**Frame Perfect: Team 21**



|  |  |  |
| --- | --- | --- |
| *Team Members (left-to-right on picture, above)* | *Class No.* | *Lab Div* |
| Eric Colter | 4939-C | 005 |
| Jordan Huffaker | 0616-H | 005 |
| Diego De La Fuente | 8720-D | 005 |
| Kyle Diekhoff | 4825-D | 005 |

|  |  |
| --- | --- |
| *Report/Functionality Grading Criteria* | *Points* |
| Originality, creativity, level of project difficulty | 20 |
| Technical content, succinctness of report | 10 |
| Writing style, professionalism, references/citations | 10 |
| Project functionality demonstration | 20 |
| Overall quality/integration of finished product | 10 |
| Effective utilization of microcontroller resources | 10 |
| Significance of individual contributions\* | 20 |
| *Bonus Credit Opportunities* | *Bonus* |
| Early completion | 0.5% |
| PCB for interface logic | 2% |
| Poster (required for Design Showcase participation) | 1% |
| Demo video (required for Design Showcase participation) | 1% |
| Design Showcase participation (attendance required)\* | 1% |

##### \**scores assigned to individual team members may vary*

|  |  |
| --- | --- |
| *Grading Rubric for all Criteria (Including Bonus)* | *Multiplier* |
| *Excellent* – among the very best projects/reports completed this semester | 1.0 - 1.1 |
| *Good* – all requirements were amply satisfied | 0.8 - 0.9 |
| *Average* – some areas for improvement, but all basic requirements were satisfied | 0.6 - 0.7 |
| *Below average* – some basic requirements were not satisfied | 0.4 - 0.5 |
| *Poor* – very few of the project requirements were satisfied | 0.1 - 0.3 |

**TABLE OF CONTENTS**

|  |  |
| --- | --- |
| 1.0 Introduction | 3 |
| 2.0 Interface Design | 4 |
| 3.0 Peripheral Utilization | 6 |
| 4.0 Software Narrative | 8 |
| 5.0 Packaging Design | 10 |
| 6.0 Summary and Conclusions | 11 |
| 7.0 References | 12 |
| Appendix A: Individual Contributions and Activity Logs | 13 |
| Appendix B: Interface Schematic and PCB Layout Design | 22 |
| Appendix C: Software Flowcharts | 25 |
| Appendix D: Packaging Design | 27 |

1. **Introduction**

Frame Perfect is a gamecube controller peripheral which allows the user to provide their own input or allow choice of several pre-made combos referred to as macros. This can be achieved by asserting the macro button and selecting a macro from a given scheme. When the macro button is asserted the output of the gamecube controller is severed and the data shifted onto the 64-bit shift register. From here the shift register moves the data to the GAL device which converts the data into data the gamecube would recognize. But, if the macro button is not asserted the controller can still send signals through the shift register as normal. This allows the user to choose between playing normally or using an advanced frame perfect combo.

The interface leader of Team 21 was Eric Colter. He designed the circuit which allows the user to sever the gamecube controller output and assert the macro output. Once this was complete he constructed the initial design on a breadboard. Then he designed the eagle schematic for the PCB and soldered all the components to the PCB.

The peripheral leader of Team 21 was Diego De La Fuente. He interfaced the TIM, and SPI to shift out data to the 64-bit shift register. These communicated with a GAL device to convert the signal from the shift register into a waveform which the gamecube understands. He also utilized the COP watchdog timer and the IRQ in edge sensitive mode in order to be able to the read the bits coming from the controller inputs and respond accordingly .

The TDP of Team 21 was Kyle Diekhoff. He collaborated with the members of the team to find times which worked for everyone. Throughout the project he acquired various materials such as wires and packaging that were needed to complete the project. He also helped to debug the hardware and software of the project.

The software leader of Team 21 was Jordan Huffaker. Before development of the software he created several flowcharts to illustrate to the team how each part of the code was intended to work together. Then he continued to develop the code and set up the source control so that each member could easily obtain and change various bits of code

1. **Interface Design**

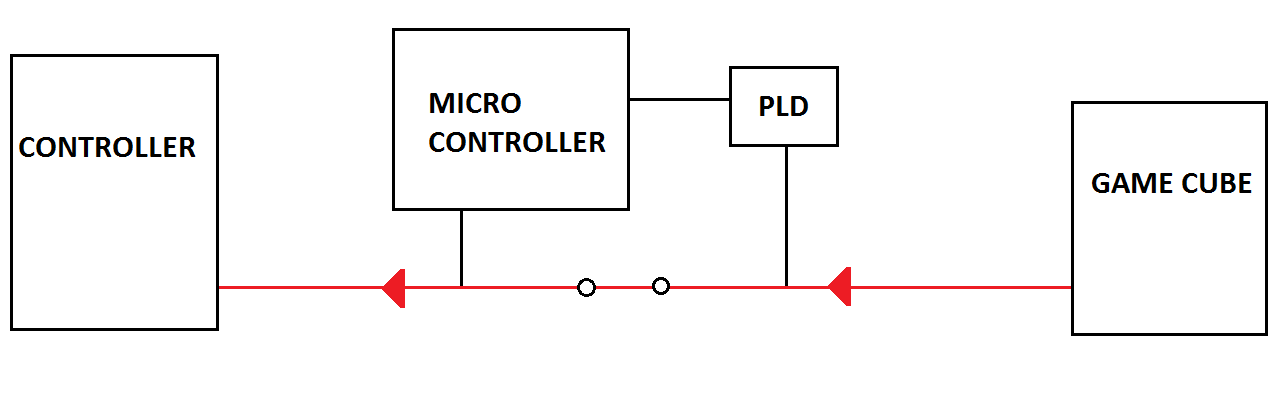
A few devices had to be interfaced with in the design, including the Nintendo GameCube, the GameCube controller, a PLD unit composed of two GAL22V10s and a 64 bit shift register, and finally a transmission gate.

In normal operation the GameCube and controller communicate with one another using a half duplex master slave protocol on a single data line. First, the GameCube sends 3 bytes to request for data, and after a 3μs delay, the controller responds with 8 bytes containing the joystick positions and which buttons are pressed [1]. The GameCube asks for data about once every 8ms. A zero bit is represented with 3μs low and 1μs high, while a one is 1μs low and 3μs high [2,3].

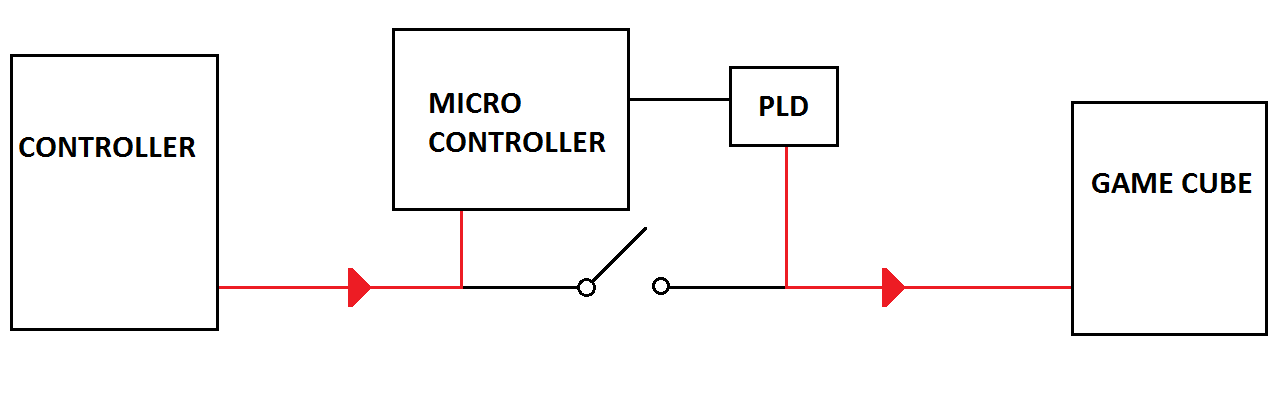
Our design places a transmission gate in the middle of the data line used by the controller and GameCube. The microcontroller is configured to read data from the controller side of the line, and the PLD unit is configured to output data on the GameCube side of the line using a pull down transistor. The microcontroller may send data to the shift register in the PLD unit, and may also send a signal to one of the GAL units to tell it to begin outputting data on the line.

When the GameCube requests data, the line is connected and its signal is allowed to go through to the controller (figure 2.1). Within the following 3μs delay, the transmission gate then disconnects the line. The microcontroller also sends a signal to the PLD to tell it to begin outputting the data previously loaded into it. After the 3μs delay the microcontroller reads the data from the controller, while the PLD simultaneously outputs data to the GameCube (figure 2.2). The transmission gate is then connected again. The microcontroller now has 8ms to interpret the packet from the controller, and make any necessary changes. This new, possibly altered packet is then shifted out the the 64 bit shift register in the PLD unit for the next communication (figure 2.3).

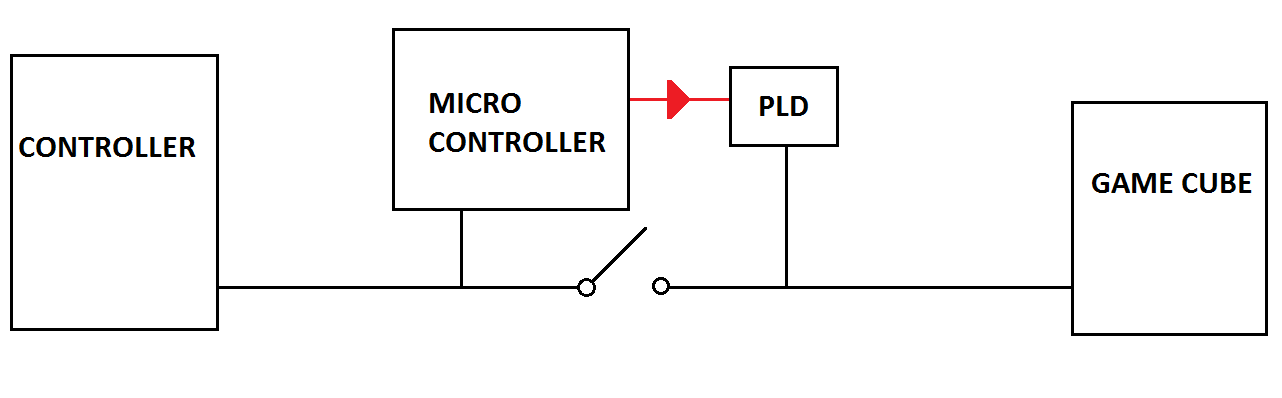
The PLD unit consists of two GAL units and a 64 bit shift register. One GAL operates as a 7 bit counter, and the other as a wave-form generator. The microcontroller directly communicates with the wave-form generator and sends a short pulse to tell the GAL to begin outputting data. The wave-form generator uses a 1MHz clock signal (1μs) , and internal 2 bit “phase counter” to translate the 64th bit of the shift register into the correct 4μs wave form on the data line. When the phase counter overflows, the shift register then decrements the counter and clocks the shift register to get the next bit. This process continues until the counter reaches zero, and output stops. Additionally, the protocol requires that a 65th bit (always a 1) is sent out. The counter will indicate to the waveform generator if it is on the final bit, so it can output a 1 and ignore the shift register.



*Figure 2.1: The Gamecube requests input from the controller*



*Figure 2.2: The Microcontroller reads data from the controller while the PLD outputs the previous packet to the GameCube*



*Figure 2.3: the Microcontroller loads the next data packet into the PLD*

# **Microcontroller Resource Utilization**

For our project, we utilized the TIM and SPI peripherals. Other on-chip devices that were used were the COP watchdog timer from the CRG, the IRQ in Edge-Sensitive mode, as well as the flash memory in order to store our macros. Each one of these resources had a specific and essential part to our design and are what enabled the software and hardware to communicate with each other in the most efficient manner possible.

For our peripherals, the TIM and SPI were specifically used primarily to aid with how we are pushing out data from our microprocessor. The TIM module was used in output compare mode to toggle port T pin 7 to create a square wave with a period of 1μs. This square wave was used as a clocking signal to the two GAL chips, so they could create the appropriate waveform for the 1 and 0 bits with the correct timing. The SPI on the other hand is used in master mode for the microprocessor with the shift register being the slave. Once the controller data has been processed, and possibly edited, by the software, the SPI outputs this data to our 64-bit shift register device at a rate of 6 Mbps. Together, the SPI and TIM modules provide the output unit (The two GALs and the shift register) with the data to output to the GameCube, as well as a clock that is timed to allow it output the correct waveform to represent the data.

The COP and the IRQ were other extremely important on-chip parts that we utilized in our design. The COP watchdog is something that we decided to utilize when we were beginning to decide how we can take our device and be able to utilize it with any gamecube controller and console set at any moment. Basically, it allows the device to resynchronize if something unexpected happens, such as missing a bit, or a controller being disconnected and reconnected. How it does this is by utilizing its abilities to reset the device when the set watchdog timer runs out. In our software, the watchdog is set and enabled to around 2 ms every time in our initializations before our interrupts are enabled and the IRQ begins to read the controller-console communication. Because of this, if the IRQ is never finished (which can come from a software error, controller is pulled out, controller hasn’t initialized when put in a Gamecube, or a Gamecube is just turned on) then the code never makes it back to our main function, where we reset the COP by sending the pulse of 0x55 and 0xAA to the reset register (ARMCOP) and thus resets back to initialization.

The IRQ was used for the entire reading process of the controller-console communication in general. How it does this is by first its initialization to Falling edge mode via setting INTCR\_IRQE to 0. Once this is done, interrupts are set and we get the first falling edge when the console-controller signal is reading through PTT4. Once inside the IRQ interrupt, we count the first 24 bits of data which are the console request for data and afterwards we open the transmission switch to interrupt the console-controller signal. For the next 64 bits, we wait for a falling edge and apprx. 1.5μs to read whether the controller signal is a 0 or a 1, which is necessary because the logic 0 or 1 from the console and controller communication is not just a flat 0 and non-zero respectively so at 1.5 us we determined was a point we could read in the normal digital way. Once this is over, the output from the GAL was determined to be complete, and the filled string will go through the software to determine whether or not this signal itself or a macro process will happen .

Finally, we utilized the flash memory to save our macros. The reason we did this was because we couldn’t fit all of our macros in the 2K of Ram that our microprocessor held. We were able to store it in flash through by making the structure that was holding the char\* into a const, which automatically makes it stay in that set of memory.

1. **Software Narrative**

The software begins with the initialization of different registers and values for the peripherals and on-chip objects. These include the TIM and SPI for communicating with the GAL and shift register and had to do with outputting the correct controller data in general. This also has to do with initializing the COP in order to allow the program to reset whenever you plugged in or out, as well as the SPI which is what reads the controller-console data. These initializations are all done in a file called microcontroller.c which includes our initialization function.

After everything had been initialized, the next step in our code is to receive and read the input from the Gamecube Controller, which is read during the IRQ interrupt service routine. When the service routine is completed, a flag is set so we check through our main control code whether or not we are in a frame of macro somewhere. At this point if we are in the middle of a macro, what we do is we grab the next frame of the macro, which are actually located in another .c file whose logic will grab the next Frame and return a value flag that will allow the original control file to know whether or not that the macro had ended and to get ready to read the next input to either output the given input packet or to enter into a new macro mode. However, in the middle of a macro and in the logic of our macro.c file, we also pass the value of the next input packet because we might want to give the user to chance to input a specific value in the middle of the macro. This is done via setting a input bit in one of the frames of value to ‘129’, which acts as a mask that asserts the inserted input’s corresponding bit into the macro frame bit.

The other option is if we were never in a macro to begin with. If that is the case, the first thing we check is whether the specific macro button has been pressed in the previous input we have received. If there isn’t then the only thing we check before just returning this input frame is if any of the D-pad buttons had been pressed. If this is the case, then the scheme will change, which changes what buttons do which specific macros, and all of this is controlled in another .c file that specifies the layout of every scheme (3 in all). But if there is a new macro button, we check which other button is pressed as well that will cause a macro (we have a hierarchy if multiple buttons are pressed). If one of those buttons are active, then the macro state for the output begins, and based on the scheme, the button combination will activate a specific macro and a flag is set stating our program is in macro mode and won’t leave macro until the macro.c file clears said flag. It will start by getting the first frame of the macro which will call the macro.c file which will determine the location of the first location in the entire structure. Any time a new input is called, it will not be used unless specific masked values in the frame of a macro call for it and this will be the case until the macro finishes. Finally, this loop of checking for macros or outputting the initial input will continue until either the controller is pulled out of the console or the console is turned off (no power reaches the controller).

1. **Packaging Design**

The team decided to use a PCB board designed by Eric Colter. The PCB utilized many sockets to allow for reprograming of the GALs and the Microcontroller. It was also decided to use a black box to store the PCB board. This black box has holes drilled in for the extension cord wires to protrude. It’s shape and size is perfect for portability with any gamecube controller and allows easy-usage by anyone. All you have to do is put in your controller into the box.

For the packaging design of Frame Perfect we chose to use a PCB board and fit it inside of a 4.3”x3.2”x1.6” black container. After the container was obtained it was decided to drill two holes for each end of the gamecube extension cord to protrude. Once these were drilled it was decided to expand the forward facing hole to allow the gamecube controller to be plugged directly into the box. It was found that this also serves the dual function of securing the PCB board so that it will not get damaged.

The PCB was designed by Eric Colter for usability. To do this the PCB was designed so that there are sockets which will allow for the user to reprogram the GALs and the microcontroller. On the PCB, there was a power line that everything is powered by, but there are two possible sources to connect the power line to. One source is the 5V rumble motor power line, which is supplied by the gamecube. The other is the linear regulator fed by an external wall outlet. By choosing where to solder a wire, one of two separate power sources can be chosen. If the user desires rumble pack functionality, the PCB may be configured so power is sourced from an external wall outlet. If the user desires to not have an extra power cable to connect to the box, the PCB can be configured to be sourced from the 5V rumble motor supply line. The Controller may be disconnected from the rumble motor supply line then to prevent extra noise on the power line.

1. **Summary and Conclusions**

By completing this project many things were learned, almost too many to count. One such thing is how to interface a microcontroller in order to read and write in a different protocol. Another is that when designing a PCB board add extra via and pads. This is essential for quick debugging, and was actually utilized in our design to correct an incorrectly routed PCB trace.

Another learned skill was writing generic, modular software. This allowed for easy modification and a was essential to being able to debug in a timely manner. With this came how to interface high level software with lower level software. This was needed due to specific timings and allowed for a much more dynamic program. Early on we learned how to separate hardware and software between groups. This allowed for multiple team members to edit and work on related parts of code or hardware without interfering with each other’s work. There was also a more modular approach to the work. Dividing the problem into several small parts helped to make the project more digestible and easy to complete.

In the very beginning we set up an outline for the code. This outline saved a significant amount of time and gave a good indication of where we were as a team. Having us all be roommates allowed for quick and efficient communication. This meant that scheduled meetings were very efficiently planned. The last thing learned was that for the game “Super Smash Bros. Melee” there are two input frames for every game frame, which was something unexpected by a group of students very familiar to this game.

If there were more time we would interface a larger form of memory. At the moment 32 kilobytes of data can store approximately 20 seconds of macros. While this is more than enough for the current scope of the project we would prefer to have a giant repository of macros to complete compelling tasks within the game. If this were completed we would write several more intricate macros to allow for an impressive display of frame perfect maneuvers.

**7.0 References**

[1] Hitmen.c02.at, 'Yet Another Gamecube Documentation', 2015. [Online]. Available: http://hitmen.c02.at/files/yagcd/yagcd/chap9.html. [Accessed: 11- Dec- 2015].

[2] Int03.co.uk, 'Nintendo Gamecube Controller Pinout', 2002. [Online]. Available: http://www.int03.co.uk/crema/hardware/gamecube/gc-control.html. [Accessed: 11- Dec- 2015].

[3] D. Raphael, 'Hacking the GameCube controller on the Raspberry Pi - Dave's Blog', Davesblog.com, 2013. [Online]. Available: http://www.davesblog.com/blog/2013/12/27/hacking-the-gamecube-controller-on-the-raspberry-pi/. [Accessed: 11- Dec- 2015].

**Appendix A:**

**Individual Contributions**

**and**

**Activity Logs**

**Activity Log for:** Eric Colter **Role:** Interface Leader

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Activity*** | ***Date*** | ***Start Time*** | ***End Time*** | ***Time Spent*** |
| Meeting to discuss GameCube protocol | 11/20 | 6:00 pm | 8:00 pm | 2 |
| Protocol research and GAL development | 11/21 | 10:00 am | 6:30 pm | 8.5 |
| Eagle schematic design and selecting parts | 11/27 | 8:00 am | 11:00 am | 2 |
| Meeting to discuss functionality and next steps. | 11/27 | 2:00 pm | 4:00 pm | 2 |
| Completing GAL code, determining hardware characteristics of Gamecube | 11/28 | 10:00 am | 5:00 pm | 7 |
| Determining hardware characteristics of gamecube, building and debugging breadboard hardware prototype | 11/29 | 10:00 am | 5:00 pm | 7 |
| Work on PCB layout | 11/30 | 3:00pm | 5:00 pm | 2 |
| Complete PCB layout | 12/1 | 12:00 pm | 3:00 pm | 3 |
| Meeting to discuss software pipeline. | 12/3 | 6:00 pm | 9:00 pm | 3 |
| Development of software pipeline. | 12/5 | 9:00 am | 6:00 pm | 9 |
| Development of GameCube disconnection recovery software and development of simple macros. | 12/6 | 6:00 pm | 2:00 am | 8 |
| Reconfiguration of breadboard hardware,  soldering of first PCB board | 12/7 | 12:00 pm | 11:00 pm | 11 |
| Debugging of PCB board, working on poster | 12/8 | 6:00 pm | 3:00 am | 9 |
| Packaging of project and writing of report. | 12/9 | 6:00 pm | 10:00 pm | 4 |
| Soldering and packaging of second PCB board, creation of project video. | 12/10 | 6:00 pm | 12:00 pm | 6 |
| Spark Challenge attendance. | 12/11 | 1:00 pm | 5:00 pm | 4 |
|  |  |  |  |  |

**Written Summary of Technical Contributions: Eric Colter**

I “reverse engineered” the GameCube and controller by examining the characteristics of the data and power lines and creating Thevenin equivalent models. For example, the data line was found to have about a 700 ohm pull up resistor on both the controller and GameCube side. I also determined how the system reacted to various conditions, such as the controller not receiving the 5V rumble motor supply, the dataline being separated with a transmission gate, and having additional pull up resistors added to the data line.

I empirically determined the correct delays to add in software in order to properly read bits with the unusual protocol used by the controller and GameCube using a GPIO pin. I debugged the software responsible for reading this protocol to receive controller data.

I programmed and designed the two GAL and 64 bit shift register PLD block that output the correct waveform on the GameCube data line. This involved programming the ABEL source files for the GAL chips, determining how to interface the output from the GAL with the line using a 2N3704 NPN resistor, and breadboarding and testing the block. This also involved developing creative workarounds to problems such as having insufficient macrocells, the lack of an asynchronous present on the GAL chip, and all of the macrocells in a GAL having the asynchronous reset tied together.

I constructed and debugged the entire design on a breadboard. I created the schematic in EAGLE, had the design peer reviewed, and created revisions. I selected appropriate packages for the parts to be used on the PCB board. I designed the PCB board, had it peer reviewed, and created several revisions as well. I also researched different parts to buy and different PCB manufactures.

I soldered the components onto the first PCB board. I went through the process of debugging the PCB board and came up for solutions for the problems of too much pull up resistance and a misrouted trace.

**Activity Log for:** Diego De La Fuente **Role:** Peripheral Leader

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Activity*** | ***Date*** | ***Start Time*** | ***End Time*** | ***Time Spent*** |
| Meeting to discuss GameCube protocol. | 11/20 | 6:00 pm | 8:00 pm | 2 |
| Development of software to read input. | 11/21 | 10:00 am | 6:30 pm | 10 |
| Meeting to talk about how project will work and how we’ll continue | 11/27 | 2:00 pm | 4:00 pm | 2 |
| Meeting to discuss software pipeline. | 12/3 | 6:00 pm | 9:00 pm | 3 |
| Development of software pipeline. | 12/5 | 9:00 am | 6:00 pm | 9 |
| Development of GameCube disconnection recovery software and other peripheral issues | 12/6 | 6:00 pm | 2:00 am | 8 |
| Demonstration and development of macros, ordering of packaging. | 12/7 | 12:00 pm | 1:00 am | 13 |
| Creation of poster and development of macros. | 12/8 | 6:00 pm | 3:00 am | 10 |
| Packaging of project and writing of report. | 12/9 | 6:00 pm | 10:00 pm | 4 |
| Soldering and packaging PCB board, creation of project video. | 12/10 | 6:00 pm | 12:00 pm | 6 |
| Spark Challenge attendance. | 12/11 | 1:00 pm | 5:00 pm | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Written Summary of Technical Contributions:** Diego De La Fuente

Through conversations with friends and others, brought up idea to create a ‘multi-shine’ button that will include several schemes of moves to group during brainstorming sessions. Although not a technical contribution, was the basic start of our project

Being the peripheral leader of the group, my first job was to first implement the logic discussed by the group in order to make the IRQ be able to read the controller input from the the line. This was done by creating logic to set the IRQ in edge-falling mode and to sample the communication then every 1.25 us after each fallen edge(every new bit). When the value is sampled, then through bit manipulation of a 8-length string type, was able to store logic in a rather compressed format of one string.

The next portion of my job had to do with how to clock and push logic to the GAL and the shift registers. In collaboration with Eric Colter, who was the hardware leader, we determined to utilize the TIM and SPI in order to do those two respective actions. What was done was there were a few functions that were created that, when called up, would push the given Input type which held our code and would push it out via SPI in MOSI mode. The TIM module was set as well in order to pulse the GAL in order for its logic to work properly. We also established that we would pulse the GAL to perform logic and push its code inside the IRQ logic as well, so as to push the last input before the first one was read.

The final portion of my job in terms of Peripherals had to with determining using the COP watchdog timer in order to make our controller more usable and allowing us to escape the code if there were any errors reading the IRQ. The COP would be initialized in our initializations function right before our IRQ logic would begin. If the program would not be able to read all 64 bits of the controller input (software error, pulling controller out of gamecube, turning off gamecube, etc. etc.) then the COP was not pulsed by our program and the program would reset.

Later on, I contributed with the functionality of the program with writing the specific frames for the macros for Fox, which was the character we focused on with this iteration of the project. I utilized my own knowledge as well as frame data already known about the game to write several combos via the macros and debugged them in order to work.

**Activity Log for:** Kyle Diekhoff **Role:** TDP

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Activity*** | ***Date*** | ***Start Time*** | ***End Time*** | ***Time Spent*** |
| Meeting to discuss GameCube protocol. | 11/20 | 6:00 pm | 8:00 pm | 2 |
| Shift Register Construction/Gal Programming | 11/21 | 10:00 am | 6:00 pm | 8 |
| Team Meeting | 11/27 | 2:00 pm | 4:00 pm | 2 |
| Research frame data | 11/28 | 11:30 am | 3:30 pm | 4 |
| Wrote Macro.c and excel script for easy macro generation | 12/5 | 9:00 am | 6:00 pm | 9 |
| Programmed Macros/Debugging | 12/7 | 11:30 pm | 8:00 pm | 8.5 |
| Programmed Macros, researched feasibility of complex macro | 12/8 | 6:00 pm | 3:00 am | 9 |
| Writing report | 12/9 | 6:00 pm | 10:30 pm | 4.5 |
| Demo in HKN lounge | 12/10 | 12:00 pm | 1:30 pm | 1.5 |
| Spark Challenge attendance. | 12/11 | 1:00 pm | 5:00 pm | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Written Summary of Technical Contributions: Kyle Diekhoff**

I interfaced eight 8-bit shift registers together in order to form a 64-bit shift register. Then I debugged the shift registers and tied the asynchronous reset to high so that they would not have random behavior. After this I wrote ABEL code to interface the GAL as a down counter to the shift registers. This was when we realized that we would have to utilize two GALs because of the amount of pins we needed.

After this I then researched frame data to assess feasibility of complex macros. There were many that I looked into, in particular was allowing a character to travel under the entire length of the stage. It was determined that we would attempt this along with several other macros.

At this point I wrote the macro.c function. This function stores macros in flash memory. The function also allows for the loaded macros to be output to the gamecube. In this time I also wrote the macro.h function to allow the function to know where each macro starts in the array and be able to change the size of various macros. Once this was complete I wrote an excel file with a macro programmed in vba to allow for easy macro generation in which you can manipulate each bit in the macro and generate the syntax to input them into the file macro.c.

This session of lab was devoted to attempting advanced macros, in particular mario under battlefield. Hours of tweaking produced the desired result 3 times out of 20. After this it was discovered that the time for an input frame of the game is 2 gamecube pulls compared to the 3 gamecube pulls which we expected. With this discovery it was deemed with our current setup it was impossible to do the frame perfect macro. After this was fixed the macro was attempted again but the results were nearly random and it was decided to be dropped for more practical applications.

**Activity Log for:** Jordan Huffaker **Role:** Software Leader

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Activity*** | ***Date*** | ***Start Time*** | ***End Time*** | ***Time Spent*** |
| Meeting to discuss GameCube protocol. | 11/20 | 6:00 pm | 8:00 pm | 2 |
| Development of software to read input. | 11/21 | 10:00 am | 6:30 pm | 8.5 |
| Meeting to discuss functionality and next steps. | 11/27 | 2:00 pm | 4:00 pm | 2 |
| Design of software pipeline. | 11/29 | 1:00 pm | 3:00 pm | 2 |
| Meeting to discuss software pipeline. | 12/3 | 6:00 pm | 9:00 pm | 3 |
| Development of software pipeline. | 12/5 | 9:00 am | 6:00 pm | 9 |
| Development of GameCube disconnection recovery software and development of simple macros. | 12/6 | 6:00 pm | 2:00 am | 8 |
| Demonstration and development of macros, ordering of packaging. | 12/7 | 12:00 pm | 1:00 am | 13 |
| Creation of poster and development of macros. | 12/8 | 6:00 pm | 3:00 am | 9 |
| Packaging of project and writing of report. | 12/9 | 6:00 pm | 10:00 pm | 4 |
| Soldering and packaging PCB board, creation of project video. | 12/10 | 6:00 pm | 12:00 pm | 6 |
| Spark Challenge attendance. | 12/11 | 1:00 pm | 5:00 pm | 4 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Written Summary of Technical Contributions: Jordan Huffaker**

As the software lead of the team, my role was primarily to ensure that all software components work together and that they work with the hardware. While the other members of my team focused on specific modules of the project, my role was a more high level approach. Due to the contributions of my position, other members of the team were able to connect their code and hardware together into a seamless unit with little difficulty.

One of the core contributions that I made was to design the software pipeline that outlines the different software components necessary and how they work together. The software pipeline is a visual diagram that excludes the fine details of the code and can be understood from a team perspective. It was through the creation of this diagram that I was able to assign Kyle the role of developing the macro functionality (outlined in the macro.c file) and Diego the role of developing the GAL-microcontroller, microcontroller-transmission gate, and microcontroller-GameCube driver software. I took on the role of developing the core of the software functionality, the ability to start macros, switch macro buttons, and switch between macro schemes.

With the distribution of work done and having completed my role of development, my second role of development took effect. My second role of development was to test the software functionality to ensure that the inputs from the GameCube controller actually produce the expected outputs. This includes more than just testing the individual piece of code that I wrote, but rather, testing the entire software as a unit. It was through this contribution that we were assured that the software produces the expected output.

The final contribution that I made was to assist in the development of macros that are used in the final project. Some macros, such as the fox waveshine, were particularly difficult to create. These macros took rigorous debugging that took multiple people to fully check and understand. This contribution was one reason that we were able to have a complete macro scheme available to fox players.

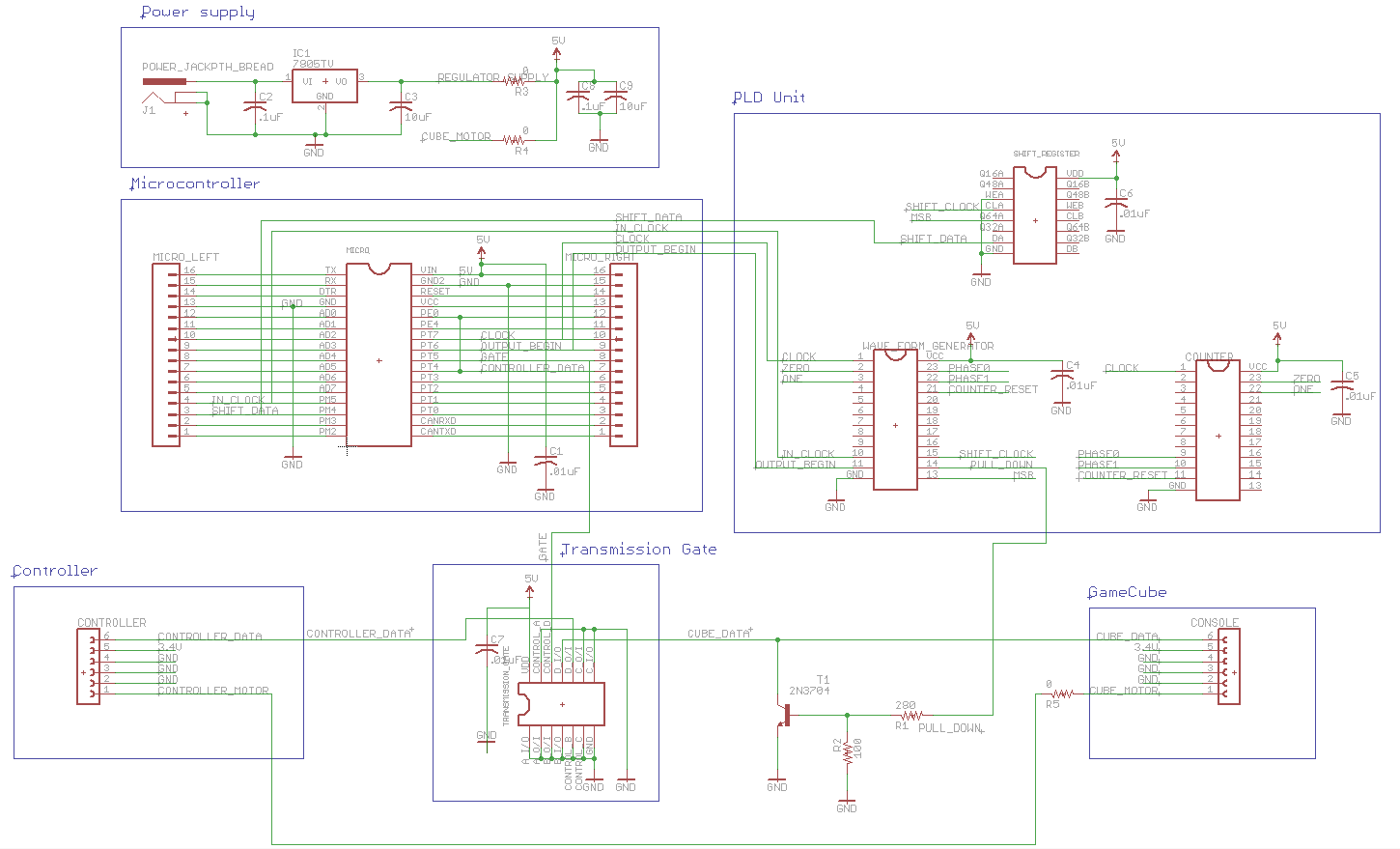
**Appendix B:**

**Interface Schematic**

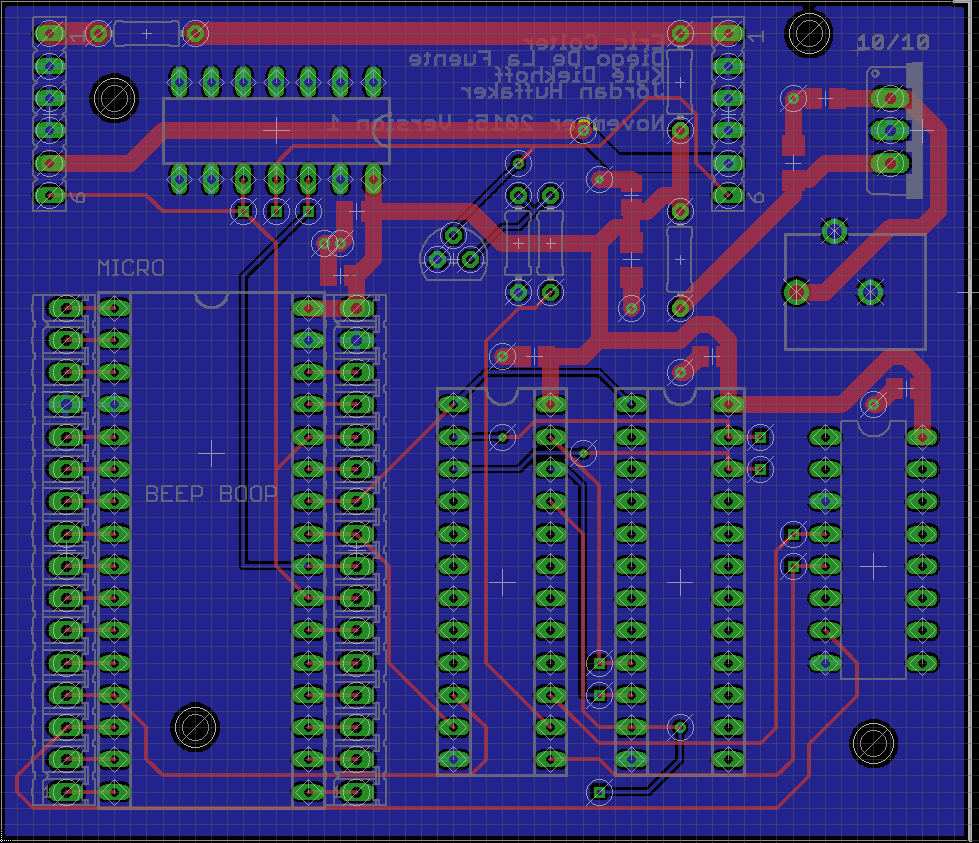
**and**

**PCB Layout Design**

**Complete Eagle Schematic**



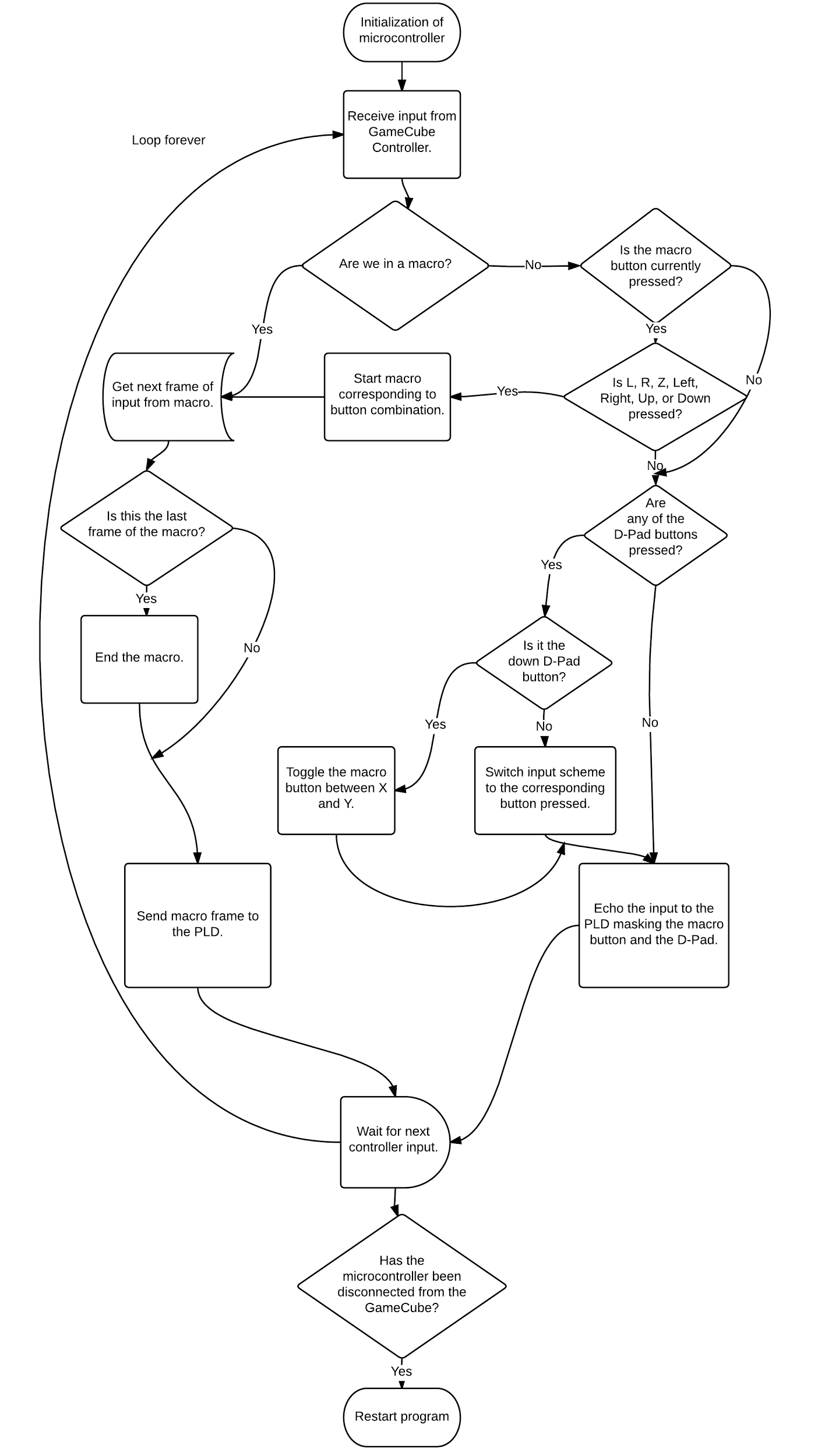
**PCB Board layout**



Schematic and PCB layout created by Eric Colter

**Appendix C:**

**Software Flowcharts**



*Software Design of Microcontroller - Documentation by Jordan Huffaker*

**Appendix D:**

**Packaging Design**



Packaging Design Front Packaging Design Top



Packaging Design Inside Packaging Design Inside

(PCB) (Wire)

*Photos Taken by Kyle Diekhoff and Jordan Huffaker*