UART library seemed to maintain a moving cursor of the end of the last set of bytes read in and started writing bytes at that position. However, we had been expecting functionality similar to the PIC32's UART module, which would write starting at the beginning of the provided buffer upon each call to the read function. We spent several days trying to debug this issue, unsuccessfully trying packet implementations, delimiters, and various other ways to indicate new data as unique from old data. We also consulted Professors Josephson and Petersen in this effort. Professor Josephson recommended we look into a library by Google called Protocol Buffers and try to understand how the Adafruit-provided Arduino code successfully interfaced with the board. Professor Petersen advised us to look into the UART implementation of the STM32 and pointed out that there was no real issue with only reading one byte at a time, as our program would be fast enough regardless. He also suggested we migrate our code to the PIC32, where we might know more about how UART is implemented.

Eventually, we realized that clearing the buffer before calling the `Uart1_rx()` function eliminated the issue of old data vs. new data, since the UART function did not actually output the full buffer's worth of characters upon each call. Clearing the buffer did not rectify the moving cursor phenomenon, but we determined that this was acceptable due to our ability to just find the beginning of each incoming packet and parse circularly from there. At this point, we were able to send and receive strings of data, reliably determining where they started and ended.

Next, we wanted to implement a packet protocol to ensure we were able to check for completeness of incoming messages and discard malformed packets. To do this, we relied heavily on Lab 1 from ECE 121, where a similar packet protocol was implemented over a wired UART/USB connection. For this project, we implemented a nearly identical protocol with the same format of head, tail, and end bytes in addition to the payload, an ID, a length, and a checksum. The checksum was calculated with a simple bitwise XOR over the ID and payload, while the ID uniquely identified each message rather than denoting a message type. This last change was made to allow us to differentiate between unique messages without relying on the possible occurrence of two separately sent but identical payloads.

At this point, we needed to transition away from using the mobile app. While it was invaluable to have a guaranteed connection when testing the board, we needed to be able to communicate between the board and a laptop, which led us to the Bluetooth Python wrapper library provided by Adafruit. Although we hadn't realized when initially sourcing the board that this library had been deprecated,


Message IDs (first char of payload):

- 0x01 = ACK (rest of message will be identical to the original message being ACKed - the second char will be the original message's first char)
    - 0x01 [msg id, 1 char] (2 bytes)
- 0x02 = assign ID
    - 0x02 [msg id, 1 char] [player id, 1 char] (3 bytes)
    - (laptop → STM)
- 0x03 = shot sent
    - 0x03 [msg id, 1 char] [player id, 1 char] [btn #, 1 char] (4 bytes)
    - (STM → laptop)
- 0x04 = shot received
    - 0x04 [msg id, 1 char] [receiving player's id, 1 char] [sending player's id, 1 char] (4 bytes)
    - (STM → laptop)
- 0x05 = win
    - 0x05 [msg id, 1 char] [player id, 1 char] (3 bytes)
    - (laptop → STM)
- 0x06 = loss
    - 0x06 [msg id, 1 char] [player id, 1 char] (3 bytes)
    - (laptop → STM)
- 0x07 = color
    - 0x07 [msg id, 1 char] [color name, unknown # of chars]
    - color choice
    - (STM → laptop)
- 0x08 = connection
    - 0x08 [msg id, 1 char]
    - connection confirmation
    - (laptop → STM)

Figure 3: List of messages being sent between Bluetooth modules

we decided to try it anyway (in the absence of any other practical options that did not involve writing a library ourselves). Surprisingly, it worked fairly well and with some debugging effort, was able to perform a simple echo between the laptop and the board. We implemented the same packet protocol state machine in our Python code (`stm_bluefruit.py`) as we did on the STM32 (`BT.c/h`). Then, we wrote yet another echo test (`echo.py, echo.c`) to send data between the laptop and STM32, except this time through packets.

HEAD	LENGTH	(ID)PAYLOAD	TAIL	CHECKSUM	END
1	1	(1)<128	1	1	\r \n

The above table is how the packets were made to be sent and received. With this, we created our own version of packet messages to signify a number of things. This is in the list below:

Each of these are sent back and forth between the computer and the STM32 and read through their buffers. These messages were used to trigger state transitions and move forward with the gameplay, along with ensuring connection status for sanity checks. This was fairly straightforward to implement and was fairly consistent when it came to the game play, aside from an occasional spike in packet losses when buttons were spammed continuously through a short period of time. Other than that, there are some caveats with the Bluetooth that are discussed in the 4.

3.3 QEI

This section was almost exactly the same as the lab 2 integration of the quadrature encoder. The only difference from lab 2 was that rather than being split into 24 increments, we split it into 8. This was because we wanted a maximum of 8 colors to be chosen for gameplay to be simplistic. Alongside this, instead of the counter being able to increment or decrement, the counter was made to be an absolute value in order to make the color picking straightforward in either direction that you turned the encoder. This utilized the QEI.h from the class libraries, but our own QEI.c library from lab 2. The values were also hard implemented to be distinct colors to ensure no color overlap between the 8 distinct colors. The overall schematic of this component being integrated is shown in Sch. 9.

3.4 WS2812 RGB LEDs

"Don't mind me, I'm just washing my LEDs" (Wachtel, 2024).

While working on the WS2812 protocol, a new Linux laptop was introduced. In order to upload code, we had to follow the instructions in Piazza and then follow an instruction from Stack Overflow[2] leading to the following set of commands:

```
sudo chmod a+wrx /dev/ttyACM0
sudo apt -y install stlink-tools
sudo systemctl restart udev
```

These three together seemed to resolve the permission issue (and should probably be added to the STM32 documentation).

The WS2812 is the name of the ubiquitous IC which is integrated into and controls most addressable LEDs on the market (including Neopixels which are just rebranded WS2812 LEDs).

The chip is based on a successive shift register digital protocol where controllers are strung in series to form a bus and updates are sent as a long string of bytes where each LED will take the first bytes and pass the rest along. Accounting for parallel power supply to the LED diodes themselves.

The information itself is sent as a set of three bytes (three 8-bit RGB values) which control the intensity of each color. The data itself is encoded as long and short pulses according to the following diagram 4.

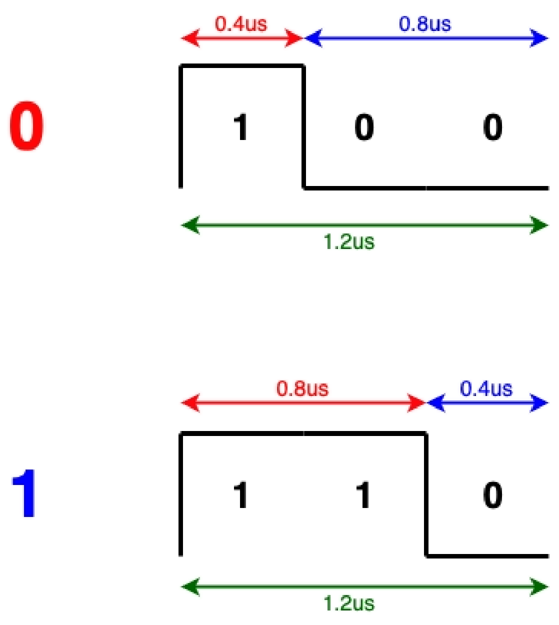


Figure 4: Diagram of encoded long and short pulses[1]

(Note: the controllerstech website that was used to get information and control method inspiration from utilizes a concerning amount of CPU (literally spiking individual cores to 100% periodically), leading to a suspicion of lots of Google adsense calls.)

Luckily, the SPI single wire protocol turns out to be really effective at simulating the correct data transmission rates. By setting the clock to the correct speed and configuring the SPI in single wire master mode an 8-bit set can be configured with either one or two successive 1's at the start, giving the correct pulse for high and low signals. Stringing together strings of these allows for full transmission of the protocol.

Configuring SPI took a bit of trial and error (especially since the recommendations from the controllers tech website were not all correct) with the main issues being getting the correct setting registers for the SPI (which would be useful to break out as part of the default library for future classes) and playing with the frequency. Controllers tech listed that a baud rate scalar of 32 should be used, but this was wrong because it made the pulses too long (leading to only high signals being sent). Changing to the smaller scalar of 16 fixed this.

To control the LEDs five functions are defined. The first initializes the SPI and is called once at the start. The next three are used to update an array storing RGB values and to recall those values, convert them to a transmit ready format and transmitting them over SPI. The last function uses a similar strategy to the IR transmission to run animation frames for a firing animation at a set speed within successive calls.

When called the following code will check for RGB LED animation status and if the flag is TRUE will service a current animation or start a new one if there is none active. As with the IR transmission function, this one is compute light when called unless it is time for a new frame and, so, it is safe to call repeatedly in the main() while loop.

```
void spellPulse(spell used_spell, int start):
    check current time and save to temp varaible tempTime


    the "start" input can start a new pulse if one hasn't been started yet


    If currently running an animation and ready to do the next
                    frame because tempTime > (lastUpdate + 10)):
        if all LEDs have been animated then:
            runningPulse = FALSE; // end the animation
            reset the last LED to defaul colour
            zero propegation tracking variable
        else:
            incrment frame propegation tracking variable
            switch (used_spell) // based on spell input, use different animation color
                case UNSPECIFIED:
                    reset the LED (pulse_propagation - 1) to default colour
                    set the LED (pulse_propagation) to white
                    break;
                case ROCK:
                    // Turns the pulse pixel off.
                    reset the LED (pulse_propagation - 1) to default colour
                    set the LED (pulse_propagation) to green
                    break;
                case PAPER:
                    reset the LED (pulse_propagation - 1) to default colour
                    set the LED (pulse_propagation) to yellow
                    break;
                case SISSORS:
                    reset the LED (pulse_propagation - 1) to default colour
                    set the LED (pulse_propagation) to red
                    break;
        // update last update time storage variable
        WS2812_Send(); // transmit new LED states to the WS2812 chips over SPI
```

We planned to make other animations for victory/loss and charging animations for delays after firing, but ran out of time to effectively do so. Those animations would be set up as similar service functions, with flags to prevent them from running at the same time.

3.5 Wand Design

We designed the wands with three main design principles in mind:

1. Be comfortable to hold and use
2. Print reliably and without support material on an FDM printer
3. Make wire routing as easy as reasonable with the other two goals

To accomplish this, the wand was designed in two halves, allowing internal structures to be incorporated while leaving flat external surfaces to print onto the build plate as seen in Fig. 5)

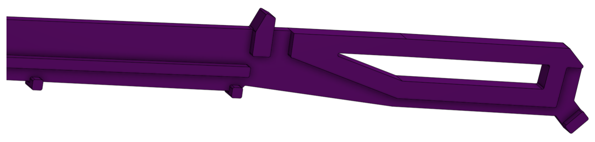


Figure 5: A view to some of the internal structural elements for providing structure and wire routing paths

The first iteration of the design was a structural concept which was decided to be too much of a firearm profile facsimile so we switched to a slightly more straight design and used a more standout colour.

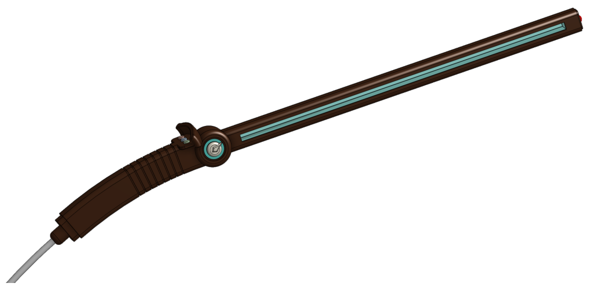


Figure 6: The original wand design (which was deemed too gun-like)

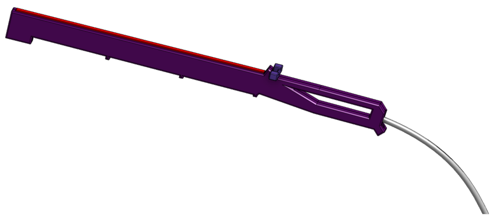


Figure 7: The updated wand with purple colour applied, notice the addition of guiding clips for the wires.

3.6 State Machine

3.6.1 STM32 State Machine

Software Modules: The software architecture is structured around modular components, each responsible for specific tasks. These include:

- ADC: Used for reading analog inputs.
- Buttons: Manages button inputs from the user.
- PWM: Controls the duty cycle of PWM signals for RGB LED control, and IR emitter pulses.
- QEI: Implements quadrature encoder interface functionality.
- Timers: Facilitates precise timing operations.
- Bluetooth (BT): Handles Bluetooth communication with the laptop.
- Neopixel: Controls RGB LEDs.
- IR: Provides functionality for infrared hit detection.

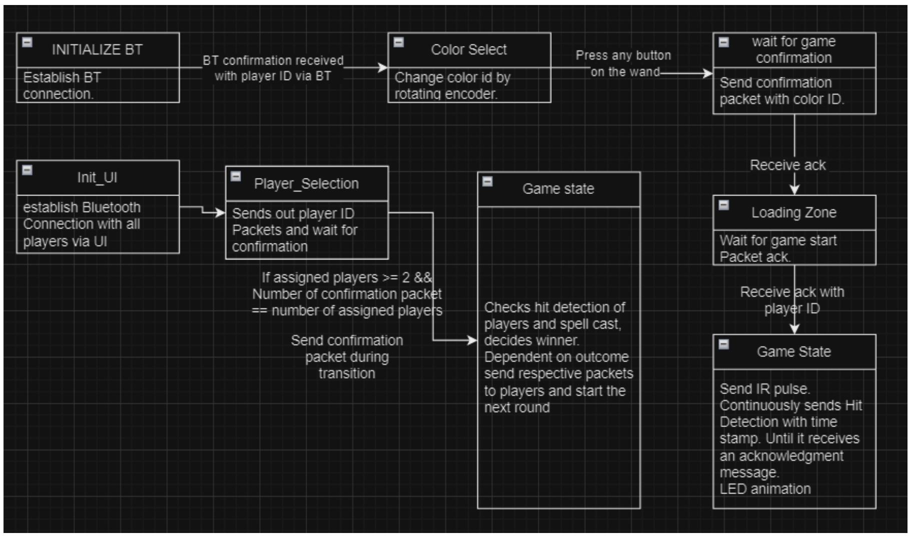

INITIALIZE BT
Establish BT connection.
BT confirmation received with player ID via BT
Color Select
Change color id by rotating encoder.
Press any button on the wand
wait for game confirmation
Send confirmation packet with color ID.
Receive ack
Loading Zone
Wait for game start Packet ack.
Receive ack with player ID
Game State
Send IR pulse. Continuously sends Hit Detection with time stamp. Until it receives an acknowledgment message. LED animation
Init_UI
establish Bluetooth Connection with all players via UI
Player_Selection
Sends out player ID Packets and wait for confirmation
If assigned players >= 2 && Number of confirmation packet == number of assigned players
Send confirmation packet during transition
Game state
Checks hit detection of players and spell cast, decides winner. Dependent on outcome send respective packets to players and start the next round

Figure 8: State-Machine Diagram

In our implementation of the state machine for the STM32 microcontroller, we have designed a robust system to manage various functionalities including Bluetooth communication, color selection, game initialization, and gameplay mechanics.

The state machine is structured into several states, each with specific tasks and transitions triggered by external events or conditions.

The program is currently in the `STATE_IDLE` state, indicating it's waiting for a connection confirmation from the UART initialization.

- The program checks if the first byte of the Bluetooth buffer (`BT_buffer`) matches the hexadecimal value `0x08`. This value corresponds to the color code assigned from the documentation.
  - If the condition is true, it indicates that the received message signifies a connection confirmation.
    - It sets the first byte of the Bluetooth buffer to `0x01`, indicating an acknowledgment (ACK) message.
    - The program sends the acknowledgment to the laptop using the `BT_Send` function, transmitting 2 bytes.
    - It increments the message ID (`activeID`) by 1, ensuring it stays within the range of 0 to 255.
    - There's a placeholder comment suggesting that if no acknowledgment is received, further actions won't proceed.
    - Finally, the program changes its state to `STATE_P_INIT`, signifying a transition to the next state.

state = STATE_P_INIT;

This line changes the state of the system to "STATE_P_INIT", indicating that the system is ready to proceed with the next initialization step.

Color Selection State (STATE_P_INIT): Here, the player chooses their preferred color using the buttons. The RGB values corresponding to the color are determined using the encoder. Upon button press, the selected color is identified and stored. Once the color selection is completed, the system transitions to the connection confirmation state (STATE_CON).

The program begins with an `if` statement that checks if any of the buttons 2, 3, or 4 are pressed. This check is performed using bitwise operations to determine if the corresponding bits in the `buttons` variable are set. If any of these buttons are pressed, the program proceeds with the following actions:

- The program enters a series of `if-else-if` statements to determine the color based on the current values of the red, green, and blue (RGB) color components. These values are likely being modified by an encoder or some other input mechanism.
  - If the RGB values match specific combinations, corresponding to predefined colors, the program assigns a string representation of that color to the variable `color` using the `strcpy` function.
  - Each `if-else-if` statement checks for a specific combination of RGB values and assigns the appropriate color name accordingly. The colors and their corresponding RGB values are as follows:
    - Navy: (255, 10, 255)
    - Rose: (0, 100, 200)
    - Green: (255, 255, 0)
    - Red: (0, 255, 255)
    - Cyan: (0, 100, 255)
    - Yellow: (255, 255, 255)
    - Purple: (255, 10, 255)
    - White: (255, 255, 255)
- After determining the color based on the RGB values, the program changes its state to `STATE_CON`, indicating that it is ready to transition to the next state.

Connection Confirmation State (STATE_CON): In this state, the system waits for confirmation from the STM32 that the color selection message has been received. If confirmation is received, the system transitions to the waiting state (STATE_WAIT), where it awaits further instructions from the game coordinator. If no confirmation is received, a safety transmission of the color selection message is triggered to ensure data integrity.

The program is in the `STATE_CON` state, indicating it's expecting confirmation from the STM32 device.

1. The program checks if the first byte of the Bluetooth buffer (`BT_buffer`) matches `0x08`. If it does, it signifies a confirmation from the STM32.
    - If confirmed:
        - It sets the first byte of the buffer to `0x01`, indicating an acknowledgment (ACK).
        - It sends the acknowledgment to the laptop using the `BT_Send` function, transmitting 2 bytes.
        - The program increments the message ID (`activeID`) by 1, ensuring it stays within the range of 0 to 255.
        - There's a placeholder comment indicating that if no acknowledgment is received, further actions won't proceed.
2. The program checks if the first byte of the Bluetooth buffer (`BT_buffer`) matches `0x01`. If it does, it means an acknowledgment (ACK) was received from the laptop.
    - If acknowledged:
        - The program changes its state to `STATE_WAIT`, implying it's ready to start some operation.
3. If neither of the above conditions is met, it enters the `else` block, indicating some other data was received.
    - It sets the first byte of the buffer to `0x07`, indicating a color message.
    - It copies the color value into the buffer to be sent to the laptop.
    - The program sends the color information to the laptop using the `BT_Send` function, transmitting the color data along with its length.
    - The program increments the message ID (`activeID`) by 1, ensuring it stays within the range of 0 to 255.
    - There's a placeholder comment indicating that if no acknowledgment is received, further actions won't proceed.

Waiting State (STATE_WAIT): This state is responsible for waiting for the game start message from the game coordinator via Bluetooth. Upon receiving the game start signal, the system transitions to the game initialization state (STATE_START).
The program is currently in the `STATE_WAIT` state, indicating it's waiting for a specific condition to be met before proceeding.

- The program begins by checking if the first byte of the Bluetooth buffer (`BT_buffer`) matches the hexadecimal value `0x02`. This condition checks for a specific message type that the program is expecting.

  - If the condition is true:
    * The program assigns the value of the second byte of the Bluetooth buffer (`BT_buffer[1]`) to the variable `playerID`, presumably indicating the ID of the player associated with the received message.
    * It sets the first byte of the Bluetooth buffer to `0x01`, indicating an acknowledgment (ACK) message.
    * The program sends the acknowledgment to the laptop using the `BT_Send` function, transmitting 2 bytes.
    * The program increments the message ID (`activeID`) by 1, ensuring it stays within the range of 0 to 255.
    * Finally, the program changes its state to `STATE_START`, implying it's ready to transition to the next state.

Game Initialization State (STATE_START): Here, the system prepares for gameplay by acknowledging the game start message and initializing the necessary variables and functionalities. It also continuously monitors button inputs from the player. Based on the button pressed, the system constructs and transmits corresponding game action messages to the game coordinator, indicating the player's actions (e.g., shooting spells). Additionally, the system monitors infrared (IR) sensor inputs for detecting hits from other players. If a hit is detected and certain conditions are met (e.g., a specific time interval has elapsed since the last hit), the system constructs and transmits hit confirmation messages to the game coordinator. The system ensures reliable transmission by employing message queuing and acknowledgment mechanisms.
The program is currently in the `STATE_START` state, indicating it's ready to start some operation.

- The program begins by checking if the first byte of the Bluetooth buffer (`BT_buffer`) matches the hexadecimal value `0x02`. This condition checks for a specific message type that the program is expecting.
  - If the condition is true:
    * It sets the first byte of the Bluetooth buffer to `0x01`, indicating an acknowledgment (ACK) message.
    * The program sends the acknowledgment to the laptop using the `BT_Send` function, transmitting 2 bytes.
    * The program increments the message ID (`activeID`) by 1, ensuring it stays within the range of 0 to 255.
- Next, the program checks for button inputs. If button 2 is pressed, it performs the following actions:

    - It sets the first byte of the Bluetooth buffer to `0x03`, indicating a shot sent message.
    - It assigns the `playerID` to the second byte of the Bluetooth buffer.
    - It assigns the value `2` to the third byte of the Bluetooth buffer, representing the button number.
    - The program sends the Bluetooth buffer with a length of 4 bytes to the laptop using the `BT_Send` function.
    - The program increments the message ID (`activeID`) by 1, ensuring it stays within the range of 0 to 255.
    - It calls the `spellPulse` function with parameters `UNSPECIFIED` and `TRUE`.
- If no button is pressed, the program calls the `spellPulse` function with parameters `UNSPECIFIED` and `FALSE`.
- The program checks if infrared (IR) detection is true.
    - If true:
        * It sets a flag to true.
- The program checks if the time elapsed since the last infrared detection plus 300 milliseconds is less than the current millisecond count, and if the flag is true.
    - If true and the condition `abs(IR_Count() - playerID * 3) <= 1` is false:
        * It calculates the modulo of `IR_Count()` by 3.
        * It sets the first byte of the Bluetooth buffer to `0x04`, indicating a shot received message.
        * It assigns the `activeID` to the second byte of the Bluetooth buffer.
        * It assigns the `playerID` to the third byte of the Bluetooth buffer.
        * Based on the modulo value, it calculates and assigns the opponent ID to the fourth byte of the Bluetooth buffer.
        * The program sends the Bluetooth buffer with a length of 4 bytes to the laptop using the `BT_Send` function.
        * The program increments the message ID (`activeID`) by 1, ensuring it stays within the range of 0 to 255.
    - It sets the flag to false.

3.7 Laptop State Machine

1. State 1: The system gets an expected number for connected Bluetooth modules from the user (standard input). Once it receives the expected number of Bluetooth modules, it transitions to State 2.

2. State 2: The system remains connecting/connected to Bluetooth modules until it detects the expected number of modules have all been connected successfully. Once the number of connected Bluetooth devices matches the expected number of modules, it moves to State 3.

3. State 3: In this state, the system assigns player IDs and sends packets with the assigned IDs. After completing this task, it transitions to State 4.

4. State 4: The system repeatedly sends ID packets until all Bluetooth modules have acknowledged the packets. Upon receiving acknowledgments from all modules, it moves to the Start of Game State.

5. Start of Game State: In this state, the system awaits incoming packets. If it receives one or more packets within a specified period, it transitions to the Handle Incoming Packets State.

6. Handle Incoming Packets State: Here, the system processes incoming packets. If a received packet is an acknowledgment, it removes it from the waiting-for-acknowledgment queue. If the packet is a "shot received" event, the system queues the receiver's ID, sender's ID, and timestamp. If the packet is a "shot" event, it queues the button number in the shot queue, along with the timestamp, and sends an acknowledgment. It then transitions to the Loop Queued Shots State.

7. Loop Queued Shots State: In this state, the system checks for corresponding receivers and organizes them into a separate queue for paired shots and receivers. It discards any packets that are too old. After organizing the shots, it moves to the Sort Pairs by Timestamp State.

8. Sort Pairs by Timestamp State: Here, the system sorts pairs by timestamp. Pairs older than a specified threshold or with opposite directions are awarded points and queued for sending back. If a pair is newer than the threshold and an opposite pair currently exists, it remains in the queue. The system then transitions to the Send Win/Loss State.

9. Send Win/Loss State: In this state, the system notifies the STM modules that haven't yzet acknowledged previously sent messages and returns to the Start of Game State.

3.8 Overall Integration

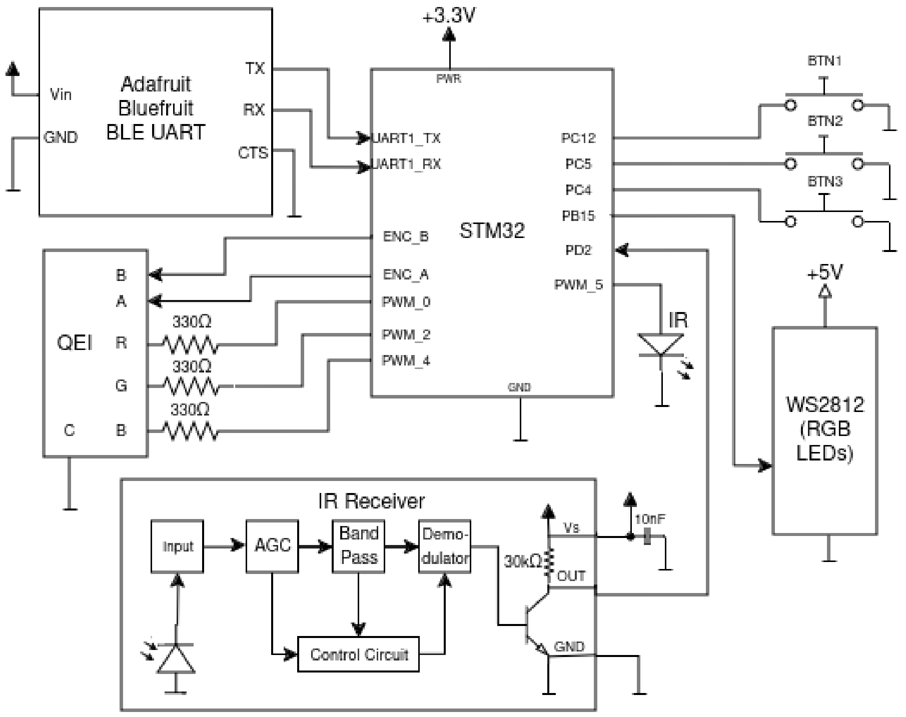


Figure 9: Electrical Schematic of overall circuit integration

As shown in the schematic diagram, each component has a specific location and role in the pinout of the Nucleo-64. Starting from the top left, the Adafruit's Bluefruit UART, a Bluetooth Low Energy (BLE) module, is connected to the STM32's UART1 pins, specifically UART1_TX and UART1_RX. These pins are responsible for transmitting (TX) and receiving (RX) data between the STM32 and the Bluefruit module. The CTS pin, which stands for Clear To Send, is connected to ground, indicating that the data is always clear to send from the STM32 to the Bluefruit module.

Just below the Adafruit Bluefruit, there's a Quadrature Encoder Interface (QEI). The QEI has six connections: C, B, A, R, G, and B. The C connection is grounded, while B and A are connected to the ENC_B and ENC_A pins respectively. This configuration is crucial for driving the pins for the encoder. Following this, the R, G, and B values are driven using 330Ω resistors. The values sent to these R, G, and B pins are determined by the Duty Cycle set to the Pulse Width Modulation (PWM) pins assigned.

Further below the QEI, there's an Infrared (IR) receiver. This receiver utilizes a band-pass filter and a photodiode, as explained in Section 3.1. The output from this IR receiver is directly inputted to pin PD2 on the STM32, which acts as a trigger pin.

To the right of the IR receiver, there's a WS2812 RGB LED strip. This strip consists of 22 LED modules connected in parallel. It operates with a 5V voltage input and uses the Serial Peripheral Interface (SPI) for communication. This strip is connected to the SPI2 section on the I/O shield, which in turn is connected to pin PB15 on the STM32.

On the top right of the schematic, there are three buttons labeled as BTN1, BTN2, and BTN3. These buttons are connected to ground as pull-downs, and their states are sent as inputs to the STM32 utilizing pins PC12, PC5, and PC4, respectively.

Overall, this schematic provides a detailed view of the connections and interactions between various components, all of which are ultimately connected to the STM32 MCU.

In terms of programming, the ultimate flow and integration of each component, including how they interact with the STM32 and how data flows between them, is described in detail in Section 3.6.1. This includes the specifics of how the UART, QEI, IR receiver, RGB LED strip, and buttons are programmed to work together to achieve the desired functionality.

4 Evaluation

The graph below depicts high time of the IR signal. We can see that it stays on for about 60ms. In order to deal with multiply edge triggers/noise, in our code we wait at least 100 milliseconds before checking if there is another edge. This way we only check for the first rising edge of the pulse. Since we know that there is a max delay of 100ms between pulses, then we can simply wait 160ms (high time plus low time) or more to confirm that we have received a full signal. To be safe we opted to wait 300ms. The oscilloscope readings below is how we checked if our IR receiver hardware was working. It was taken by reading the signal seen by the IR receiver from the IR emitter from about 18-22 ft away. In this case we expected to see the same number of rising edges as pulses sent which is essential for the wands to receive and emit.

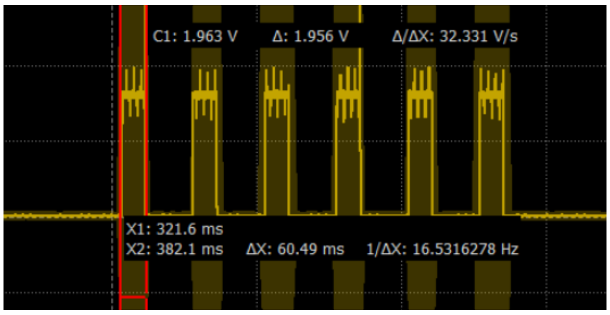


Figure 10: High edge to edge time of receiver

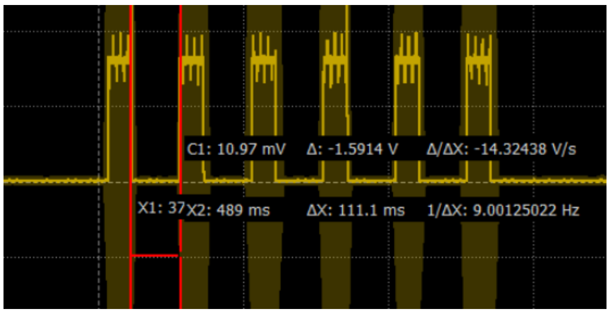


Figure 11: Low edge to edge time of receiver

Note: No data was collected on distance, as there was no visible signal loss observed within the parameters of our experiment. In simpler terms, the IR emitter and receiver maintained a clear line of sight well beyond the 18 feet constraint in our testing on opposite sides of a 22ft room, thus data collection was unnecessary.

Below is a wide view of the SPI signal. Although difficult to see, it's propagating a 24bit signal where if the high time is 721 ns, it's considered a 1, if the high time is 346 ns, then it's a 0. This is how the LED's know what color to change each LED. The data collected below is proof that our SPI library works in the way we are expecting. The figures below shows how each signal is sent in 24 pulses and how the high time of a 1 pulse and a 0 pulse is different:

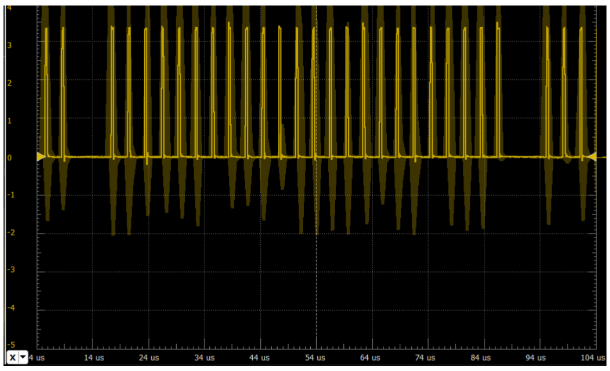


Figure 12: A raw capture of RGB data being sent to WS2812 LEDs leveraging the SPI transmission hardware measured with Analog Discovery 2

The figures below show the difference visually between a 1 and 0 pulse:

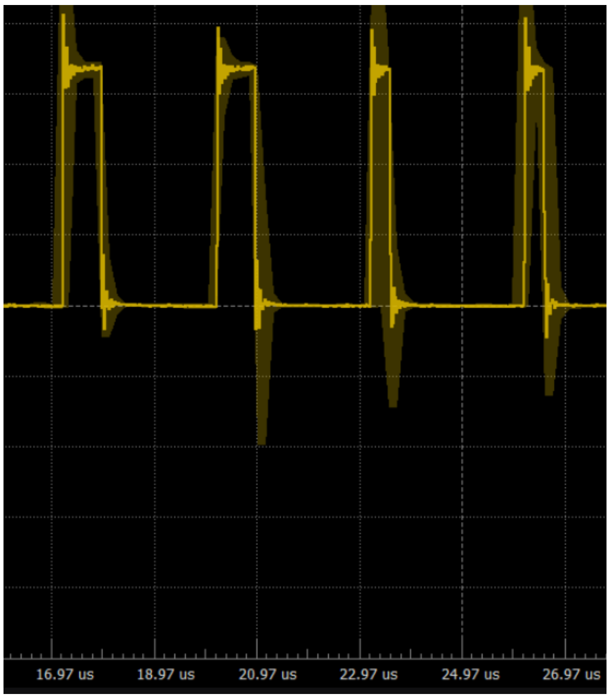


Figure 13: A close up of 8 bits of RGB data signal measured with Analog Discovery 2

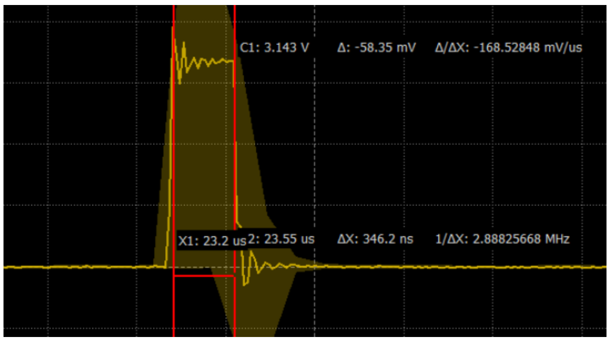


Figure 14: A zero bit transmission signal measured with Analog Discovery 2

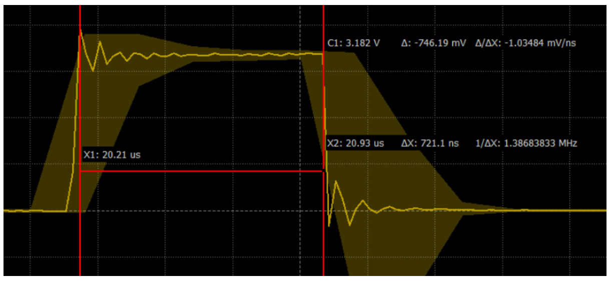


Figure 15: A one bit transmission signal measured with Analog Discovery 2

To evaluate our Bluetooth library, we measured the average time taken for each packet to successfully transmit from a laptop to the Bluetooth board, be read and handled by the STM32, and finally be sent back to the laptop. Over the course of 1000 iterations, the average time for this process was 99 milliseconds. As we were initially aiming for 100ms, we deemed this successful. Unfortunately, we faced a high rate of dropped packets in the process. Specifically, of the 715 packets we attempted over the aforementioned 1000 iterations, we experienced 169 dropped packets with a drop rate of 23.636%. This high drop rate was especially evident when running our full program, where many packets were dropped and needed to be resent. This would be an important issue to fix for anyone attempting to utilize our library, as it introduces a significant amount of delay during actual gameplay.

5 Discussion + Conclusion

In our project journey, half of us learned how to count and the other half learned their ABC's, all in bits. We've made significant progress towards achieving our objectives, though we acknowledge there's still room for improvement. With more time, we could have addressed many of the challenges and gameplay issues that arose during development. Such as figuring out the random increase in packet drops, shorting components in the wand, and RGB miscolorings. We have theories to what these could be, but due to the lack of time to debug these issues as they come up, we have the minimum viable product needed. Nevertheless, our project is a testament to our dedication and problem-solving skills.

By successfully implementing Bluetooth communication between the laptop and STM32, and creating an SPI library, we've laid a solid foundation for future projects to be built off of these libraries. Our main goal was to create a fun game similar to laser tag using wands, and while we've made good progress, there are areas where we can refine our work. These areas rely in the electronic wiring and more smooth transitions in our programming to ensure proper data flow in our state machines between systems.

Using two STM32's with IR wands, players can cast spells with the push of a button and see the result in the terminal on the computer. However, we encountered some gameplay issues that need further tweaking. One of the major challenges we experienced in gameplay was not knowing that IR worked through PLA, leading to leaky IR and therefore a lack of perfect directional IR signals. This was unfortunate, but we attempted to work around it with electrical tape. This made it work fairly well, but still had some bugs. To mitigate this, we implemented some software to avoid hitting yourself, but due to packet drop rates that would spike for unknown reasons, an IR LED burning out last minute, and some mishaps in soldering, we were unable to finalize debugging with our project (but we got very close).

Despite these challenges, our project showcases the potential of Bluetooth communication in this course, alongside the use of an SPI library. This introduces lots of future use cases where students could either develop this project in the future, or have a TA develop them further for the use cases of this course or other courses.

Alongside this, we did alter some goals after realizing that some things did not work as we expected. This involves the beacon sending varying IR signals in frequencies, which we altered to read number of pulses rather than frequencies; the encoder being able to cycle through different frequencies, this was altered for the encoder to only cycle through 8 different colors as we decided that the player ID assigned by the laptop would determine the number of pulses sent for ease of use; finally, the encoder taking input from Bluetooth module and changing colors altered to the player choosing a color from the QEI and having that define their player to the laptop, only in terms of coloring and allowing a state transition. These goal changes were in lieu of better design decisions when better understanding of the components occurred.

Although our project isn't perfect, it showcases our understanding of the sensors and sensor technologies that we've learned in this class. By focusing on the implementation of the sensors in this class, we feel that we have shown our understanding and had a good approach to this final project. We do wish we had chosen a little bit of an easier project, but we had fun with this project and we would definitely attempt doing it again given the time.

References

[1] NA. Ws2812 leds using spi.

[2] user: guitou. flash a st board with stlink and linux, 2020-07-23.