Gameboy flash cartridge

Contents

[Foreword 3](#_Toc48245713)

[Hardware modifications 4](#_Toc48245714)

[Battery mod 4](#_Toc48245715)

[Screen mod 5](#_Toc48245716)

[Cartridge Hardware 6](#_Toc48245717)

[General workings 6](#_Toc48245718)

[Bus decoding 7](#_Toc48245719)

[SD card communication 9](#_Toc48245720)

[Cartridge firmware 10](#_Toc48245721)

[Using FreeRTOS 10](#_Toc48245722)

# Foreword

This document is meant as a place for me to note any research made for myself and others. It is meant as ‘proof’ for the concepts I’ve learned about during the making of this project and as a guide for other people to better understand the choices I’ve while making this project.

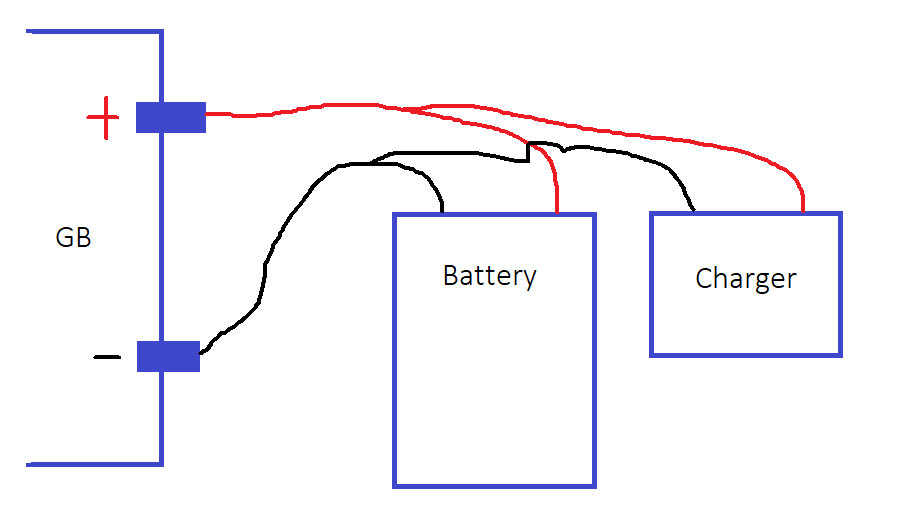
# Hardware modifications

## Battery mod

As I’ve stated in my General document, I bought a GBC as a way to play older games on older hardware. One thing I hate about the GBC, is the fact that the thing uses AA batteries. That’s why, I want to modify the hardware by adding a rechargeable battery. This should allow me to use the GB more comfortably and longer.

I mainly got the information I needed from a YouTube video (<https://www.youtube.com/watch?v=tGUx15ZgTeQ>). This is where I got the information to use a rechargeable Li-ion battery and a Li-ion battery charger circuit.

As I had these laying around at home, I didn’t have to buy them. However, I was scared about the voltages used. The battery itself usually sits around 3.7V (as Li-ion batteries normally do) and the charging voltage is 4.2V. The GB normally uses 2 AA batteries, which connected in series have a maximum voltage of around 3V. Because of the way the battery and charger are connected (see picture below) it puts the charging voltage on the pins that normally only take those 3V.

  
Image depicting battery and charger connections to GB.

I didn’t know it that was a problem, so I took it to the internet. From [this](https://blog.gg8.se/wordpress/2009/07/22/testing-gameboy-input-voltage-ranges/) source, I got the battery and charger voltages are within normal range and I basically wouldn’t have any trouble with the charging voltage. The article spoke about having connected the GB with 9V for an extended period of time without any hiccups, so I trust it won’t be a problem to connect 4.2V. I’ve connected by GBC to a power supply of 4.2V without any trouble as well.

**TODO: Connect GB for a long time with battery circuit**

**TODO: Connecting LED battery indicator**

**TODO: Give some metrics (charge time, discharging time)**

## Screen mod

A great mod I saw in a video, was a screen mod. In it, the original screen is replaced with a modern backlit screen. The original GBC screen has no backlight illuminating the screen. Because of this, the GBC has a better battery life, but sacrifices visibility. A user must hold the GBC by a light in EXACLTY the right angle in order to maximize brightness, whilst minimizing glare.

As this is quite annoying and because I want to be able to game in a dark room, I chose to order [this](https://www.aliexpress.com/item/4000338739866.html?spm=a2g0s.9042311.0.0.7b6c4c4dyz4Ypk). It is a GBC compatible backlit screen with included case. Because of the battery mod, you need to modify the case and I didn’t want to ruin the original casing (and I was missing the battery door). There is a version without case, if you’re interested.

Instructions aren’t really given to install the screen, so here’s what I did:

* Connect screen to check orientation and if it works or not (very important)
* Fit screen inside the case and test monitor. As the screen is a bit smaller, it needs to be fitted using sticky foam. Be careful that the borders don’t overlap with the screen and that the screen is parallel with the case.
* When happy with screen placement, you can close the GBC. The screen has a capacitive button, that you need to place in front of the IR LED’s. Using this button, one can change the brightness of the screen. The board probably has some pins you can connect to the buttons, if you don’t want to sacrifice the IR interface (haven’t looked into this, as I don’t care about the IR LED’s, might change it in the future).

   
Image of screen mod part. Taken from <https://pt.aliexpress.com/item/4000338739866.html?spm=a2g0s.9042311.0.0.7b6c4c4dyz4Ypk>

# Cartridge Hardware

## General workings

At first, I need an general concept how a flash cart could work. To get a general idea, I took to the internet to get inspiration. A simple cartridge has a ROM (Read Only Memory) chip and some circuitry. The game’s code and data is saved on the ROM chip. When the GB tries to read data from the ROM, there’s usually some logic that decodes the signals from the GB, that in turn retrieve the data from the ROM and passes it to the GB. This is called a MBC, or Memory Bank Controller.

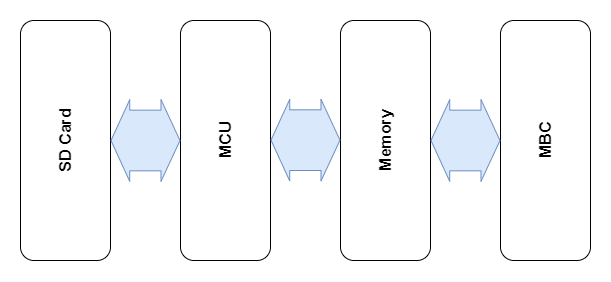
A cart can also contain additional RAM. This RAM is usually battery backed, which means the RAM retains data after the GB has been powered down. This is very useful for save data, as the GB can read and write to this memory. And again, there’s usually some logic between the GB and physical memory.

I lost where I got the information from, but when you look at the different links on the [Gameboy Dev site](https://github.com/gbdev/awesome-gbdev), you can piece it together.

In a traditional flash cart, the games and saves are stored on an SD card, but are not accessed directly by the GB in that way. As I’ve discussed in my research in the ‘Bus decoding’ part, the GB can’t access the SD card directly and needs an intermediate. That’s where I got the idea of using RAM memory as a memory buffer for the ROM and RAM that would normally go on a cart. The RAM can be used to replace the ROM, as they work similarly. I also need to frequently change the contents of the “ROM” (every time when a new game is loaded), so using RAM is pretty advantageous, as it can withstand a lot or reads/writes.

It would also be wise to have a firmware based MBC. This would enable me to implement the different MBC in firmware, rather then have it be hardware. I’ve looked on the internet and stumbled upon things called FPGA’s. These are devices with programmable logic units that can preform basic operations. These blocks can be chained together, in order to preform a multitude of operations. It’s basically hardware you can program. Because reprogramming the FPGA probably needs a dedicated programmer, it probably won’t be possible to change the firmware by and end user (without experience in the field), but that’s a compromise I’ll have to make.

I think that the best approach, would be to combine an FPGA with a MCU (Microcontroller Unit). The FPGA would then be used to act as the MBC and communication point for the GB, whilst the MCU would do more complex stuff (communicating with SD card, loading games, etc). Using a MCU also enables me to base the system on firmware, which would make the system very flexible. The firmware can then be loaded from the SD card, enabling the option to easily update firmware (also by end user). The FPGA probably can’t be changed on the fly, so I should give it a lot of options, in order to increase flexibility (options like direct communication with MCU, selection of MBC, etc).

  
Diagram depicting general parts and interactions between components.

## Bus decoding

The GB communicates with the cart via a 32-pin parallel interface. This interface is made up of these components:

* CLK, which gives the timing for a read/write
* RD, which signals that the GB is trying to read from the cart
* WR, which signals that the GB is trying to write to the cart
* CS, which signals that the GB wants to communicate with the cart
* A0-15, a 16-bit address bus, used to signal which address the GB is trying to read from or write to.
* D0-7, an 8-bit data line, used to send or receive the data written to or read from.
* Reset, used by the cart to reset the GB

The way the communication works, is by specifying at what address a read or write should be performed with the given data.

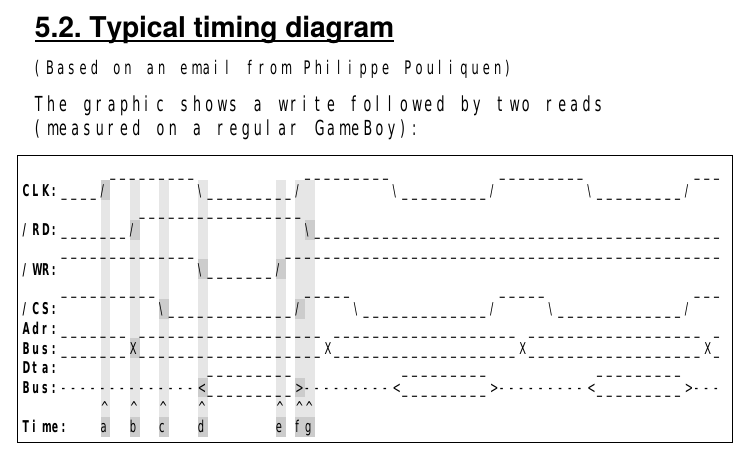
For example, a read is done by putting the address from which you want to read on the address bus and specifying a read operation with the control signals (RD, WR, CLK & CS). Then the cart will output the data on the data bus, which the GB can read from.

A write is done in a similar manner. The GB puts the address it wants to write to in the address bus and specifies a write operation using the control signals, while outputting the data to be written onto the data bus.

Using these operations, the GB can effectively communicate with the cart, by reading from and writing to it. When making the hardware for the cartridge, we need to decode the signals being sent to the cart and handle the requested operation.

I’ve looked at [this](https://dhole.github.io/post/gameboy_cartridge_emu_1/) article, where a person by the name of ‘Dhole’ used a STM32F4 board to emulate a cart. In that project, Dhole used a digital analyser to analyse the cart port and deduce what the different signals meant. Since I don’t have a digital analyser, I used this as a reference of which signals I’ll need to manipulate. Below is an image and graph for more clarity.

**TODO: Try communicating with GB with prototype board**

  
Image showing signal timing diagram. Source: <http://marc.rawer.de/Gameboy/Docs/GBCPUman.pdf>

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Operation | RD | WR | CS | Address Bus | Data Bus |
| Read | L | H | L | X | X |
| Write | H | L | L | X | X |
| Ignored | X | X | H | X | X |

Truth table for control signals. Made by analysing article by ‘Dhole’

**TODO: Test truth table**

# Cartridge firmware

## Using FreeRTOS

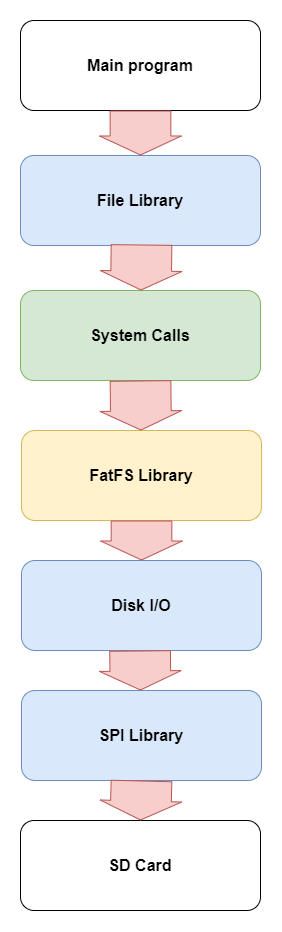
## SD card communication

As per the user requirements, I want to use an SD card to save games and additional data. To use the SD card, I need a communication peripheral and some firmware to go along with it. This includes:

* Low level communication (sending raw bytes)
* SD card communication protocol
* Handling the filesystem

My original plan was to make this from scratch, as I wanted to have full control over the firmware and I didn’t want to deal with errors due to improper synchronisation with other tasks (see [Using FreeRTOS](#_Using_FreeRTOS)).

I started with implementing some SPI functions to use with the SD card. The SD normally has 2 communication modes: SD mode and SPI mode. SPI mode is very useful in our application, because SPI hardware is very common for MCU to have. If you’re not acquainted with SPI, I highly recommend [this](https://www.youtube.com/watch?v=fvOAbDMzoks) video by ‘GreatScott’ explaining it.

I found 2 very helpful articles, explaining how SD card work with SPI which I mainly used for my implementation. You can find these articles [here](http://www.dejazzer.com/ee379/lecture_notes/lec12_sd_card.pdf) and [here](http://elm-chan.org/docs/mmc/mmc_e.html) (first one written for the Electrical Engineering Department of University Buffalo, the second one written by a user named ‘CHaN’). Anyhow, I used these sources to write an SPI driver and I started making an SD communication library. However, things were taking quite long and when I looked at what I still had to do, I started to look for libraries.

I used a library in the past called ‘FatLib’ by GitHub user ‘gallegojm’, which is a FAT-FS library to use on an Arduino to communicate with an SD card (repository can be found [here](https://github.com/gallegojm/Arduino-FatLib)). It’s a nice library with a lot of different functionalities. When I was looking how to port it though, I found out that the program I was using to program my prototype board (PSoC Creator) didn’t support C++, so that plan went out the window. I did find out that FatLib library used a different library to handle all FS logic, the ‘FatFs’ library written by ‘CHaN’.

When I was looking for information, I stumbled upon an article by Balau, explaining how to access the SD card using SPI on the STM32 Nucleo. Here he explains that usually when implementing FS, there is a layered model to go by (see image left). The idea is to separate all the logic that goes into using an SD card, into layers that can be interchanged. If I follow this layered model, I won’t have to write FS logic (yellow). I’ll only need to write the things in blue and design an interface for the OS to handle file requests (green).