Button Remapping for GameCube Games such as Super Smash Bros Melee

ECE445 Final Paper Report

Michael Qian, Yeda Wu, Srikanth Yaganti

Group 14

TA: Evan Widloski

05/06/2020

Abstract

This paper describes our completed senior design project. The objective of this project is to allow GameCube players to remap their controller buttons to alternate button configurations. This functionality exists in many modern console systems such as the Sony PlayStation or the Nintendo Switch but was never developed for the GameCube. Being able to remap buttons has various uses for GameCube games since it allows users to customize each game's gameplay to their own preferences. Our final product consists of a PCB that would be connected between the GameCube console and controller as well as a phone app that would allow users to set their own button configurations.

Contents

1. Introduction

The project we have been working on is the “Button Remapping for GameCube” project. Our project goal is to take user inputs on a GameCube controller and remap them into alternative values according to some user specified remapping configuration. A few additional goals for our project include having negligible delay from the remapping, having an easy way to configure and reconfigure the remappings stored in the microcontroller, and having the ability to toggle between configurations. Beyond these objectives, we also prioritized minimizing the cost of production and the physical size of our product.

The block diagram in Figure 1 shows an overview of the functionality of our project design. It can be divided into four essential subsystems: the config selection, the config creation, the controller interface, and the console interface. The config selection subsystem consists of an array of buttons and LEDs that allow the user to reset, modify, and toggle between remapping configurations stored within the microcontroller. This subsystem is the primary interface with users and thus has been designed to optimize ease of use and clarity. The config creation subsystem represents the interface between the microcontroller and a phone app we designed which communicate with each other through a Bluetooth module. The controller subsystem and the console subsystem have the purpose of communicating between the microcontroller and the GameCube controller and console. Their main requirements are to match the original bus protocol of the GameCube and to ensure a minimum delay in the signal.

There are also the power requirements for this design. The GameCube console supplies an internal 3.43V and a 5V to the controller. The microcontroller and the config creation subsystem both require a 5V supply but the 5V supply from the console doesn’t provide enough current for all the components in our project. Therefore, an external 5V supply was required.

This project also includes a considerable software component. In order to enable button remappings, we wrote functions that can read and write in GameCube’s specific protocol. In addition to this, the functions must be able to interpret and respond to the various commands that the console sends and remap the responses of the controller. Furthermore, we wrote a software application to allow users to create configurations and send them over to the microcontroller.

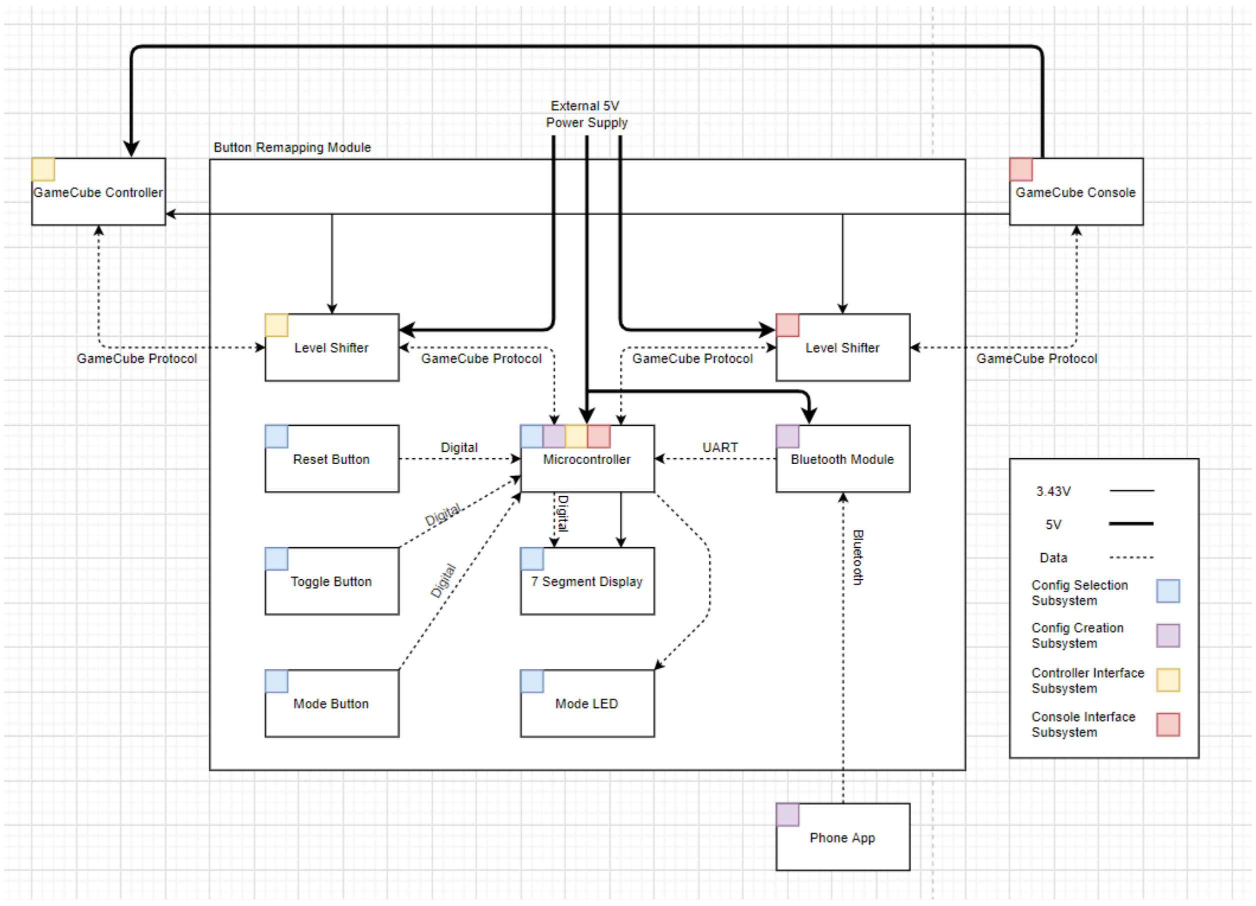

Figure 1. Top-level block diagram of the entire design

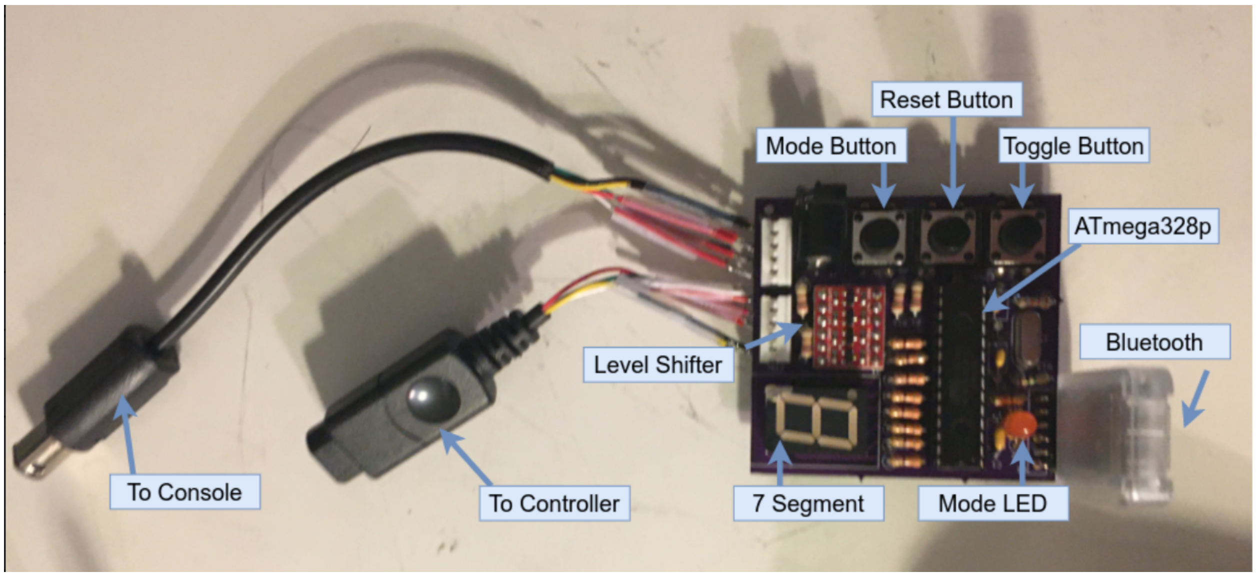

Figure 2. The finished product with labels.

2. Design

2.1 Design Procedure

2.1.1 Microcontroller

We had considered various different microcontrollers while designing our project but ended up choosing the ATmega328 after evaluating both the hardware and software requirements of our project. The microcontroller had to manage the communication between itself and the GameCube console, the controller, and the Bluetooth module. It also required enough IO pins to handle an array of buttons and LEDs. Additionally, the microcontroller must be capable of generating a 1MHz signal in order to replicate the GameCube bus protocol. Lastly, the microcontroller must have enough on-chip memory to store up to ten button remapping configurations and must be programmable in C and assembly. The ATmega328 checked all these requirements and had the additional advantages of being relatively inexpensive and available in a small, easy-to-solder package [1].

2.1.2 GameCube Read/Write Algorithms for Controller/Console Interface Subsystems

GameCube bits are in a non-UART format [2]. This requires us to write our own code for reading and writing bits. For reading bits, we had to decide between using pin change interrupts or busy wait loops to detect and read bits. For writing bits, we had to decide between using either timer interrupts or delay loops to keep signals high/low for set durations of time. The benefit of using interrupts is that other actions can occur while the interrupts are not being handled, and the detriment of using interrupts is if there is a non-negligible delay when entering and exiting the interrupt. The benefit of using loops is that it is more straightforward to write, but the detriment is that no other actions can occur while polling for a pin change. We ultimately chose to use loops for both reading and writing because there was too much delay between entering and exiting interrupts for reading and writing to be implemented on a 16 MHz microcontroller. In order to use interrupts, we would need a faster and costlier microcontroller.

Another design decision was to determine if we should use C or assembly to write the read and write algorithms. The benefit of using C is that it is easier to code using. The benefit of using assembly is that we would be able to accurately know the timings of every operation. We have decided to use assembly because we determined that having exact control over pins was more important than ease of coding on a microprocessor with a slow clock speed.

2.1.3 Remapping Algorithm

We also needed to determine the remapping algorithm for the controller. One method is to poll the controller with the microcontroller when the console polls the microcontroller, then remap the values and send it back to the console. This method is straightforward, but it may take too long for the microcontroller to respond back to the console. Another method is to use the microcontroller to poll the controller, remap the buttons, save that value, and send it immediately to the console when receiving the console polls command. We chose to implement the second method because we determined that the delay of the first method would be too long.

2.1.4 Config Selection Subsystem

We decided to have 3 buttons for the user. One button would toggle between configurations. Another button is for resetting the configurations. The third button is for switching between the 2 modes: the button remapping mode and the configuration editing mode. To determine if a button is pressed, we could either use pin change interrupts or poll for the values. Pin change interrupts are beneficial because the button press can be handled immediately when the button is pressed, but the downside is that the interrupts would be lost when interrupts are turned off in time-critical sections of code. Polling is beneficial when interrupts are off during most of the runtime. The downside of polling is that we will miss button presses if we do not poll often enough. We chose to use polling because, for the majority of the time, interrupts are turned off due to constantly polling for reads and performing writes. Through manual testing we determined that we poll fast enough to detect button presses.

2.1.5 Config Creation Subsystem

We decided that an easy way for users to create configurations is through their computer. Users can create a simple config.txt file on their computer where they can write up to ten of their button remappings. After creating their file, all users have to do is run the python script we have created to send their configurations to the button remapping module. This script communicates with the remapping module via Bluetooth, specifically through HC-05. When the user runs the script and sends config data, the module will parse the config data and store it in structs for each of the ten configurations. This allows users to easily switch between all of their configurations.

2.1.6 PCB

The main considerations taken while designing the PCB layout were to minimize the size of the board, to ensure ease of solderability, and to reduce the delay on the signal path between the console and the controller. We also followed many general PCB design standards outlined in the IPC-2221 while designing our board [3]. This includes decisions such as filling in all unused planes with ground to improve power dissipation and using the current requirements of signals to calculate appropriate trace widths as described by equation (2.1).

$$min.trace\ width = \frac{1}{(layer\ thickness)\times 1.38}\left(\frac{max.current}{0.048\times(max.temperature)^{0.44}}\right)^{1.38} \quad (2.1)$$

2.2 Design Details

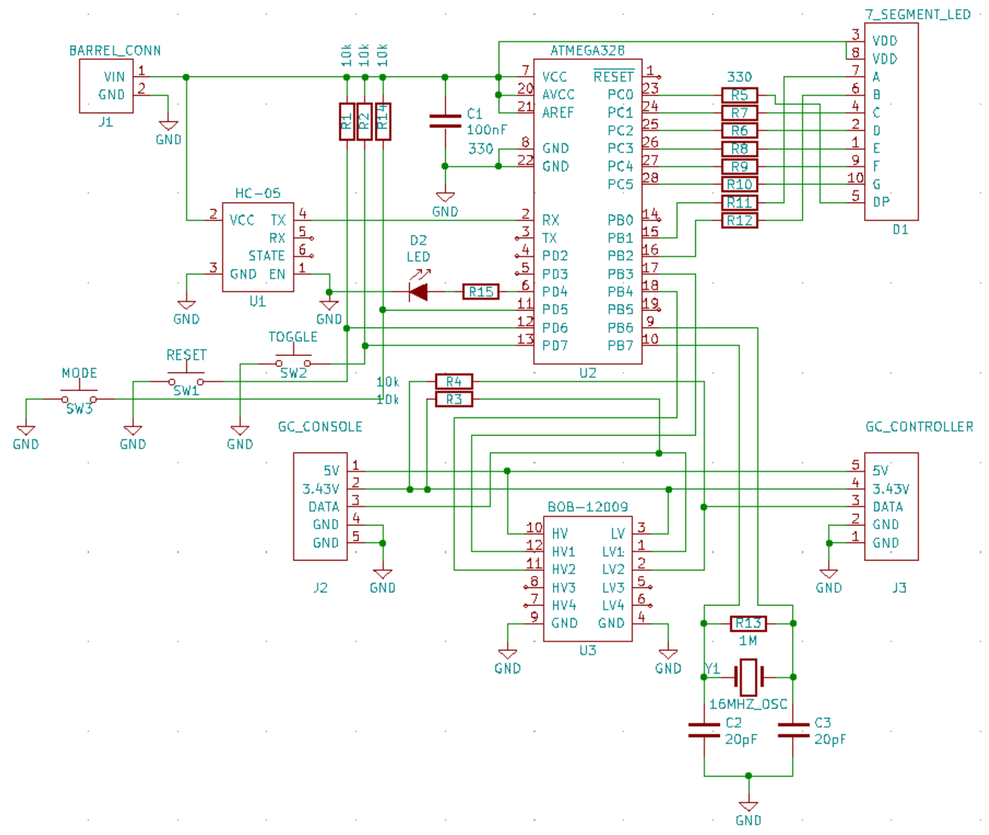

Figure 3. Schematic of the entire design

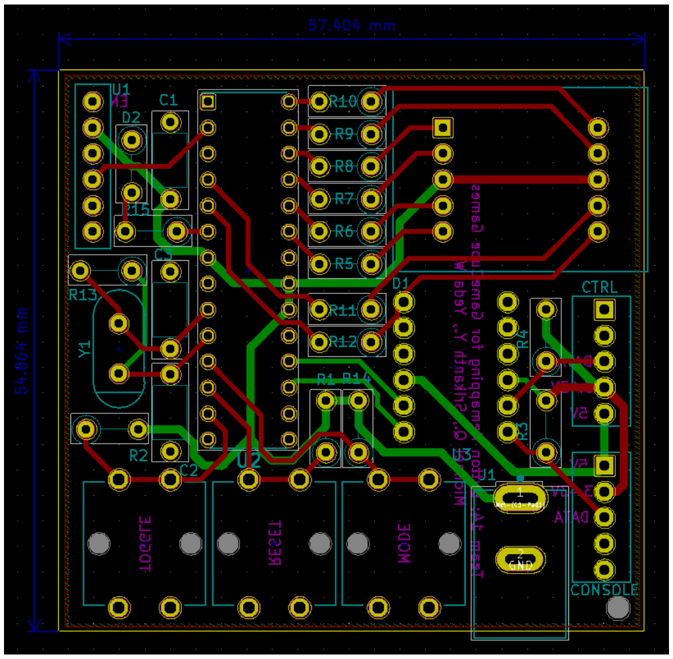

Figure 4. PCB layout of our design

2.2.1 Microcontroller
The microcontroller we are using is the ATmega328. Along with the two connectors and the Bluetooth module, it is also connected to a crystal oscillator which generates a 16MHz clock signal. This clock speed was chosen even though the data rate of the GameCube was 1MHz because we wanted to use as little time as possible with the remapping algorithm to reduce the delay. To support this clock speed, the ATmega328 needed to be powered with 5V since the performance of the microcontroller dropped at lower voltages and higher clock speeds [1]. Lastly, a decoupling capacitor was added to this chip to decrease the noise interference from the power supply.

2.2.2 Controller/Console Interface Subsystems

Level Shifters
The logic level of the GameCube bus protocol is 3.43V but the ATmega328 is powered by a 5V supply and thus produces signals that run at 5V. Therefore, the microcontroller cannot be directly connected to either the Gamecube console or controller in order to avoid damaging any of these components. The solution is to place a level shifter both between the microcontroller and the console and between the microcontroller and the controller. The specific level shifter we chose was the BOB-12009. We chose it because of its very low propagation delay of 5ns and because it had multiple channels which took care of both locations where level shifters were needed on a single chip [4].

GameCube Bits
Zero and one bits are represented in GameCube's unique protocol. A zero bit is low for 3µs and then high for 1 µs. A one bit is low for 1 µs and then high for 3 µs [2]. We must follow this practice if we want to write 0 and 1 bits. For reading bits, to determine if the bit is a 0 or 1, we can probe the line 2µs after the line goes low.

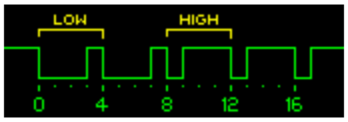

Figure 5. Zero and one bits for GameCube [2]

Interrupts vs Loop
We first tried determining if we could read/write using interrupts. We wrote some basic code to time how quickly we can jump in and out of an interrupt. The fastest square wave we achieved was 1.6 µs on a microcontroller with a clock speed of 16 MHz. This means that we would need a microcontroller that was significantly faster in order for the interrupt handling delay to be considered insignificant.

C vs Assembly
In order to achieve precise timings, assembly was used for the reading and writing of bits. This idea was inspired by NicoHood's Arduino Nintendo Library [5]. This is to guarantee that our code is waiting the exact amount of time necessary for the next pin read or pin write. The issue with using C can be illustrated in the example below. Supposed we have a loop to output 0 bits:

```
for (<initialization>; <condition>; <afterthought>) {
        set_pin_low();
        _delay_us(1);
        set_pin_high();
        _delay_us(3);
}
```

The issue with this code is that we hold slightly longer than 3µs due to the fact that we will be branching back to the start of the loop. In addition to this, we are not certain of how the code compiles to assembly so we are unsure of how to account for the <condition> and <afterthought>. If we wrote this in assembly, we can know exactly how many clock cycles each operation is, and thus account for the jump and any other logic by using the exact amount of NOPs necessary to hold a pin high/low for a certain amount of time.

Write Algorithm

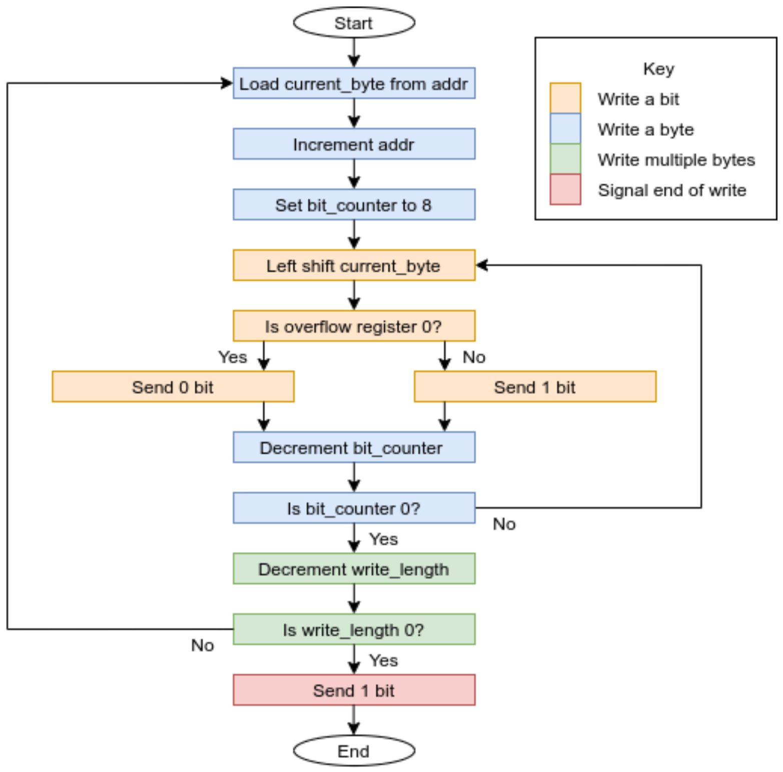

Figure 6. The write algorithm allows the microcontroller to send a specific byte sequence from a uint8_t array when the user specifies the addr and write_length.

Read Algorithms

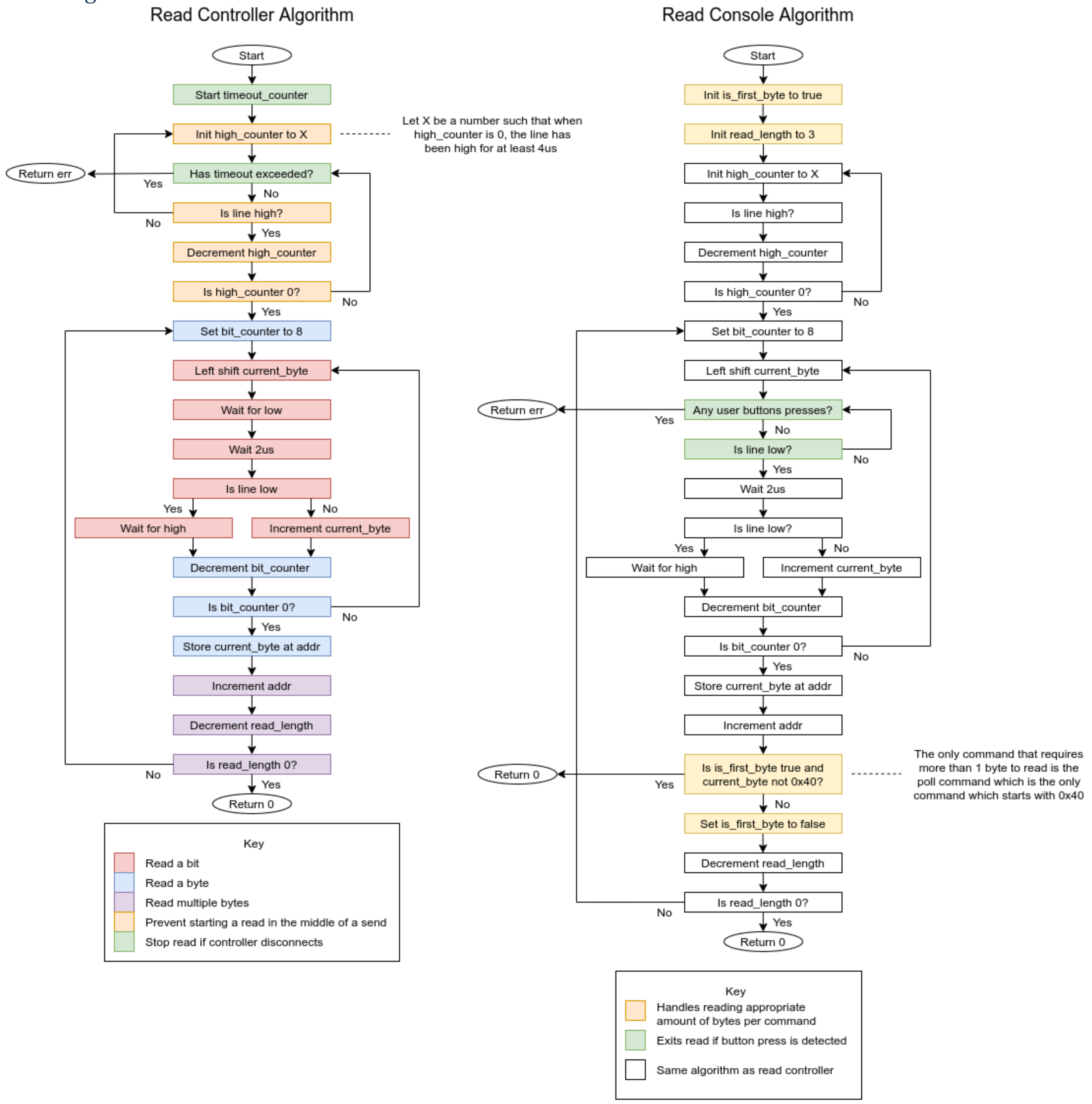

Figure 7. The read controller algorithm allows the microcontroller to send a specific byte sequence from a uint8_t array when the user specifies the addr and read_length. The read console algorithm differs from the read controller algorithm because the user doesn't specify the read length. Instead, the algorithm determines how much to read. It will read 1 byte for the init command, 1 byte for the origin command, and 3 bytes for the poll command [2].

2.2.3 Remapping Algorithm

The naive method requires the microcontroller to poll the GameCube controller and read the response before being able to respond to the console (see Figure 8). Polling the controller takes $4 * 24 = 72\ \mu s$.

This is because each bit takes 4 μs to send, and there are 24 bits in the polling request [2]. Reading the controller's response take $4 * 8 * 8 = 256\ \mu s$ because we must read 8 bytes of data from the GameCube controller. This means that the naive method has at least a 328 μs delay between the microcontroller receiving a poll command and responding with the remapped buttons. This takes too long as normally a GameCube controller responds to the console in roughly 4-5 μs [2].

For the chosen method, we are able to respond to the console directly after detecting the read. Steps 1 and 4 take 72 μs each because they are both poll commands. Steps 2 and 5 take 256 μs because they are both poll responses. In total, all steps but step 3 will require $72 + 72 + 256 + 256 = 656\ \mu s$. This means that we can handle any game that polls slower than 656 μs without skipping poll commands.

A poll command occurs at variable rates. For Super Smash Bro's Melee, the rate is fixed at 8 ms, but for other games it may be different. For example, the GameCube options menu polls from 0.7 ms to 19 ms [5]. That is why we chose to conservatively poll the controller right after the console polls the microcontroller. This way, we can always guarantee that the microcontroller won't be communicating with the controller when the console tries to communicate with the microcontroller.

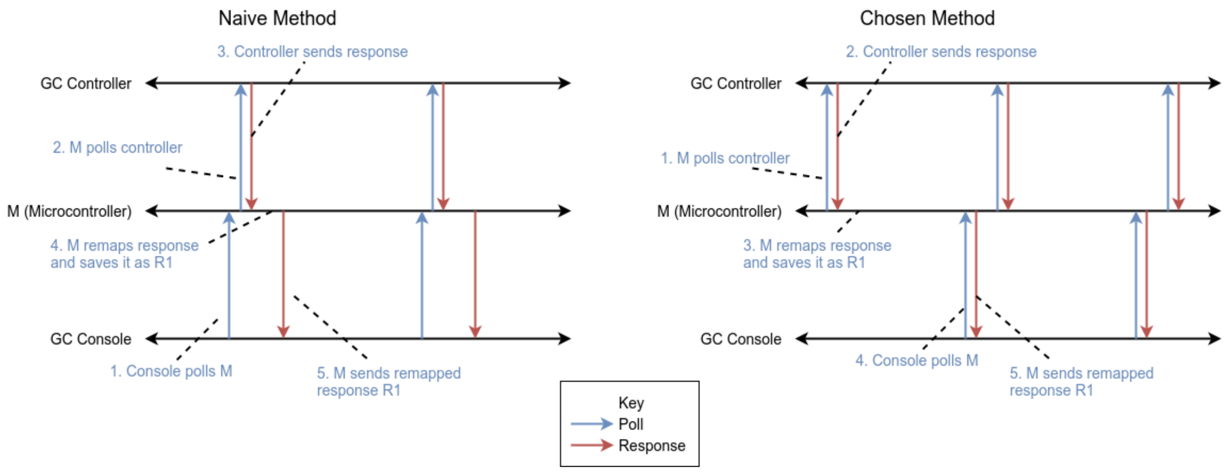

Figure 8. Comparison of polling methods.

2.2.4 Config Selection Subsystem

This subsystem is integral for users to create their own button configurations and for setting and sending configurations to the button remapping module. The most important part of this subsystem is the communication between the python script, which users can use to send their configs, and the module. In order to achieve proper communication we designed a simple communication protocol. Basically, when sending configuration information from the script we send a string of bytes. First byte always represents how many bytes are left to read. The next byte specifies which config's button remapping we are sending followed by twelve bytes of button data in order seen in the config struct below. This way when the button remapping module receives the data, we can easily parse it and store that config information on the microcontroller.

Storage of these configurations is done simply via a struct for each configuration. After parsing the data from the python script, we store button data into the corresponding configuration struct.

```
#define BUTTON_START   0
#define BUTTON_Y       1
...
#define BUTTON_D_RIGHT 10
#define BUTTON_D_LEFT  11

typedef union {
    uint8_t raw[NUM_BUTTONS];
    struct config {
        uint8_t start;
        uint8_t y;
        uint8_t x;
        uint8_t b;
        uint8_t a;
        uint8_t l;
        uint8_t r;
        uint8_t z;
        uint8_t d_up;
        uint8_t d_down;
        uint8_t d_right;
        uint8_t d_left;
    } config;
} Config_t;
```

For users to create configurations, all they need to do is create a config.txt file. In the config.txt file, they can create up to 10 configurations. To create a config, first the user has to state the number of the config(0-9) and then in the button order above write “button : new button value” on each line. For example, “A:B” means button A will be mapped to button B in this config. Users can have all 10 configs one after the other just by separating each config by their numbers.

Our polling algorithm requires us to constantly read and write. The reads have active waits and writes have delays. These sections of code require interrupts to be turned off so using pin interrupts for detecting button presses would not be possible. We instead chose to poll for button presses between remappings and during active wait of detecting a console command.

Buttons and LEDs

We needed three buttons to adjust the configurations of the microcontroller. We opted for 12mm push buttons for this task because of their small size and durability. We needed LEDs to display the current configuration being used and the current mode of the microcontroller (the button remapping mode or the configuration editing mode). We used a red 7-segment LED and a red radial LED to accomplish these. We determined that 330Ω resistors were needed to produce the desired brightness (10mA of forward current) for forward voltages of 2V [6]. This was determined using equation (2.2).

$$R = (V_{out} - V_f)/I_f = (5V - 2V)/10mA = 300\Omega \qquad (2.2)$$

Bluetooth Module

The bluetooth module we have selected is the HC-05. It converts the incoming bluetooth signal into a UART interface with a programmable baud rate [7]. This makes it very compatible with the ATmega328. Since the bluetooth communication would only be in one direction (from the phone app to the microcontroller), only the TX pin of the HC-05 needed to connect to the microcontroller as seen in Figure 3.

2.2.5 PCB

The final design of our PCB (shown in Figure 2) is a two-layered board measuring 57mm by 55mm. The board is compacted to close to its absolute minimum size given the components we are using which will reduce the overall cost of fabrication. All the components we use are through-hole in order to ensure solderability. We also added a structural drill hole near the console/controller connectors to fix the connector wires to the board with a zip tie in order to add some structural support. To minimize the delay between the connectors and the microcontroller, the minimum trace length was used and no vias were used thus decreasing the resistance of the trace and the propagation delay of the signal path.

2.2.6 Power

Most of the power needs of this project come from the microcontroller and the Bluetooth module which require up to 7mA and 40mA respectively [1] [7]. The microcontroller has an operating voltage range of 3.3V to 6V but it is recommended to at least have 5V when using a 16MHz oscillator. Thus, the entire design requires a conservative estimate of about 50mA of current at 5V. The GameCube console supplies a 3.43V and a 5V power to the controller. The 5V from the console has a current limit of 30mA [2]. Therefore it cannot supply power to the rest of the design. Therefore, an external power supply in the form of a barrel connector and a USB cable was necessary.

3. Verification

3.1 Write Verification

We connected the microcontroller to one of the console's ports. Then, we open up Super Smash Bro's Melee and enter a game. When the console sends the poll command, the microcontroller sends the "jump" command. We then verified that the character jumped on the screen.

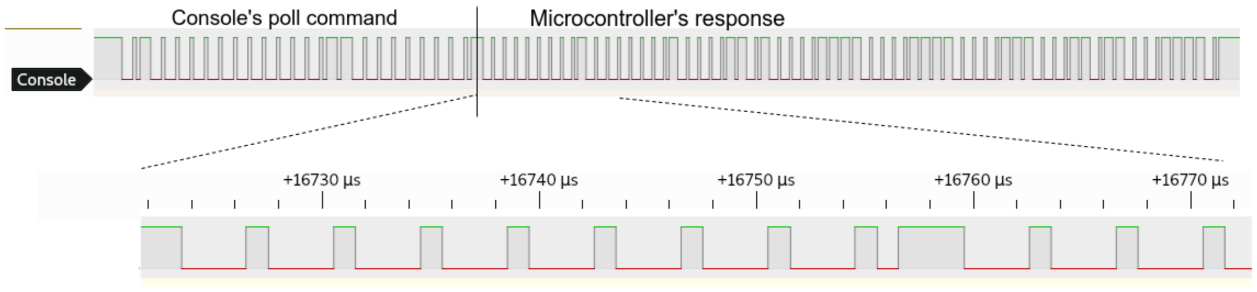

Figure 9. Write verification through using a logic analyzer. When zooming into the microcontroller's response. it can be seen that 0 bits are written as 3µs low then 1µs high, and 1 bits are 1µs low and 3µs high. Additionally, at the end of the response, it can be seen that the last bit sent is a 1.

3.2 Read Controller Verification

This verification was performed after verifying that we can properly write 0's and 1's. We used 2 microcontrollers. One microcontroller would output a fixed length byte sequence once to the second microcontroller with bits in the GameCube protocol. The second microcontroller would know the length of the byte sequence and read it in. If the value that the second controller read was correct, it will turn on an LED.

Read controller was also verified by having the microcontroller sit between the controller and console. The console would poll the microcontroller and microcontroller poll the controller like in the polling timings explained later. Then, we would perform no remappings and send the last poll response the controller outputted to the console. The controller acted normally when we started and played a game.

3.3 Read Console Verification

We set up 3 LEDs for the 3 commands that the console can send: init (0x00), origin (0x41), and poll (0x400302) [8]. Each LED would only light up if a specific command was read. When we plug in the microcontroller to one of the console's ports, we see 1 LED light up quickly for the poll command, then 1 LED light up quickly for the origin command, and finally 1 LED stay lit for the poll commands.

The console sends the poll command which also indicates that the controller was detected and initialized. If the console did not receive a valid response for the init command, it would not send the origin command, and if the console did not receive a valid response for the origin command, it would not send the poll commands.

3.4 Remapping Verification

We verified that the naive method of remapping doesn't work because the microcontroller responds too slowly to the controller.

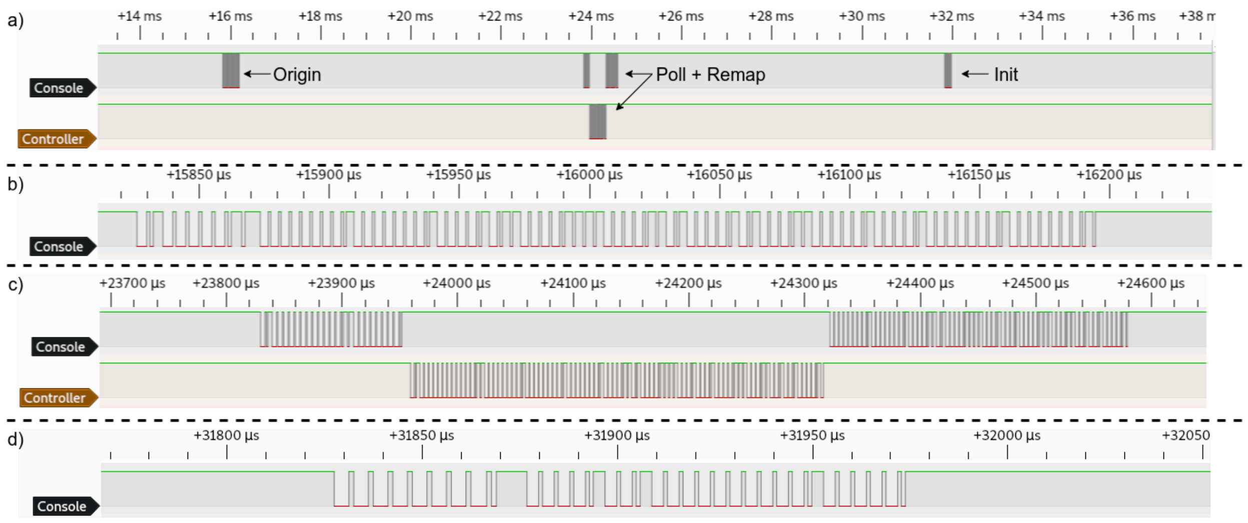

Figure 10. The console initializes the microcontroller and sends out the origin and poll commands. Unfortunately, the microcontroller's poll response was too slow, so the console sends out an init command because it believed that the microcontroller disconnected from the console (a). Parts b, c, and d show close ups of the origin, remapping, and init sequences respectively.

To verify that the second remapping method worked, we tested the remapping unit with a game and also probed the logic lines (Figure 11).

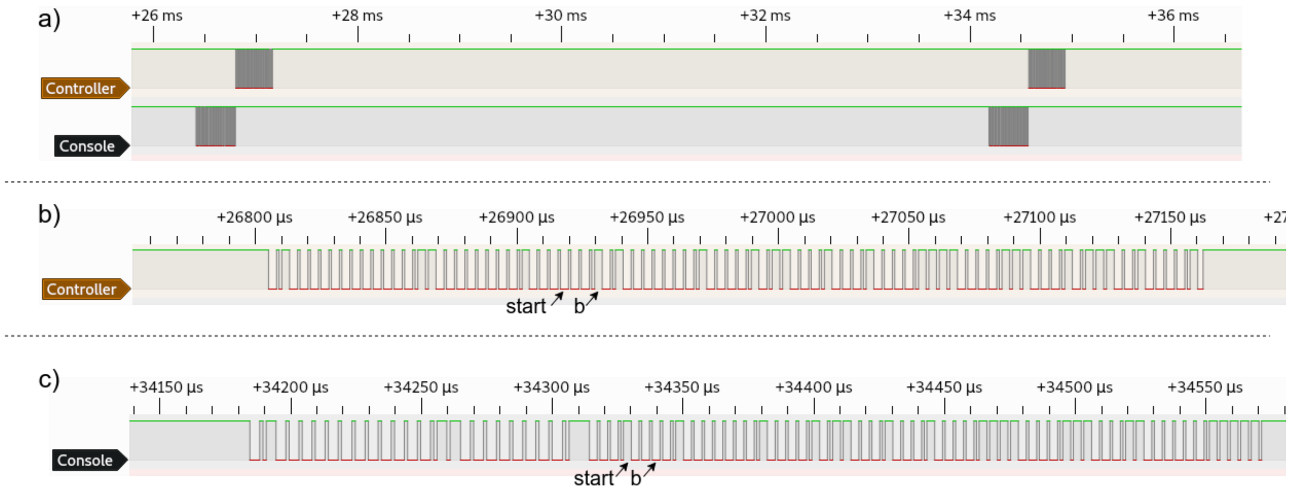

Figure 11. Console and controller logic lines were probed when the user decided to remap the B button to perform the action of the Start button (a), and it is clear that we alternate between the console sending a poll and receiving a response from the microcontroller and the microcontroller sending a poll and receiving a response from the controller. When observing the controller line, it is clear that the B button is the only button that is pressed (b). When the microcontroller responds to the console, the Start button is the only button that is pressed (c). This indicates that the B button was remapped to the Start button, which was expected.

3.5 User Buttons Verification

When we press the toggle button, we notice that the 7 segment LED changes to the next number and the configuration has changed to the next configuration. When we press the reset button, the 7 segment LED turns off and the buttons are now in the default configuration. When we press the change mode button, the LED for the mode toggles on/off, and we enter the other mode on the controller.

4. Costs

At $50 per hour and 10 hrs per week for 3 people for 16 weeks, we expect this to result in $24,000 for the labor cost. Below is the cost breakdown for one manufacturing one part:

Part	Quantity	Cost (prototype)	Cost (bulk)
ATmega328 Microcontroller	1	1.90 (Digikey, ATMEGA328-PU-ND)	1.58 (Digikey, ATMEGA328-PU-ND)
Gamecube extension cord	1	5.85 (Amazon, company: Mizar)	2.75 (Amazon, company: Pegly)
HC-05 Bluetooth Module	1	19.14 (Digikey, 1738-1164-ND)	2.58 (DHgate, seller: Tenypure)
Assorted resistors, capacitors, buttons, crystals, led	1	3.00 (Digikey, est.)	0.25 (Digikey, est.)
7-segment Display	1	1.24 (Digikey, est.)	0.83 (Digikey, est.)
5V Power supply	1	4.69 (Amazon, company: Yosoo)	2.30 (DHgate, seller: Vizgiz)
Level shifter	1	2.95 (Sparkfun, BOB-12009)	0.54 (Digikey, 296-21929-2-ND)
PCB	1	2.30 (PCBWay)	0.13 (PCBWay)
Total		**41.07**	**10.96**

In total, if we wish to create 10 parts, then the overall development cost will be **$24,410.70**.

5. Conclusions

Overall we believe we had a very successful project. We can successfully communicate the GameCube console and GameCube controller using our button remapper module as a middle man. We can properly read and write to the GameCube controller to mimic controller communication with the console using our button remapper module. We are able to successfully take the button inputs we read from the controller and remap the button inputs quickly to ensure that there is no perceivable delay in communication between the GameCube controller and the GameCube console. We are also successfully able to read and write to the GameCube console mimicking the GameCube communication protocol. In addition users can easily create configurations by editing config.txt and running the python script to send user configs to the button remapper module. We believe that the simplicity of our button remapper module and creating of configs allow anyone to quickly understand and utilize our product.

Next steps for our project will be to enable remapping for more input data such as joysticks. We believe that we can expand this project to remap data to reduce sensitivity of the joysticks, inverting the joystick movements, and even using joystick movements as button inputs. In addition we believe expanding the user config creations to phones is the next step for setting configurations on the button remapping module. Finally we would upgrade the display on the button remapping module to show the button remaps for the current configuration that is being used.

In terms of ethics, we hold responsibility for our project, which is the first rule in the IEEE Code of Ethics [9]. Additionally, our product could introduce a situation within professional gaming if the device is used when non-Nintendo/performance-enhancing devices are not allowed. This would be an unethical use of our device. In addition to this, it may be possible for people to tamper with our microcontroller such that certain button presses can lead to button macros. These two situations violates the IEEE Code of Ethics #9 because, if used in such a way, the trust in and reputation of professional gamers will be harmed [9].

6. References

[1] "8-bit Atmel Microcontroller with 16/32/64KB In-System Programmable Flash," ATMEL. [Online]. Available: https://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf. [Accessed: 30-Mar-2020].

[2] "Nintendo Gamecube Controller Protocol," Nintendo Gamecube Controller Pinout. [Online]. Available: http://www.int03.co.uk/crema/hardware/gamecube/gc-control.html. [Accessed: 30-Mar-2020].

[3] "Generic Standard on Printed Board Design," *IPC*. [Online]. Available: http://www-eng.lbl.gov/~shuman/NEXT/CURRENT_DESIGN/TP/MATERIALS/IPC-2221A(L).pdf

[4] "BSS138 N-Channel Logic Level Enhancement Mode Field Effect Transistor," On Semiconductor. [Online]. Available: https://www.onsemi.com/pub/Collateral/BSS138-D.PDF. [Accessed: 30-Mar-2020].

[5] NicoHood. Arduino Nintendo Library. (Version 1.2.1) [Source code]. https://github.com/NicoHood/Nintendo. [Accessed: 30-Mar-2020].

[6] "Model No.: YSD-160AR4B-8" CHINA YOUNG SUN LED TECHNOLOGY CO.," LTD. [Online]. Available:https://cdn.sparkfun.com/datasheets/Components/LED/YSD-160AR4B-8.pdf. [Accessed: 30-Mar-2020].

[7] "HC-05 -Bluetooth to Serial Port Module," ITeadStudio. [Online]. Available: http://www.electronicaestudio.com/docs/istd016A.pdf. [Accessed: 30-Mar-2020].

[8] Dowty, M. (2004). Unicone Gamecube Emulation [Source code]. http://svn.navi.cx/misc/trunk/wasabi/devices/unicone/fpga/gamecube/gamecube.v. [Accessed: 30-Mar-2020].

[9] "IEEE Code of Ethics," *IEEE*. [Online]. Available: https://www.ieee.org/about/corporate/governance/p7-8.html. [Accessed: 30-Mar-2020].