Handheld Emulation Station

Final Report

Group: sdmay19-25

Client/Faculty Advisor: Dr. Julie Rursch

Team Members:

Nick Lang

Sean Hinchee

Matthew Kirpes

Nic Losby

Jacob Nachman

Team Email: sdmay19-25@iastate.edu

Team Website: https://sdmay19-25.sd.ece.iastate.edu

1. Executive Summary

The project we chose to do was make a handheld emulation station in order to play old school retro games. We felt that this would be a good project that, not only encapsulated all of the skills we have learned throughout our college career, but also was a project we would enjoy to work on throughout the entirety of our senior year.

The project consists of three main components that we have been working on, a Printed-Circuit Board (PCB) which holds the raspberry pi that runs the system, a kernel module for the system with General Purpose Input and Output pins (GPIO) for button input, and a software emulator to run games for the original GameBoy. We felt that these three components were necessary to, not only make the system we have imagined for ourselves, but applied a wide range of skills we have learned throughout our various classes here at Iowa State including Electrical Engineering (EE) classes, Operating Systems, as well as the various software classes we have been in throughout our college career.

The EE component of the project (the PCB) is where all of our circuitry lies. We have designed it to charge the system as well as be able to connect the battery and draw from it to power the raspberry pi. It connects the screen, battery, pi, and buttons all together to make a functioning system.

The kernel module and GPIO part of our system is where the operating systems knowledge comes into play. We have developed a fully functioning kernel module with our systems specific needs in mind. This will allow our system to run on low latency as we do not need anything fancy or complex in order to run our system. It contains a semi-functioning GPIO library we have developed ourselves which allows for low latency button input.

The software side of the project, the GameBoy emulator is the primary software component of our project. It is written in the Go programming language and has the core components of an emulator. We emulate the Gameboy version of the Z80 processor, which is a 8 and 16 bit processor. We emulate reading and writing to memory, render images to the screen, and have a partially functioning GPU. Overall, this component of the project was extremely difficult, and took up a lot of our time but was a very good learning experience from a software standpoint.

All of the above is packed nicely into a case that we have made using TinkerCAD software and 3D printed ourselves. This allows us to make the case at a cost efficient price, as well as to change and customize it to our exact specifications, as well as makes it easy to change whenever we make a hardware or dimensional change.

2. Requirements and Specifications

2.1. Functional Requirements

We only have one target user group for our Handheld Emulation Station, which is gamers who want a retro experience in there pockets. Since we only have one user group, our functional requirements and uses cases are targeted towards their experience of the product. Our functional requirements for the user are as follows:

- Long lasting battery life
- Universal save files
- Various games to play
- 3rd party software support
- Load and save games

We want our users to get the most out of their emulation station and we feel that these requirements are the core of what our users should want and the core of what they should be able to do with their system.

2.2. Use Cases

The main use case for our product is to allow a user to have a powerful as well as versatile emulation platform in their pocket to play the retro games that they know and love.

Our platform also offers a way to easily backup saves files to the cloud to maintain data integrity. This feature would also allow users to sync their save states between multiple devices.

2.3. Non-Functional Requirements (tied to clients and/or target users)

The target user group for our project in conjunction with the competing, existent, solutions leaves our project in a position where we must prioritize compactness, ergonomics, and versatility. Our product must be able to fit in a user's pocket, or, at minimum, be small enough to make transport a trivial concern. As such, the user experience is oriented around making a product which is able to perform a variety of tasks for an extended period of time across many locations.

3. System Design & Development

3.1. Design Plan

Our design plan for the emulation station consists making the PCB, Emulator, Kernel Mode and case throughout the course of our second semester here at Iowa State. We figured that working on all 3 simultaneous was the best and most efficient way to divide our time, as well as give us the ability to make changes depending on the changing needs of another component on the system.

The PCB was designed with integration and efficiency in mind. We wanted to get the most power out of our battery in order to be able to power our 3.5 inch screen, Raspberry Pi computer, all of the buttons, and accessories our design has. We also needed to be able to charge and power the Raspberry Pi at 5V whereas the battery only supplied 3.7V. We went down the path of using example circuits in the datasheets from the manufacturer. This did not pan out the way it was expected. After researching why the power circuit was not functioning properly, it was discovered the manufacturer or the parts supplier supplied the incorrect datasheet and the correct datasheet was nowhere to be found. After dealing with this setback numerous times we purchased a reliable alternative in order to meet our project needs.

The design of the kernel module was focused on keeping latency low for the inputs of our system. It is very important to have low latency when playing games due to the fast reaction times needed for certain mechanics in various games. The kernel module is implemented to target the /dev, or, udev, interface in Linux. That is, the physical input buttons are transported over GPIO to allow for low-latency transport of user inputs. A filesystem is implemented using the linux/fs.h and linux/device.h libraries and GPIO inputs are implemented using the linux/gpio.h library. A user should be able to perform a 10-byte read from a given /dev/rtXX file to see the button state for the associated GPIO pin where XX is a 0-padded number in the range of [0,13]. GPIO pins are read at the moment of a /dev file read event, so an opened file descriptor can be held and repeatedly read for GPIO pin state at the given moment. The table defining the relationship between pin numbers and /dev file name are shown in Figure 3. In the future, the table of relationships could be expanded to accommodate human-readable file names, e.g. /dev/rtjoystick, /dev/rtstart, etc.

The Gameboy emulator portion of our system is our attempt to design something new and simple that uses the hardware of our design, as well as to attempt to implement a challenging part of the project. We started off by designing a fully function Gameboy Z80 hybrid CPU and added support for all 510 opcodes that the CPU needed to function through the use of various Gameboy hardware documentation. We simultaneously emulated the memory system by following the memory map provided in the documentation, which enabled us to read and write memory from a game cartridge to the system. From there we went on to implement a screen renderer in order to draw images to the screen, and finally, finish it up by connecting the system to the renderer by emulating the system GPU.

We finalized all of this by making a custom case tuned to our system specifications using CAD software and printing it, allowing us to put all of our electronics into a nice compact container that will keep the system internals safe, as well as make it look sleek and user friendly.

3.2. Design Objectives

Our goal with these design choices was to make a fully integrated system that users could hold in their hand and play the games that they want to. We focused on compact, custom software, good battery life, and low latency in order to fulfil the needs of our target audience. We want them to be able to feel like they are using a system we enjoy

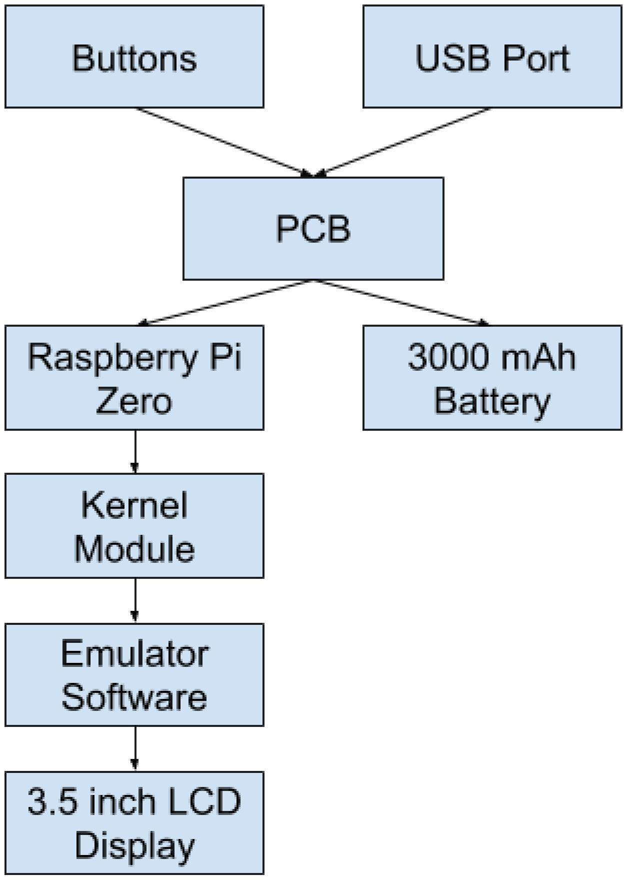
3.3. System Constraints

Our current biggest system constraint is that the emulator does not render to the screen properly. This is because our GPU is not fully complete and cannot properly render the screen from video memory. On top of this, our current prototype is not very ergonomical with its boxy build and the screen runs at roughly 30 fps.

3.4. System Architecture Diagram

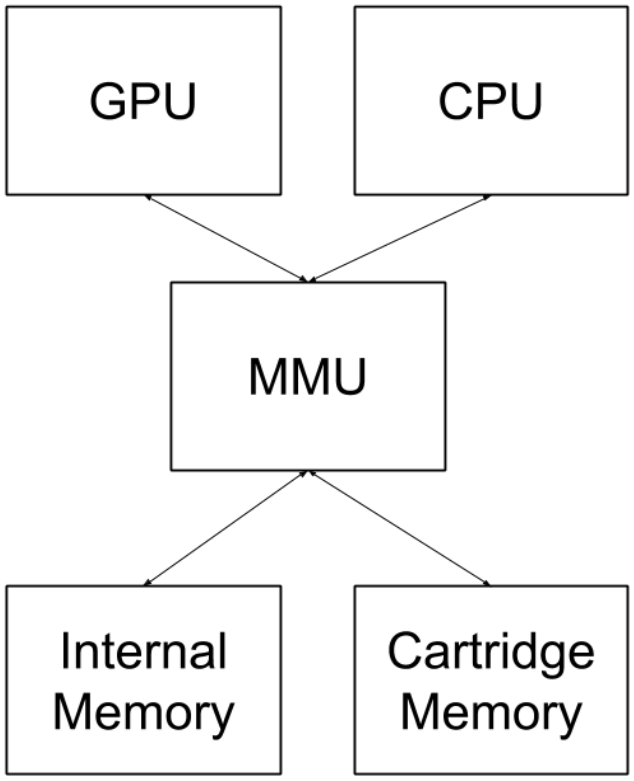
Our current system block diagram is as follows:

Figure 1



Our system contains button and usb inputs which get sent through the PCB to the raspberry pi zero that runs our software. The raspberry pi has a kernel module on it with a custom GPIO library that reads the user input from buttons or the data input from USB. From there the inputs are read by the emulator and, as the CPU cycles through, it draws images to the screen.

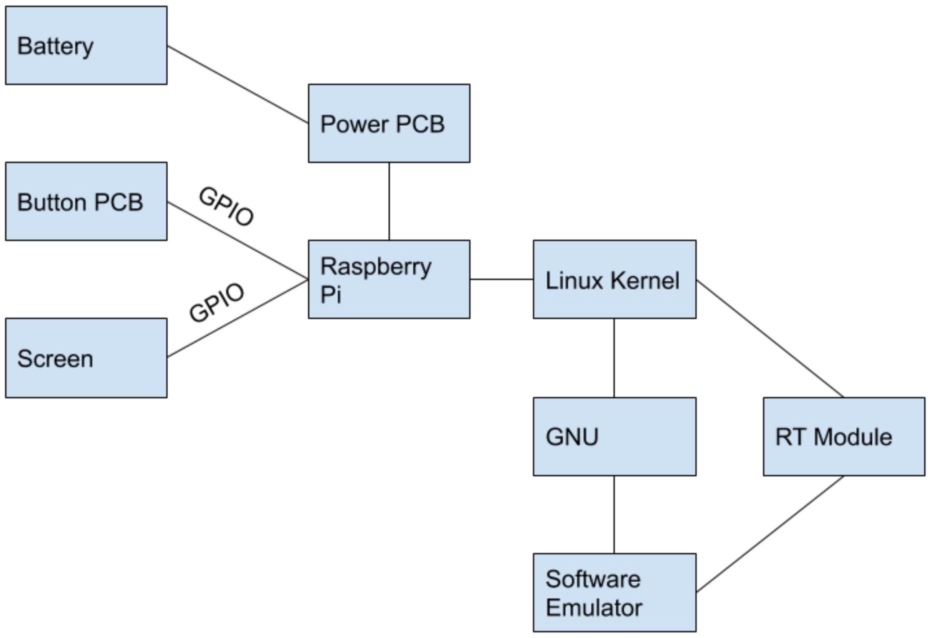
Figure 2



The diagram above is the system diagram for the emulator itself. The memory management unit (MMU) is the core component of how the emulator functions. When a cartridge (ROM) is loaded into the emulator, hex values are read from the cartridge and the CPU determines what to do based on the cartridge values. Depending on the command, the MMU will write new values to the registers, send values to the CPU, send values to the GPU to render the screen, and stores and loads the needed values into internal memory.

3.5. Design Block Diagram

Figure 3



3.6. Modules/Interfaces

The kernel module was designed to provide the following interface scheme:

Figure 4

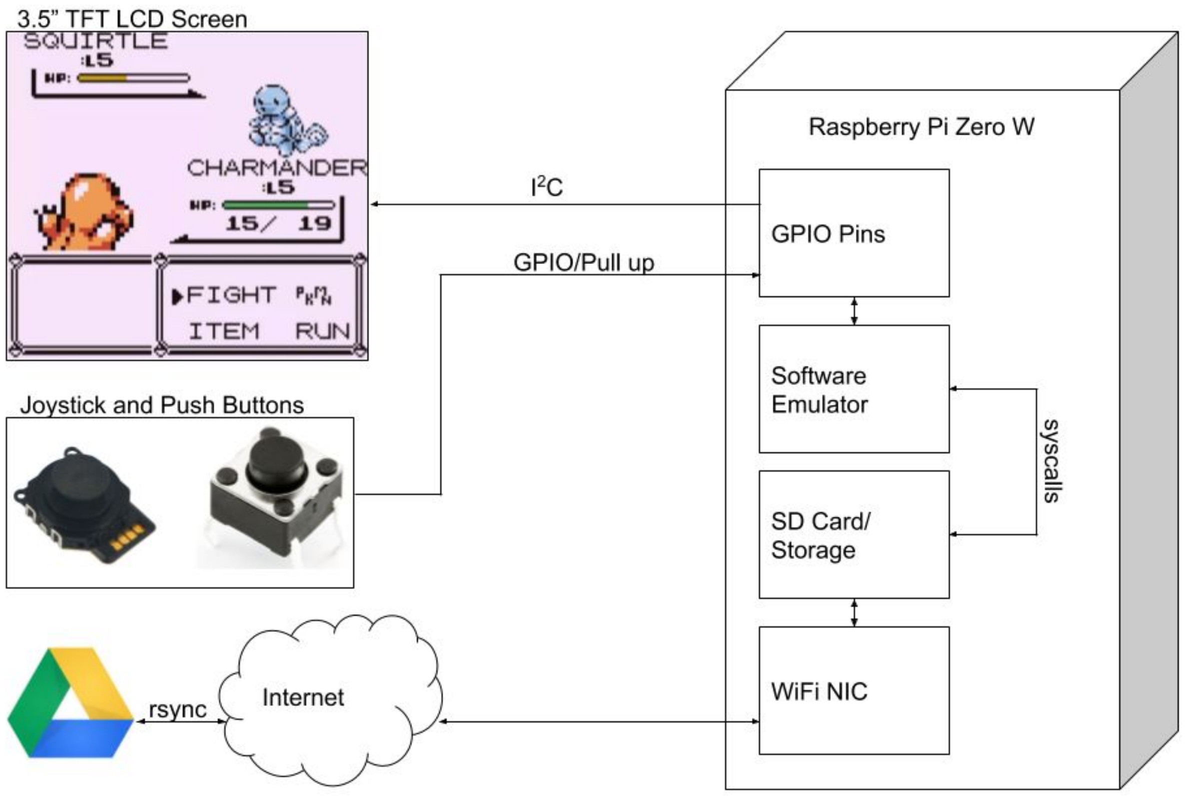
GPIO Pin Number	Dev File	Peripheral
03	/dev/rt0	Y-axis Joystick
05	/dev/rt4	D on D-Pad
07	Reserved	Speaker
12	/dev/rt1	X-axis Joystick
13	/dev/rt2	U on D-Pad
15	/dev/rt3	R on D-Pad
16	/dev/rt5	L on D-Pad
29	/dev/rt12	L trigger

31	/dev/rt13	R trigger
33	/dev/rt6	X
34	/dev/rt7	A
35	/dev/rt9	Y
36	/dev/rt8	B
37	/dev/rt11	Start
40	/dev/rt10	Select

4. Implementation

4.1. Implementation Diagram

Figure 5



4.2. Technologies/Software Used

There were various technologies and software used in the making of our emulation station. Listed below are the technologies we used with an explanation of what they were used for:

- **TinkerCAD:** TinkerCAD is an online, web-based CAD software which we used to make models for our case. It has a low learning curve, simple to use, and is very dynamic with file types which were all perfect features for our needs since we do not use CAD software very often.
- **Golang:** Golang is the programming language we chose to write our emulator in. Features include a very nice testing library, good code readability, simple error handling and the added benefit of learning a new programming language that could be used as an alternative to C in future projects.
- **EasyEDA:** EasyEDA is an online, web-based Electronic Design Automation (EDA) tool that can be used to design, implement, and test circuits. We chose this software for similar reasons to tinkerCAD in the form of simplicity and ease of access. It also has built-in features for converting system schematics to PCB designs that we could send to be printed. In addition, this software ties into the parts library of LCSC a large parts supplier. At a click of a button, we were able to order both the PCB and the parts to be soldered and have them shipped together.
- **SDL2:** SDL stands for Simple DirectMedia Layer which is the screen rendering library we implemented for our emulator. It provides a simple graphics API that we used to draw images to the screen. It was originally a C library that we found a Golang wrapper for and were able to implement

4.3. Software/Design Choices

4.3.1. Software Design

The reason we chose to write the program in Golang is because we felt it was, not only a better alternative to C, but that we would use this as an opportunity to learn a new programming language and give us the ability progress our skills. The programming language also provides great infrastructure for things like setting up testing, error handling, as well as garbage collection so we do not have to worry about memory mismanagement. The language is also very readable which was helpful when code was getting peer reviewed.

When designing the emulator, we chose to follow the Gameboy CPU manual as well as the GBdev wiki for the implementation of the Gameboy z80. This allowed us to know all of the opcodes that this processor needed, as well as provided us with the memory map for all of the system components. We chose the SDL library as there was a lot of documentation on it, it was very easy to use and set up on linux distros (i.e. the Raspberry Pi Zero), and allowed us to render things to the screen rather easily. When integrating the system, we had it set up so that each component of the CPU was it's own piece of software and would send the output from one piece to the next, as well as have a native API that one component could use to connect to another component. This made the integration of the system very.

4.3.2. Hardware Design

We chose to design our PCB with simplicity in mind. We wanted to assure that we would be able to power our entire system with all of its components, with as little power loss as possible, as well as making sure that the PCB was small so that we could fit it into our desired size, as being a compact solution is one of the things we were aiming to do. We attempted to design our own charging unit ourselves so that the device would not overcharge or short circuit, but that proved to be a very difficult task, and we ultimately had to swap it out with a purchased unit for our system. Another interesting design choice we had was actually not one but two PCB boards. This is because, for the sake of being a compact solution, it was actually better for us to design one dedicated to peripherals, and another for the main unit and the placement of the Raspberry Pi Zero W. This allowed us to make the second board smaller and place it on the side of the main board, allowing us to make it even more compact. This also allowed us to keep the costs down as printing 10 boards under 100mm^2 was \$2.00.

All of our component choices (buttons, USB, charging circuit, etc) were made based on the best way to keep our prototype small. EasyEDA, the online PCB designer we utilized, integrated well with the part selection the LCSC maintains. We had the ability to import any part they had along with the part footprint which made PCB design much faster. When picking between Surface Mount (SMD) and Through-Hole (THT) the price differences between various options were negligible, so we went with the most compact solution for our peripherals which was SMD. We chose 0603 as the size due to the minimal footprint but also being large enough to still solder by hand. We chose our screen based of the open source device blob for the screen, kernel support for the screen, as well as the end thickness of the screen. Screen technology should have been a factor in the initial process.

4.4. Standards and Best Practices

5. Testing, Validation, and Evaluation

5.1. Test Plan

5.1.1. Emulator

In developing the software emulator for the gameboy our testing approach was to create tests for each individual component in a vacuum. We also develop tests that required multiple components to test their integration and the system as a whole. These test were used to continually test our emulator as it was being developed to ensure that any changes did not break previously working features.

5.1.2. PCB

The plan for testing the power circuit went as follows. We would test the PCB for continuity before soldering anything. Then after this test was passed, we would visually inspect the PCB for any potential defects. After this, we would solder the board fully. We would test the continuity of the part to the solder joint and/or pad on the PCB if any pad was left exposed. This ensured we had a solid connection between the part and the pad. Following this, we would solder a battery’s leads to the battery pads on the PCB and test the voltage across the battery. Then we would plug in the micro USB cable into the microUSB port and test VCC across the micro USB connector ground on the PCB. If at least 5V was measured, we would flip the power switch or bridge the two pads on the PCVB connecting VCC to the ICs. This would power the circuit overall. We would then test the voltages going from VCC all the way to the 5V output or at least what should have been the 5V output. Usually the power circuit wass outputting wellover 20V. We tried taking an off the shelf step down converter to step the 25V back down to 5V. This was successful and we were able to power a LED however we were unable to sustain any amount of real power draw. We then removed the step down converter. For completeness, we shorted the output terminal and the circuit shut off as expected only outputting voltage when a charging pulse was detected. We also simulated a dying battery by using a lab bench power supply and slowly dropping the voltage down to 3.4V. This was how we discovered the feedback pin on the boost converter IC was actually the VCC pin as the output did not stop. We would have over discharged any battery we used and the testing battery was desoldered from the PCB.

Testing for battery life is as follows. First we charge up the batter to its full capacity. Then we measure the voltage output form the battery at full capacity, around 4.3V. We then stress test the system by playing a demanding game for 10 minutes. Afterwards we check the battery voltage yet again. We keep track every 10 minutes for an hour and after an average drop of 0.02V per 10 minute time period we fully discharge the battery. This battery gets shut off around 3.4V. This gives us around 7.5 hours of estimated battery life.

5.1.3. Kernel Module

Kernel module testing and validation was performed via a series of assert(3) calls to validate software operation being successful, verbose logging, and comparing the inputs and outputs side by side to a Python program using a common and well-documented GPIO management library for the Raspberry Pi.

5.2. Unit Testing

The software emulator utilizes the go test tool unit testing environment. Each opcode in the CPU has an individual test and each opcode is tested for state contamination as well as bad values. Each test builds a CPU, indexes affected registers, performs the opcode, and then tests for state contamination as well as bad values. Similarly the memory unit has test cases for reading and writing to each of the memory ranges to ensure that reading and writing is performed properly.

5.3. Interface Testing

The real-time kernel module interfaces - exposed as /dev/rtXX files - are tested via the opening of each in turn and splitting the output of a given read at a point in time between a shell-level monitor and a monitor within the software emulator and comparing the results to each other for consistency. That is, the /dev/rtXX filesystem should agree with, programatically, the Python monitor script and the agreed value should be reflected within the software emulator.

5.4. System Integration Testing

As per the interface testing section, 5.3, the primary method of establishing integration between the system as a whole is by validating the chain of inputs from the press of a button to the reception of a given input within the software emulator. The complete chain is:

1. Physical button is pressed
2. Button press is passed over GPIO to the Raspberry Pi
3. Kernel driver registers the button press during file read event
4. File read event passes value to desired programmatic state in the emulator

Testing of the integration sequence can occur between each step via utilizing well documented and known-to-work tools which indicate that the system is functioning as intended such as multimeter testing, Python script validation, kernel log validation ,etc. At each point within the testing sequence, the correct state should be reflected in real-time.

5.5. Validation and Verification

Validation occurs at multiple stages within a given testing sequence or operation for both the software emulator and the kernel module.

The kernel module rigorously tests the state of operation and does not take chances with bad state. That is, operation will cease when state is known to be in an inoperable or unsafe combination. Additionally, said state and the circumstances surrounding the invalid state -- via variable state printing or otherwise -- will be logged to the kernel debug log available via dmesg(1). These log messages are logged using printk(9) in combination with the macros KERN_WARNING and KERN_INFO to further enumerate the intent and severity of the state violation.

The software emulator sports a robust test suite as well as decisive and involved error handling at run time. Chatty logging is an option available for run-time and errors are handled through a suite of function interfaces which allow for accurate and expressive message handling. Explicitly debug-oriented calls can be placed in source at any location via the debug() function, printed only when chatty mode. Errors can be handled dynamically or be passed into the function efatal() which accepts a formatted message and an error, creating a line-unique error message coupled with a fatal ending of the run-time operation. Furthermore, the efatal() function calls the sysfatal() function which ends the run-time operation in a fatal manner, providing a line-unique message in the error/debug log. The cohesive concatenation of a non-nil error being passed to efatal() will output something similar to: [timestamp] Fatal: [line-specific message] -- [error text].

6. Project and Risk Management

6.1. Task Decomposition/Roles and Responsibilities

The roles and responsibilities for the project are as follows:

- **Jacob Nachman:** Development time mostly spent on the emulator. Worked on the opcodes, CPU implementation, GPU implementation, and integration. Assisted in the original version of the PCB design and the screen renderer, as well as took charge of most of the documentation and paperwork for the semester.
- **Nick Lang:** Development time spent on the emulator. Worked on the various types of memory that the system needed to emulate, implemented save states for the system, and made the case design for the final prototype. Also assisted with integration of the system and designing testing for the emulator.
- **Matthew Kirpes:** Development time spent on the emulator. Focused on assisting with the memory implementation as well as the integration of the system.

- **Nic Losby:** Primary developer of the PCB. Focused time on iterative design making sure that the board had the ability to power all of our components as well as worked on system optimization. Primary handler of ordering system components and case printing.
- **Sean Hinchee:** Lead designer for the kernel module and GPIO library, as well as the primary developer of the software testing infrastructure and screen rendering library. Assisted with implementation of CPU as well as debugging and testing the system as a whole.

6.2. Project Schedule

6.2.1. Gantt Chart (Proposed vs Actual)

Proposed															
Tasks	**Semester 2 (week)**														
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Develop Emulator	■	■	■	■	■	■	■	■							
CPU	■	■	■	■	■										
GPU					■	■	■	■							
MMU	■	■	■	■											
Kernel Module			■	■	■	■	■	■	■						
PCB	■	■	■	■											
Assemble 1st Prototype						■	■								
Initial Testing					■	■	■	■	■						
Final Product Completed										■	■				
Final Testing												■	■	■	■
Final Presentation															■

Actual															
Tasks	**Semester 2 (week)**														
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**

Develop Emulator	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
CPU	■	■	■	■	■	■	■	■	■	■	■	■			
GPU												■	■	■	■
MMU	■	■	■	■	■	■	■	■	■	■					
Kernel Module			■	■	■	■	■	■	■	■	■	■	■		
PCB	■	■	■	■	■	■	■	■	■	■					
Assemble 1st Prototype									■	■					
Initial Testing									■	■	■	■			
Final Product Completed													■	■	
Final Testing															■
Final Presentation															■

6.3. Risks and Mitigation

6.3.1. Assumed Risks (before we started)

6.3.1.1. Kernel Module

Documentation was known to be sparse and potentially inconsistent for kernel-layer interfaces and libraries. As such, it was an expected potential risk that development of the module may be slow or significantly delayed due to lack of or incorrect documentation. Additionally, despite potential first impressions, the task to be performed by the kernel module is surprisingly niche, where many related solutions perform either implement significantly more or significantly fewer features than the needs of our group's kernel module. As such, utilizing other sources for reference is not necessarily a viable option as many kernel interfaces are designed for specific solution task scales, of which many are many greater in scope than our solution.

6.3.1.2. Software Emulator

The assumed risks with the emulator consisted a lot on the documentation front. We were not sure how reliable the documentation we were looking at was. For all we know, it could have been completely incorrect and we were following in blindly. After some cross referencing we decided to start with the Gameboy CPU manual and go from there. Other than that, we did not assume that making an emulator would be terribly difficult.

6.3.1.3. PCB

Since this was the first time anyone in our group even thought about designing and making a Printed Circuit Board we knew this was going to be a challenge. We also knew that Lithium Polymer batteries can be temperamental and can catch on fire if mishandled thus fire was an accepted risk. We accepted this risk and began to read up on how to make a PCB and watched numerous YouTube videos on the subject of making a schematic to getting a PCB made and shipped to us.

6.3.1.4. Case

We assumed there would only be the challenge of brushing up on prior CAD skills in order to design a case. We were unaware how clunky a rectangular case could feel in the beginning.

6.3.2. Actual Risks (challenges)

6.3.2.1. Kernel Module

Development of the Linux kernel module ran into a number of issues during its composition and refinement. Near the tail end of the project, the most significant issues in obtaining a consistent read from a GPIO pin were encountered. More explicitly, continuous reads of 0 would occur and be in opposition with validation from a Python GPIO script testing the same inputs. Upon further investigation at this stage in the kernel module development, it was implied by a sample of sources which indicated that the issue may stem from a lack of proper handling with regards to the Broadcom chip on the Raspberry Pi. That is, specific interrupt handling may need to be implemented within the kernel module to allow for consistent management of the GPIO pins. Research yielded few useful results in this area and lead to the collective decision to abandon development of the kernel module to the end of furthering work on the software emulator.

The emulator also suffered from changes currently occurring within the common interfaces for Linux modules and programs to express arbitrary resources as file systems due to the ecological shift to systemd resources and interfaces. Conflicting information was repeatedly found for how precisely to express resources as file systems in a manner which is not oriented for large-scale software projects outside of scope from our project design. Specifically, there are divergences in terms of how projects are intended to express themselves via the /sys and /dev device trees.

Additionally, the kernel module had to be re-licensed to allow the module to link against specific GNU libraries. This is evidenced by the call in the module source to the macro MODULE_LICENSE.

6.3.2.2. Software Emulator

The actual risks for the emulator that we encountered proved to be a lot more than we had anticipated from our research. All of the posts we were reading for research were actually from people who have written emulators previously, which caused us to vastly underestimate the amount of time it took to make one from scratch. Our documentation still proved to be problematic at times so we had to cross reference various sources. We also had to implement 510 opcodes instead of 255 due to a misunderstanding in how the opcode infrastructure was designed. On top of that, the GPU proved to be difficult to because the Gameboy GPU actually draws three screens instead of just one.

6.3.2.3. PCB

One unexpected risk was part shipping times and delays. Parts and the PCB had to come from China for the first part of the semester as it was very cheap and in order to iterate efficiently we utilized a Chinese based company for PCB manufacturing. Due to the Chinese New Year our PCB took twice as long as it usually did during the power ciruti debugging phase which pushed us back further than we had hoped to be with the PCB. Another challenge faced was the part supplier and/or manufacturer of one of the cheaper parts we were using, uploaded the incorrect datasheet for the part and thus the pinout was incorrect.

6.3.2.4. Case

The actual risks for the case was a broken 3D printer which took many hours to get back to a working condition. As well as how to commision use of someone else's printer while the one we have is nonfunctioning.

6.3.3. Mitigation of Risk

6.3.3.1. Kernel Module

Development of the kernel module was frozen in the wake of the end-of-semester deadlines coming up for the sake of advancing deliverables within the software emulator. This decision was finalized via collective vote on the team Slack server which was unanimous. As such, a future solution would be desired and a number of potential alternatives have been identified, including the potential to leverage an existing /sys device tree found within, specifically, the Raspbian kernel build of the Linux kernel. It is possible that, in the future, it would be possible to complete the kernel module, but a great deal of additional research and extensive testing would be required, all of which are time consuming in the face of deadlines and alternative deliverables.

6.3.3.2. Software Emulator

Our plan of mitigation for the risks of software emulation was pretty straightforward. We were able to find various documentation sources, as well as use other open source emulators as references when the documentation did not make a lot of sense. This helped us improve our skills in figuring out how to pick and choose reliable and necessary information to achieve a certain goal. As for the opcodes, we were able to use the cross referenced documentation to write a python script to write the opcodes as needed (they were various byte operations so rather easy to automate). As for the risk mitigation of the GPU issues, we attempted to draw larger than the screen would allow us, but ran into issues trying to render that properly or get it to display at all.

6.3.3.3. PCB

As mentioned previously, the power circuit had many troubles. For the boost converter IC, the VCC pin was switched with the feedback pin on the IC and thus our output voltage was 25V instead of 5V and could not sustain any substantial power draw. This is what caused us to purchase a power circuit instead. After using an off the shelf solution our debugging became non-existent and introduced a new feature of powering the system off the charger while still charging the battery.

6.3.3.4. Case

In order to mitigate the risk of the broken 3D printer, the entire hot end had to be disassembled, baked and soaked in acetone in order to remove the clogged materials. While doing this we were able to hire someone else to print parts for us as a backup plan. After repairing the printer and updating the firmware, we were able to achieve a very nice print overall.

6.4. Lessons Learned

6.4.1. Kernel Module

The kernel module was a fantastic source of learning material with regards to Linux module development, Linux device tree development, system integration, and API documentation.

Linux kernel module development is a realm of skill utility which is not extensively found within Iowa State University course syllabi and as such possesses a fairly noticeable learning curve compared to some other potential fields. Although development in C is not necessarily an issue, learning to effectively utilize open-source documentation and manually investigate points where documentation is incorrect is missing is an exceptionally valuable skill. The vast majority of time spent on this section of the project was spent reading - other - source code and documentation and attempting to apply the knowledge existent to implement or refine - via validation or otherwise - portions of the real-time input kernel module.

Due to the nature of the real-time input system, the specification for inputs had to be rigorously documented and consistently implemented. That is, all other integrations along the pipeline of the system integration should be able to -- with the relevant API documentation -- blindly document and implement functionality leveraging the real-time interfaces.

6.4.2. Software Emulator

We learned so much implementing our own gameboy emulator. One of the biggest things we learned is that almost all emulators are designed by people who have made one before. The learning curve was very high, the documentation was rocky, but it was a fantastic learning experience. Emulating a processor, not only helps us to improve our programming skills, but it also gives us a good experience at learning how to interpret documentation, make our own design choices, and helped us to improve on the computer engineering and processor design skills we have learned throughout college. I think that this part of the project was a very useful experience and one that will be remembered by us all going forward in our careers. Working our way through faulty documentation and being able to deduce outcomes using our intuition and a mix of sources is one skill that will be valuable throughout the entirety of our working careers.

6.4.3. PCB

We learned so much during the entirety of this project. From the perspective of the hardware we were able to learn about electronics schematics design, datasheet reading, schematics to PCB, PCB layout, PCB ordering, PCB testing, microsoldering, testing for power efficiency, deciding which parts should be used over other parts, system integration, and finally but not least, battery safety handling procedures. There were no fires this time when handling batteries.

While certain aspects of this list are within the scope of classes here at Iowa State University such as PCB design, the course is an elective and did not fit in to any of our schedules or priorities prior to Senior Design. If we had taken this course, we are sure it would have helped immensely throughout this project.

We also learned even though manufacturers or part suppliers provide documentation, there is no guarantee the documentation is correct. While looking for parts online too, it is very easy for not legit suppliers to provide false information about battery capacity as it is not trivial to thoroughly test the claims.

6.4.4. Case

We were able to revisit prior skills with CAD software and design a case from scratch. We learned how to pay others for use of their 3D printer as well as how to fix a team member's printer when the extruder became very clogged. We also learned designing an ergonomic case is no easy task as hands are very finicky and modeling their shape in CAD is an exuberant amount of time and effort.

7. Conclusions

7.1. Closing Remarks for the project

Overall, the project was a success. Although a great deal of roadblocks were encountered and some goals or portions of the project fell through, at the end of the day all goals were set by us, the students, and crafted out of a passion for the product and powered by self-motivation.

7.2. Future Work (potential directions for the Project

In the future, marketing this product would be very easy. There is a large market for other products like this one. Many online personalities have been reviewing a product very similar, even based off of the Raspberry Pi Zero W. The way we can differentiate ourselves from the competition is to switch over to the Raspberry Pi Compute Module 3+. We would achieve an effective 3x of our compute power and thus be able to emulate more systems out there including the N64 which is not currently possible on both our product and all other existing handheld products. By adding and additional thickness of 5mm we would be able to effectively double the battery life and keep our long hours of play time even with the compute power increase.

Since thinness is the name of the game when it comes to marketing a product, shaving even 1mm is preferred. We are currently using a clunky TFT screen with poor viewing angles. A change over to an IPS LCD would not only increase the brightness which would allow for better playing conditions in direct sunlight but also offer better screen refresh rates. In taking the proposed route of integrating a different IPS based LCD for a screen, we could shave 5-7mm easily. This would allow for an additional battery cell being added doubling the battery capacity. Thus the second generation of this product would stay at the same thickness of our current generation as well as have a drastically increased compute ability, more pleasant screen experience, and with the same long lasting battery life.

8. Appendix

8.1. List of References

- http://gbdev.gg8.se/wiki/articles/CPU_Instruction_Set
- http://gbdev.gg8.se/wiki/articles/CPU_Registers_and_Flags
- https://github.com/CTurt/Cinoop/blob/master/source/gpu.c
- http://derekmolloy.ie/kernel-gpio-programming-buttons-and-leds/
- https://blog.fazibear.me/the-beginners-guide-to-linux-kernel-module-raspberry-pi-and-led-matrix-790e8236e8e9?gi=d3d72bb5c899
- https://www.youtube.com/watch?v=Fj0XuYiE7HU
- https://www.youtube.com/watch?v=VxMV6wGS3NY
- https://www.youtube.com/watch?v=35YuILUlfGs
- https://www.youtube.com/watch?v=SpKK6o4ffts

8.2. Team Information

8.3. Project

8.3.1. https://git.ece.iastate.edu/sd/sdmay19-25

8.3.2. https://git.ece.iastate.edu/sdmay19-25/emu

8.3.3. https://git.ece.iastate.edu/sdmay19-25/mod

8.3.4. https://git.ece.iastate.edu/sdmay19-25/pcb

8.4. Members

8.4.1. Nicholas Losby

- Major: Computer Engineering
- Post-Graduation: Work at RSM

8.4.2. Sean Hinchee

- Major: Software Engineering
- Post-Graduation: Work at Microsoft

8.4.3. Jacob Nachman

- Major: Computer Engineering
- Post-Graduation: Work at Collins Aerospace

8.4.4. Nick Lang

- Major: Computer Engineering
- Post-Graduation: Work at Buildertrend

8.4.5. Matthew Kirpes

- Major: Computer Engineering
- Post-Graduation: Work at ACT