A Game Boy DMG-01 Replica

Jacob Malin
malin146@umn.edu
University of Minnesota - Twin Cities
Minneapolis, Minnesota, USA

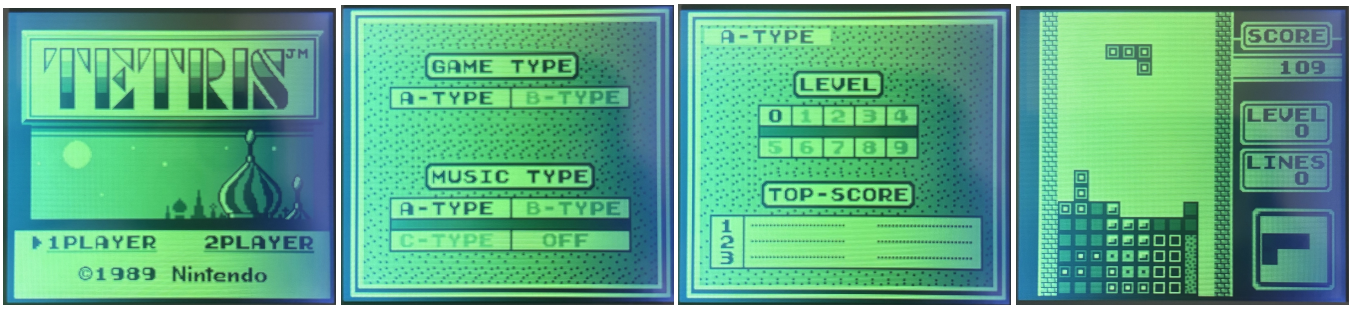

Figure 1: The game menus. From left to right: The title screen, the game selection screen, the level select screen, and the gameplay screen.

1 INTRODUCTION

My project is to replicate a Nintendo Game Boy DMG-01, the original Game Boy released in 1989. The project uses an AVR-BLE development board with an ATmega3208 as the microcontroller. The implementation features 8 input buttons, an LCD screen, an internal SD card, a power switch, and is battery-powered using 4 AA batteries. The device is coded with an implementation of Tetris based off of the version 1.1 American release for the Game Boy. This implementation has graphics upscaled from the reference and has the A-type marathon main game mode. The device is contained in a custom 3D-printed shell and contains a custom PCB that I milled and soldered myself. The AVR-BLE was also modified to fit within the case.

2 DESIGN AND IMPLEMENTATION

2.1 Hardware

The primary communication protocol used between components is the SPI protocol. With the microcontroller as the SPI host, it connects via a shared bus the display, the SD card, and the GPIO extender, which controls the button input.

2.1.1 Buttons. Given the limited number of GPIO pins on the AVR-BLE, an external device was needed to handle the 8 button inputs. The inputs are the 4 d-pad buttons (up, down, left, right) as well as A, B, Start, and Select. Originally an input shift register was considered however, since it does not include pull-up resistors, resistors would need to be soldered into the final product. As a time-saving measure, the GPIO extender was chosen instead since it contains internal pull-up resistors. Originally I was concerned about using a GPIO extender as the most common model MCP23008 uses I2C, which is a slower protocol than SPI, however I was subsequently able to find an alternative model MCP23S08, which uses SPI and was used in the final device.

Figure 2: The finished product.

Given that I would be using a GPIO extender for the buttons, I was hesitant about how the component would be soldered into the final project, however luckily or unluckily, the PCB that I was going to use for button input did not arrive, which inspired me to mill a custom PCB for the buttons, in which I was able to include a location to solder in the GPIO extender.

The plastic and silicone of the buttons (and power switch) are actually replacement parts created for the Game Boy DMG-01. Since I knew that producing those parts would be beyond the scope of

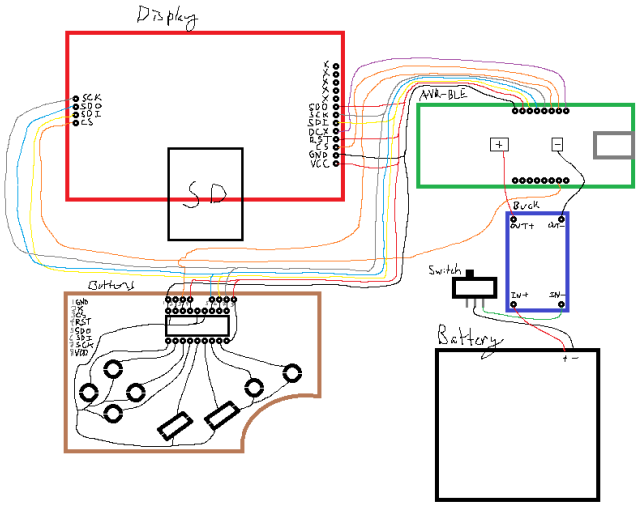

Figure 3: The hardware diagram.

what I could hope to achieve in this project, I knew that ordering these parts intended for the Game Boy itself would add a touch of polish to the product, and sell the impression that this is a replica of the original system. As a result, by buying professional replacement buttons, the button input feels very tactile and is very satisfying to press.

2.1.2 Display. The display this project uses is a 2.8-inch LCD screen that implements the ILI9341 specification. I was initially very hesitant to purchase this display given that the original DMG-01 display was 160x144 pixels. The ILI9341 is 240x320 pixels and has 262K color depth. It notably also has touchscreen features. All of this made me concerned that the microcontroller would successfully be able to drive this display. My concerns were not unfounded and indeed this display was uniquely difficult to drive, however in the end what caused me to buy this display was that it was one of the few commercially available displays that had the correct size for this project, to be at least as large as the original Game Boy screen. Indeed, the display was the perfect size to barely fit within the case, even requiring me to shave a few millimeters into each side of the case so that it would fit. The display also came with pins that had to be desoldered and removed so that it would not collide with the AVR-BLE.

Originally there was also a plan to use a red LED as a battery indicator on the side of the display, as is present in the Game Boy DMG-01. However, the size of the display prevented this possibility. However, by luck, the display was exactly too large and an unused portion of the display overlapped with the battery indicator. The screen itself was able to be used as a battery indicator in place of the LED.

Attached to the display came an SD card holder which became indispensable in order to hold the massive amounts of data that was required for the graphics of the Tetris game. A 16GB microSDHC card with a microSD to SD card adapter was used within the project. Notably, nowhere near 16GB was necessary for this project, but instead, the 16GB version was purchased because it was cheap. There was originally hope that the graphics data would be able to fit within the 32KB program data, however, this proved impossible very early into the project, and by the end, 25KB of program memory was utilized even with offloading to the SD card.

Finally, in a similar line of thinking to the buttons, a screen protector replacement part from the original Game Boy was purchased, which perfectly hides the size of the display and significantly increases the similarity in appearance to the original Game Boy. One of my strongest accomplishments in this project was the high degree of polish that I was able to achieve with the looks of the device from the front.

Since the AVR-BLE pin holders were too large to fit in with the display, the plastic portions were removed, and wires were soldered directly onto the AVR-BLE.

2.1.3 Power. Thankfully, all components, the display, the SD card, and the GPIO extender were able to use 3V3 power, which is supplied by the AVR-BLE. However, 4 AA batteries, as the original Game Boy had, produce 5V. This was remedied by using a buck converter. The buck converter was first attached to the battery holder and then tested using a potentiometer to adjust the device to 3V3 voltage. The battery holder and buck converter were then soldered into the device in series with the power switch, and the output from the buck converter was then soldered into the back of the AVR-BLE. In order to make the AVR-BLE fit within the space, the coin battery holder was removed, which allowed easy soldering of the buck converter. The reason that the power switch was wired in series with the buck converter and power supply was to prevent the passive power drain of the buck converter and to maximize battery life while the device is powered off.

Figure 4: A closeup of the cartridge.

2.1.4 Case. The original 3D file for the case was borrowed from the internet, however, the model was incomplete, so major modifications were required in order to make the model viable for this project. The slot for the power switch was modified to fit the switch that I had purchased and was also modified to fit the original plastic switch component. Much of the support for the button PCB was modified and holes were added so that it could be screwed in. Screw holes were added for the display and AVR-BLE, as well as for the model to be able to be screwed shut. The ports along the side of the Game Boy DMG-01 were removed, and a port was added for access to the AVR-BLE for easy modification of the software after assembly. Finally, the battery holder was modified to fit the larger battery holder that I had purchased.

The case was printed using resin printing, which allowed for a very smooth finish. Several prototype and backup prints were made in the event of the failure of the resin print.

While the finished product is not intended to be able to emulate real Game Boy cartridges, to complete the aesthetic, a fake cartridge was also 3D printed using FDM printing. A sticker was also printed out and affixed to sell the illusion.

2.2 Software

Given that the game required simultaneous button input and game loop, freeRTOS proved very useful. As well it added very simple code for timing the game loop, which helped significantly. The code is mainly split into the buttons, display, window, and Tetris modules.

2.2.1 Buttons. One of the two main tasks is the Buttons task. In the initialization, the extender is set so that all 8 pins are inputs and all 8 pull-up resistors are enabled. A freeRTOS queue is also created to pass button down/up commands to the Tetris task. During the buttons task, the GPIO extender is polled every 50ms for the current button state. If any button states have changed a button command is pushed into the queue.

2.2.2 Display. The display module assumes that the graphics are made up of an 18x20 grid of cells. Unlike the original Game Boy where each cell is 8x8 pixels, each cell is 13x13 pixels. Each screen of 18x20 cells is stored in two blocks of SD memory. The cells are stored in sets of 12 cells per block of SD memory. Whenever a screen or a cell is requested, it is read from SD memory. If the block of cells is requested more than once in a row, it will not re-read from SD memory and instead read from the cache.

When creating the screens and cells, I downloaded reference images for the version 1.1 release of Tetris for the Game Boy. I then by hand used image editing software to upscale and clean up all 300 cell types and the 8 screens that I used for the game. I then used a Python helper to convert those images into c arrays of enumerations. Then I copied all of those arrays into SD memory while compressing the cells so that they would fit in minimal memory.

The LCD screen is so large that not all of it is in use. The overall size is 240x320, however, only 234x260 pixels fit within the size of the original DMG-01 screen. Because of this, all renders to the display are at an offset of 2 pixels down and 57 pixels to the right. This leaves ample space for my battery indication 'led' on the left of the screen.

2.2.3 Window. I am particularly proud of the Window abstraction which handles the logic for each of the game menus. By having a consistent interface for each window type, it allows for the windows to be swapped in a state-machine-like fashion. Each window has a changeover function, a button handle, and a render function. When loading a new window, the changeover function is called. The button handler is called once per button down or up, and the render function is called once per frame. The game runs at about 62.5 frames per second, this is slightly faster than the original Game Boy's 60 frames per second, but it is decently close.

2.2.4 Menus. There are many windows in the menuing system including: logo, copyright, title, select, and a-type select. All of these follow the specifications of the version 1.1 Tetris game. I am particularly proud of the animations that are included within these menus. For example, the Nintendo logo scrolls down when the game first boots, all menu items when selected will flash between black and gray, and the score entry system will flash between black and white to show when a character is selected.

I took a lot of care when designing the windows to have the minimal renders possible at any given time. Even with the CPU clocked to full 20MHz speed and with SPI running at 10MHz, the display is still noticeably slow when rendering the full screen. Rendering the entire screen can take up to 1 second, which is significantly too slow considering the menus run at 62.5 fps which is renders on an interval of 16ms. As such, I only ever render to the entire screen during the changeover between windows. Every other render to the screen only changes the cells that need to be. For example, when the user hits the right button on the title screen, only two cells are written, the arrow is moved to the two-player text and the arrow is erased in the one-player location. Then until the user hits another button, no further writes to the screen are made.

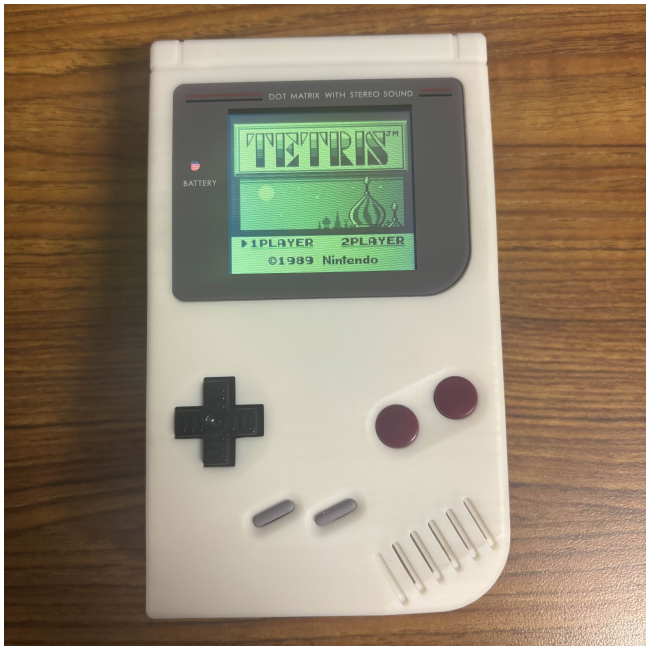

Figure 5: The front view.

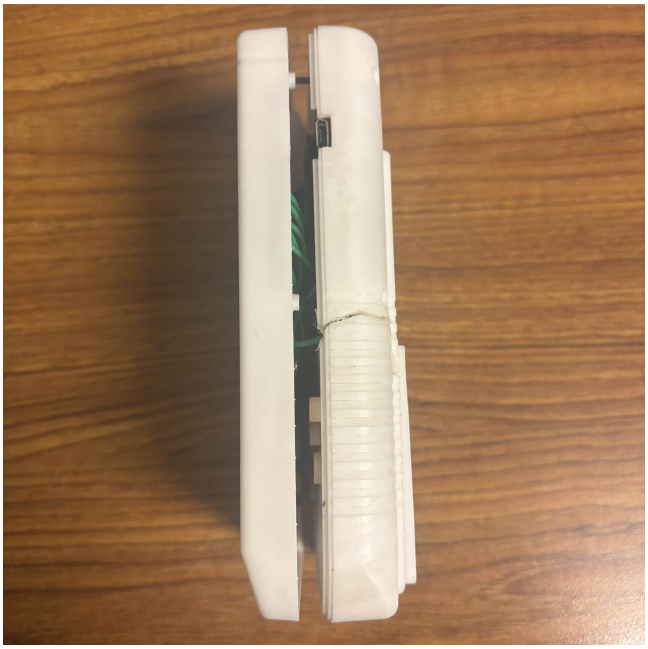

Figure 6: The side view.

Many of the menus, in order to facilitate minimal writes to the display, maintain two copies of their data, and during the render phase, check for any changes made to its data since the last render, and make spot writes to the display based only on what is necessary. While this saves a lot of time writing to the display, it causes a lot of complicated code, especially for animated features such as the blinking menu items.

2.2.5 Gameplay. The gameplay implements the full set of features available in the a-type game mode on the version 1.1 release. This includes tetrimino movement, rotation, soft-dropping (which moves tetriminoes down faster), delayed auto shift (which allows for holding down of the left and right movement), a pause menu, and enable/disable of the next tetrimino preview. Only the active tetrimino and the next tetrimino are 'entities' all other blocks are added to the board state once they lock into place. This board state is checked every time a piece locks into place for a line clear, awards points to the user, and plays a line-clearing animation. The point system follows the original release with points awarded for rows soft-dropped and points awarded based on the number of lines cleared and the level. Every 10 lines cleared (depending on the starting level as well), the level is increased, which speeds up how fast the pieces drop. The game ends once a piece locks into place without having moved from the spawn location.

On game end, the game end animation and screen are played, and once the user hits a or start, score/name entry mode is activated. This allows the user to enter their name if they achieved a high score for that level. Following the specification of the original Game Boy, the scores are not saved after power off.

To ensure the pieces are dropped in a truly random order, a random number generator is used in combination with a seeding system inspired by the Game Boy Tetris release. Every time the user presses a button, the current time in the TCB0 count register is added to the random number generator. Since no two human users will be capable of hitting buttons in the exact same way at the same time, this produces effectively random numbers.

3 FINAL PRODUCT

The final result of my project is a working Game Boy replica that can play Tetris and can be used portable, relying on battery power. I had originally planned to make a smaller version of a Game Boy, but that goal quickly shifted into a replica of a Game Boy. However, I am more than pleased with this result. I was able to achieve my initial goal of having a working display, 8 buttons, and a power switch. As well I was able to achieve my goal of learning how to 3D print and how to solder. As well, though it was not one of my goals I learned how to print a circuit board. I had also set out to use freeRTOS, which I did use. Finally, I had mentioned that I would overcome the challenge of only having 4KB of memory, which I was successfully able to overcome.

One of the sacrifices that I made was I was unable to add music/sound effects to the game. I do have a location in the 3D print for the speaker to be housed, however, I ran out of time to program the functionality. Notably, I also ran out of RAM as well. When I attempted to add a third freeRTOS task, the program would no longer boot the menu.

4 REFLECTION

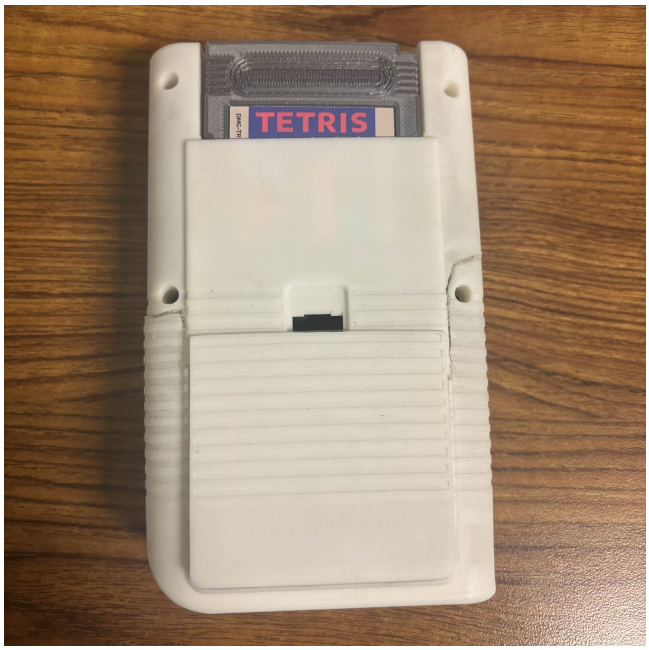

Figure 7: The back view.

I should have spent way more time prototyping my 3D print. Unfortunately, for a while I was waiting for my button PCB to arrive, and because of that I delayed further prototyping so that I could save on the cost of 3D printing prototypes. Because of this, I

did not have time to do a prototyping round before my resin print, which means that many mistakes made it into the final product. To list a few, I cracked the back of the case because the battery holder that I had extended did not have enough space to hold the battery holder. As well, when adjusting the battery holder, it caused the cartridge slot to become too small, and the battery cover to not fit into the slot. These problems were fixed by clipping off pieces of the case, but this was not my preferred solution. Had I more time, I would do 1-2 more rounds of 3D printing.

Also, I had not accounted for how much space is taken up by wiring. I used too much wiring for the size of the case, and as a result, I was unable to fully close the case. Were I to redo the soldering, I would cut the size of the wires down or perhaps mill a PCB for the entire system, to save on space.

Also, I ran out of time to add a location to screw in the buck converter, which was unfortunate. Thankfully, it is all packed in so tight that the buck converter is not moving around, but it is still not ideal.

Near the end of the software design process, I was running out of time to create abstractions. For example, one useful module would have been a board module, however I ran out of time to implement one. As such, there is a lot of shared/hard-to-read code for board functionality. It would have also been nice to have more time to cut down on memory usage since I feel if I had more time, I would have been able to implement both the music and the B-type game.