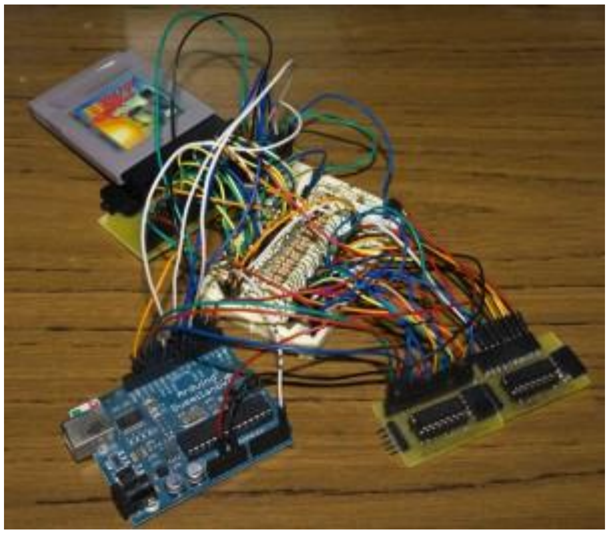
Recently I’ve been collecting retro gaming consoles such as the Gameboy and SNES and whilst playing a game on the SNES I actually lost all my saves when I turned it off (turns out the battery got disconnected from the cartridge). The thought came to me, how can I backup the save game from these devices? There are cart readers around but they seem to be harder to find in this day and age so I thought why not make one myself using the Arduino as it’s so versatile!



Before I begin looking into extracting the save games I’ll look into dumping the ROM which was a way for me to learn how to communicate with the Gameboy cartridge. I’ll now guide you in how to communicate with the Gameboy cartridge to read the ROM so without further delay, let’s begin!

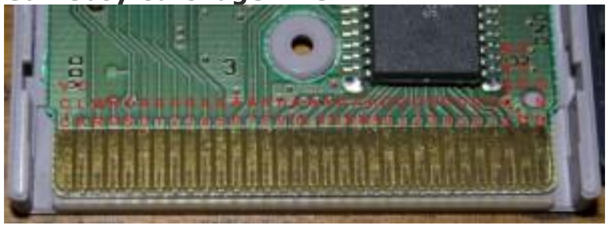
Gameboy Cartridge

Let’s firstly take a look inside a Gameboy cartridge. A typical Gameboy cartridge contains:

- ROM is where the game’s data is stored

- SRAM is where your save games/high scores are keep. Some cartridges (like the one of the left) don’t have this chip because they don’t store any data or it’s built into the MBC
- MBC is the memory bank controller, it allows us switch ROM banks to read the game’s data from the ROM.
- MM1134 IC to control when SRAM should be run from battery or not
- 3v coin cell

Gameboy Cartridge Pins



You can see it has 32 pins and those pin functions are:

- VCC – Power (5 volts)
- CLK – Clock signal (not used)
- ~WR – if low(grounded) and if RD is low, we can write to the SRAM and load a ROM or SRAM bank
- ~RD – if low (grounded) and if WR is high, we can read the ROM and SRAM
- CS_SRAM – if enabled, selects the SRAM
- A0 – A15 – the 16 addresses lines that we tell the ROM which particular byte of data we want to read
- D0 – D7 – the 8 data lines that we read the byte of data selected by the 16 address lines. These data lines can also be used to control which ROM bank to load (important for later).
- Reset – needs to be connected to VCC
- Audio in – (not used)
- GND – Ground

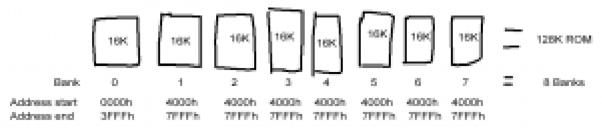
For reading the ROM we are only interested in the WR, RD, A0-A15 and D0-D7 lines.

Gameboy Address

The Gameboy can only address 16bits (16 address lines: A0-A15) which means that the number of combinations of addresses is 2 ^ 16 = 65,536 with 1 byte for each address means that 65,536 bytes can be read/written to. These 16 address lines also control the Gameboy's internal RAM, etc and so they decided to assign the addresses 0000h to 7FFFh for the Gameboy's ROM Cartridge which gives us 32,767 bytes to work with. You can view the full list of addresses and what they correspond to here. Note: you will see that 0000h to 00FFh is not really part of the ROM.

In"0000h" and "7FFFh" the h stands for hexadecimal (hex) which is a way that's used to express the addresses instead of just writing the decimal number; you can convert to and from hex by using an online calculator or if in Windows by opening the calculator and choosing scientific mode.

32K bytes for a Gameboy game sometimes isn't enough so what they were able to do is have multiple 16K ROM banks and have a chip (MBC) to change the ROM bank if we requested it. We would then just re-read the same 16K address range and be presented with new 16K worth of data.



For example, we could have 8 banks x 16K bytes which gives us 128K bytes to work with. You could have the main code in the first 16K bytes then ask the MBC to switch to the next 16K bytes because you might have an image on the 2nd bank and then switch to the 4th bank of 16K as you might have text there. So you can see how by having a MBC they overcame the issue of only being able to address 32K bytes.

Visualising the use of hex as an address

I've mentioned that we use hex addresses instead of just writing the decimal number and we know that there are 16 address lines (A0-A15) but let's see how we visualise the conversion of hex to binary so that the appropriate address lines are set to 1 (high).

Let's take for example the hex address 0148h which is 328 in decimal but more importantly is 101001000 in binary. When we want to tell the ROM that we want to read from 0148h, we turn the address lines on or leave them off.

Address lines	15,	14,	13,	12,	11,	10,	9,	8,	7,	6,	5,	4,	3,	2,	1,	0
On (1), Off (0)								1,	0,	1,	0,	0,	1	0,	0,	0

As you can see the address lines are going from A0 right to A15 left and then we take our binary number and align it to the right. We would be turning on address lines 3, 6 and 8 which the ROM would read as 0148h; this is the basics of how address lines are used to tell the ROM the address we want to access.

Gameboy Cart Physical Header

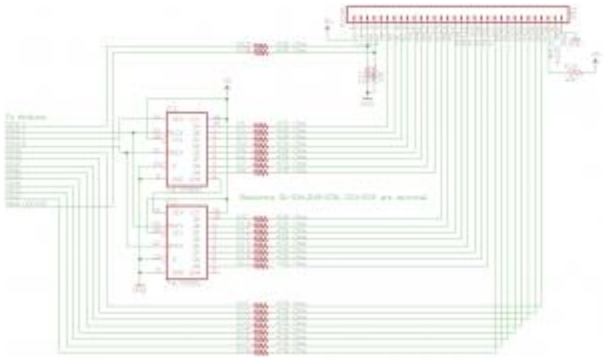
The only downside to this project was I had to rip the cartridge header from a Gameboy as I wasn't able to find any online shops still selling them! I just built a small PCB that I mounted the header to and then just female headers on the other side, it worked well. You can build your own Gameboy Cart Adapter here**.**

Setting up the Arduino

We need to connect the Arduino up to the Gameboy cartridge but we'll need 26 lines and the Arduino only has 13 digital pins. What we can do is use our old friend the Shift Register (more about Shift Registers here) to shift out the address A0-A15 lines for us. Now all we need is 26 – 16 (address lines) + 3 (shift register lines) = 13 lines which is achievable as we can set some analog pins on the Arduino to act as digital pins.

The data that we read from the ROM has to go somewhere right? We'll be using the serial to transfer the data back to the computer. Let me say that when setting up the Arduino as we are using serial

I’ve found that using digital pins 0 and 1 cause issues so we will avoid them, plus it says this in the Arduino Reference which I found out later.



- VCC we can connect to Arduino’s 5V pin.
- WR and RD which will be on digital pin 13 and analog pin 5 respectively. We will pull these to ground with a 10K resistor and we’ll connect a 470Ohm resistor to protect them when we do set them to high.
- A0 – A15 are connected through our shift registers. A0-A7 is handled by shift register 1 and A8-15 is handled by shift register 2. The outputs from the shift registers have 470 Ohm resistors (these resistors are optional but good if you are wiring this up for the first time. It appears the cartridge has some built in resistors somewhere.) We connect Clock to pin 12, Data to digital pin 11 and Latch to 10.
- D0-D7 will be on digital pins 2 to 9 respectively (D0 = digital pin 2, D1 is pin 3, etc). 470 Ohm resistors can also be placed on these lines for safety but once again they are optional.
- Reset is pulled up with a 10K resistor.
- GND is connected to Arduino’s ground.

Setting up the computer side

The reason for setting up the computer side is so that we don’t have to use the Arduino serial monitor to receive the data, I mean who would want to copy and paste the ROM to a file themselves? We can utilise Python and create a script that will open the serial port and place everything read into a file. Python for me was a tricky language to deal, it’s not quite like C or Java.

Firstly download and install Python 3.2 and pySerial, then download myiG_GBCartRead_ROM_Only_v1.0 script. To explain my script in short, it opens the serial port, waits for the START command, converts the serial data received and writes it into a file called myrom.rom then stops once the END command is received.

Cartridge Header

Before we dive into reading the ROM we need to know firstly which MBC the cartridge uses and how many ROM banks of 16K bytes there are, this is done by reading the cartridge header. There are many other bits of useful information in the header such as the game title, ram size, etc. A more extensive list can be found on page 56 of TheNintendoGameboy PDF, this PDF document is also worth keeping handy.

0147h – Cartridge Type (MBC type)

This address defines which MBC the cartridge uses.

0 – no MBC

1 – MBC1

2 – MBC1+RAM

etc

0148h – ROM size

At this address, the number of ROM banks is specified and is done by multiplying 32K by 2 times the number we read. There are exceptions to this rule.

0 – 32K byte

1 – 64K byte (i.e. 32K x 2 = 64K, 16K banks so 4 ROM banks)

2 – 128K byte (i.e. 32K x 4 = 64K, 16K banks so 8 ROM banks)

etc

Read the first 16,383 bytes of the ROM – Bank 0 (0000h – 3FFFh)

The first ROM bank 0 ranges from 0000h to 3FFFh and no switching of banks is required. Bank 0 will always be at this address and is like this way so that we can always read the Cartridge header no matter which MBC the Gameboy cartridge uses.

Alright we've learnt a fair bit so let's jump into some code to read the first bank of ROM, here's the download you will need to upload to the Arduino: iG_GBCartRead_ROM_Bank_0

?

```
int latchPin = 10;
int dataPin = 11;
int clockPin = 12;
int rdPin = A5;
int wrPin = 13;
```

Just our standard assignment of pins.

?
```
void setup() {
 Serial.begin(57600);
 pinMode(latchPin, OUTPUT);
 pinMode(clockPin, OUTPUT);
 pinMode(dataPin, OUTPUT);
 pinMode(rdPin, OUTPUT);
 pinMode(wrPin, OUTPUT);

 // Reads D0 - D7, set as inputs
 pinMode(2, INPUT);
 pinMode(3, INPUT);
 pinMode(4, INPUT);
 pinMode(5, INPUT);
 pinMode(6, INPUT);
 pinMode(7, INPUT);
 pinMode(8, INPUT);
 pinMode(9, INPUT);
}
```

We now set begin the serial connection, set the pins used to talk to the shift registers and RD/WR pins as outputs. We set the digital pins 2 to 9 as inputs as they need to read the data that comes from the cartridge connectors D0 – D7.

?
```
void loop() {
  unsigned int addr = 0; // Variable ranges between 0 to 65,536
```

Now for the start of the main loop, we set up our address variable as an unsigned integer and it needs to be unsigned because we have a 16 bit address that is 2 ^ 16 = 65,536, so we range between 0 and 65,536. If we didn't put it as unsigned the range would be -32,768 to 32,768 which would really mean 0 to 32,768 for us as we only deal with positive addresses (this would be fine because we never go over 32,768 but we'll do things right). Thus once it would reach 32,769 it would "overflow" and actually become -32,768!

?
```
digitalWrite(rdPin, LOW); // RD 0
 digitalWrite(wrPin, HIGH); // WR 1

 Serial.println("START"); // Send the start command
```

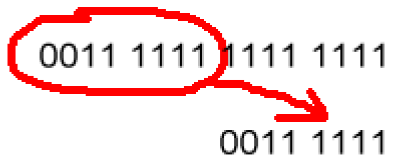
We set RD to low and WR as high in order to read the ROM, you can check what to set RD/WR to here.

After that we send the text "START" by serial over to our computer.

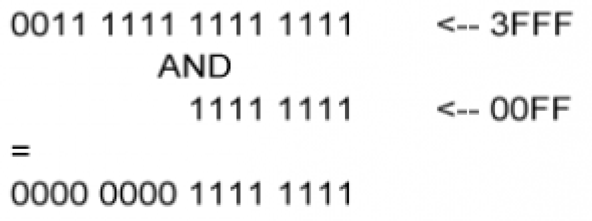
?

```
// Read addresses 0000h to 3FFF, i.e. ROM bank 0
 for (addr = 0; addr <= 0x3FFF; addr++) {
 digitalWrite(latchPin, LOW);
 shiftOut(dataPin, clockPin, MSBFIRST, (addr >> 8)); // Most left 8 bits
transferred
 shiftOut(dataPin, clockPin, MSBFIRST, (addr & 0xFF)); // Most right 8 bits
transferred
 digitalWrite(latchPin, HIGH);
 delayMicroseconds(50); // So ROM can process our request
```

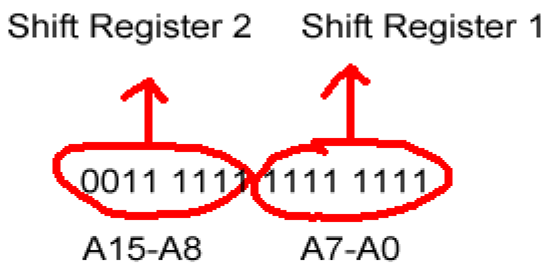
We begin our for loop used to read every address between 0000h and 3FFF which is ROM bank 0, so we set up x as 0000h and the end as 3FFF. Next is the standard Shift Register code, pull the latch to low, shift out the address and pull the latch to high. We delay 50 microseconds so the ROM has had enough time to register our request.



For example, lets say the address is 3FFF (0011 1111 1111 1111). What you'll notice is that we do "x>>8", this means only give us the "most left" 8 bits and does this by moving the bits to the right by 8; we have a 16 bit address so moving all the bits to the right 8 times which means we are left with the "most left" 8 bits only. The other "1111 1111" bits have moved to the right that they have "disappeared".



The next part "x & 0xFF" we do an AND operator on the "most right" 8 bits. 1 AND 1 = 1, 0 AND 1 = 0. As we didn't specify a 1 on the 8 "most left" bits this basically gets rid of them and we are left with only the 8 "most right" bits.



So we've now split up the 16 bit address into 2 x 8 bit parts which we send to each shift register and then they send it to the A0-A15 addresses.

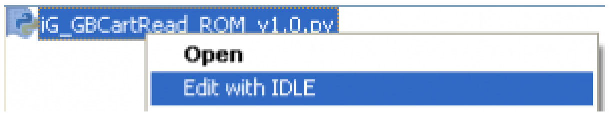
?
```
byte bval = 0;
for (int z = 9; z <= 2; z--) {
  if (digitalRead(z) == HIGH) {
    bitWrite(bval, (z-2), HIGH);
  }
}
Serial.println(bval, DEC);
```

We now read the 8 bits coming from the D0 – D7 address using the for loop. If we find any bit that is on we use bitWrite to set that bit to high on our bval variable, basically we are converting the hardware bits we are receiving from D0 – D7 into software bits in the variable bval. We then print out bval as a decimal number.

?
```
}
Serial.println("END");

while(1);
}
```

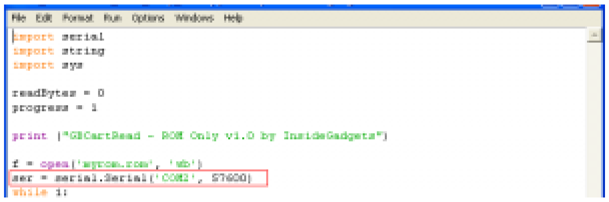
After the code has read all the addresses, we print out END and then just stop.

So now it's time to test it out, go ahead and upload the downloaded "iG_GBCartRead_ROM_Bank_0.pde" file to your Arduino.



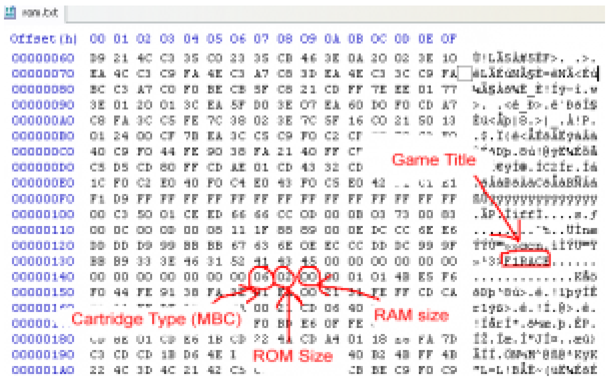
Now open up the script by right clicking it and selecting "Edit with IDLE".

You will need to make a change in the script so the serial connections to the correct COM port on your computer, for me it's COM2. You can find this out from the Arduino software by going to Tools -> Serial port when the Arduino is plugged in. Once you have made the change, hit F5 to run the script.



```
import serial
import string
import sys

readBytes = 0
progress = 1

print ("GBCartRead - ROM Only v1.0 by InsideGadgets")

f = open('myrom.rom', 'wb')
ser = serial.Serial('COM2', 57600)
while 1:
```

You should see that some hashes should start printing every few seconds, once it's done it will stop.



```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000060   D9 21 4C C3 35 C0 23 35 CB 46 3E 0A 20 02 3E 10
00000070   EA 4C C3 C9 FA 4E C3 A7 CB 3D EA 4E C3 3C C9 FA
00000080   BC C3 A7 C0 F0 BE CB 5F C8 21 CD FF 7E EE 01 77
00000090   3E 01 20 01 3C EA 5F D0 3E 07 EA 60 D0 F0 CD A7
000000A0   C8 FA 3C C5 FE 7C 38 02 3E 7C 5F 16 C0 21 50 13
000000B0   01 24 00 CF 7B EA 3C C5 C9 F0 C2 CF [illegible]
000000C0   40 C9 F0 44 FE 90 38 FA 21 40 FF CF [illegible]
000000D0   C5 D5 CD 80 FF CD AE 01 CD 43 32 CD [illegible]
000000E0   1C F0 C2 E0 40 F0 C4 E0 43 F0 C5 E0 42 [illegible]
000000F0   F1 D9 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00000100   00 C3 50 01 CE ED 66 66 CC 0D 00 0B 03 73 00 83
00000110   00 0C 00 0D 00 08 11 1F 88 89 00 0E DC CC 6E E6
00000120   DD DD D9 99 BB BB 67 63 6E 0E EC CC DD DC 99 9F
00000130   BB B9 33 3E 46 31 52 41 43 45 00 00 00 00 00 00   F1RACE......
00000140   00 00 00 00 00 00 00 06 02 00 00 01 01 48 E5 F6
00000150   F0 44 FE 91 38 FA [illegible] FE FF CD CA
00000160   [illegible] 00 [illegible] CD 06 40 [illegible]
00000170   [illegible] F0 [illegible] E6 0F FE [illegible]
00000180   [illegible] 01 [illegible] E6 1B [illegible] CD A4 01 18 [illegible] FA 7D
00000190   C3 CD CD 1B D6 4E [illegible] 40 B2 48 FF 4B
000001A0   22 4C 3D 4C 21 42 C5 [illegible] CB BE C9 F0 C9
```

Now let's read the file called myrom.rom with a hex editor, the one I use is called HxD Hex Editor. If you scroll down just a tiny bit the data we'll see is the cartridge header, can you see the game title? It's located at 0143gh. The cartridge type (MBC) is on 0147h and the ROM size (which is important to us in order to read the whole ROM chip) is at 0148h also the RAM size is at 0149h.

Read the next ROM bank (4000 – 7FFFh)

So we have read the first 16,383 bytes of the ROM and now we need to switch ROM banks so we can read the next 16,383 bytes (4000 to 7FFF) and so on. Here's the download you will need to upload to the Arduino: iG_GBCartRead_ROM_Banks

We complete the following steps for MBC2; the only difference between the MBCs for ROM reading is what address you need to write to in order to switch banks, it's a good idea to keep this link handy.

1. Put the WD pin to high
2. Select the specific address that allows us to change ROM banks, in our case 0x2100 (10 0001 0000 0000 which means A8 and A13 are enabled)
3. Use the D0-D7 lines to write the ROM bank number we wish to use, e.g. 0000 0001 would select bank 1, 0000 0010 for bank 2, etc.

4. Put the WD pin to low so that we can begin to read the ROM bank
5. Read the address 4000 to 7FFF which will contain the ROM bank we requested

We repeat steps 1 to 5 and specify the next ROM bank number in step 2 until all ROM banks have been read. If you recall the ROM size of my cartridge of F1RACE is 128Kbytes which means 16K x 8 banks, so we ourselves have to stop once we reach the 8th ROM bank otherwise we will receive duplicate data.

Did you notice something weird with Step 3? Remember that we actually used D0-D7 before to read the data from the ROM but now we write data to it. What happens is that the MBC knows that we can't write to the ROM so it intercepts our request and knows that we want to change ROM banks.

Now for the code, we modify the main loop a bit and add a new function to switch ROM banks.

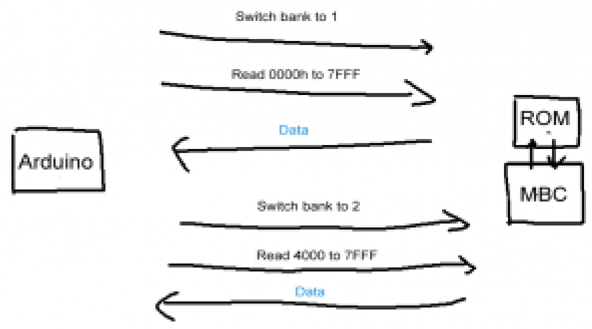
?
```
void loop() {

 unsigned int addr = 0; // Variable ranges between 0 to 65,536
 Serial.println("START"); // Send the start command

 int romBanks = 8; // Change to the number of ROM banks
```

We've now added a variable called romBanks which you need to change to the ROM banks to the number your Gameboy cartridge has (remember do this by reading Cartridge header – ROM bank 0).

?
```
// Read x number of banks
 for (int y = 1; y < romBanks; y++) {
 selectROMbank(y);
 if (y > 1) {
 addr = 0x4000;
 }

 // Reads addresses 0000h to 7FFF in the first run, then
 // only 4000h to 7FFF in the second or more runs
 for (; addr <= 0x7FFF; addr++) {
   digitalWrite(latchPin, LOW);
   shiftOut(dataPin, clockPin, MSBFIRST, (addr >> 8));
   shiftOut(dataPin, clockPin, MSBFIRST, (addr & 0xFF));
   digitalWrite(latchPin, HIGH);
   delayMicroseconds(50);

   byte bval = 0;
   for (int z = 9; z >= 2; z--) {
     if (digitalRead(z) == HIGH) {
       bitWrite(bval, (z-2), HIGH);
```

```
      }
    }
    Serial.println(bval, DEC);
 }
 Serial.println("END");

 while(1);
}
```

Now we have a for loop that loops 8 times because my ROM has 8 banks.



What it does is in the first run when y =1, the selectROMbank method selects ROM bank 1 because we will actually be reading two ROM banks (0000h to 7FFF) in one go in the first run. Remember that Bank 0 is always at 0000h to 3FFF and Bank 1+ is 4000h to 7FFF. After the second or more runs, y is 2 (or more) which activates the if statement to change the address to 4000h because this is the address we use to read the other ROM banks.

Next is the for loop we had before except we don't assign the addr variable anything because we set that before in the if (y > 1) part, so all we do is set the end of the loop at 7FFF.

?
```
// Select the ROM bank by writing the bank number on the D0-D7 pins
void selectROMbank(int bank) {
 digitalWrite(rdPin, HIGH); // RD 1
 digitalWrite(wrPin, LOW); // WR 0
 delay(5);
```

Now we reach the selectROMbank function, it takes the bank number as the input. First thing it does is switch RD to 1 and WR to 0 as per the RD/WR page and then we just wait a little while.

?
```
// Set D0-D7 as outputs
 pinMode(2, OUTPUT);
 pinMode(3, OUTPUT);
 pinMode(4, OUTPUT);
 pinMode(5, OUTPUT);
 pinMode(6, OUTPUT);
 pinMode(7, OUTPUT);
 pinMode(8, OUTPUT);
 pinMode(9, OUTPUT);
```

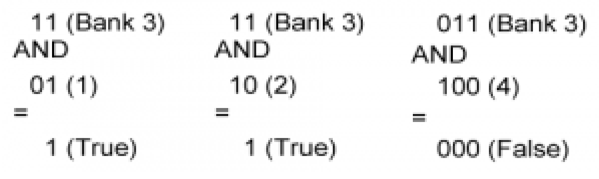
We set the D0-D7 pins as outputs as we will need to write to these pins soon.

?
```
unsigned int x = 0x2100; // Address where we write to
digitalWrite(latchPin, LOW);
shiftOut(dataPin, clockPin, MSBFIRST, (x >> 8));
shiftOut(dataPin, clockPin, MSBFIRST, (x & 0xFF));
digitalWrite(latchPin, HIGH);
delay(5);
```

We set the address to 2100h as per the switch banks URL and do the usual shifting out to the shift registers.

?
```
// Select the bank
if (bank & 1) {
  digitalWrite(2, HIGH);
}
if (bank & 2) {
  digitalWrite(3, HIGH);
}
if (bank & 4) {
  digitalWrite(4, HIGH);
}
if (bank & 8 ) {
  digitalWrite(5, HIGH);
}
if (bank & 16) {
  digitalWrite(6, HIGH);
}
if (bank & 32) {
  digitalWrite(7, HIGH);
}
if (bank & 64) {
  digitalWrite(8, HIGH);
}
if (bank & 128) {
  digitalWrite(9, HIGH);
}
delay(5);
```

Now we select the bank we want to use. Using the AND operator with the bank input number and a multiple of 2 we are actually able to check which individual bits are enabled in the bank variable. Remember how we did hardware bits converted to software bits? We're now doing the reverse of that.



```
  11 (Bank 3)      11 (Bank 3)      011 (Bank 3)
AND              AND              AND
  01 (1)           10 (2)           100 (4)
=                =                =
   1 (True)         1 (True)        000 (False)
```

For example, if bank was equal to 3 the first (bank & 1) would pass as 11 AND 1 gives 1 and if statement executes. Next (bank & 2) would go through as 11 AND 10 gives back 1. For (bank & 4), this wouldn't execute as 11 AND 100 gives 0.

?
```
// Set back to reading ROM
digitalWrite(rdPin, LOW); // RD 0
digitalWrite(wrPin, HIGH); // WR 1
```

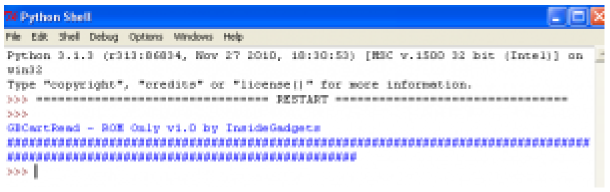
Next we set RD and WR back as they were so we can now read the loaded ROM bank.

?
```
// Reset outputs to LOW
 for (int l = 2; l <= 9; l++) {
 digitalWrite(l, LOW);
 }

 pinMode(2, INPUT);
 pinMode(3, INPUT);
 pinMode(4, INPUT);
 pinMode(5, INPUT);
 pinMode(6, INPUT);
 pinMode(7, INPUT);
 pinMode(8, INPUT);
 pinMode(9, INPUT);
 delay(5);
}
```

We now set the D0-D7 pins back to LOW and set them back as inputs so we are now ready to read the ROM bank.

```
int romBanks = 8; // Change to the number of ROM banks
```

Before you upload this to the Arduino, you need to change the romBanks variable to the amount of ROM banks you have which as you should know is indicated by reading address 0148h and referencing that with page 56 of TheNintendoGameboy.pdf, for me it's 8 banks.

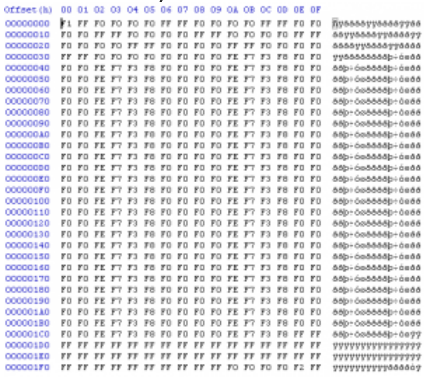


```
Python 3.1.3 (r313:86834, Nov 27 2010, 18:30:53) [MSC v.1500 32 bit (Intel)] on
win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
GBCartRead - ROM Only v1.0 by InsideGadgets
#################################################################################
#################################################
>>>
```

After uploading, go back to my script and run it. Just so you know I am in the process of making improvements to my script so it can adjust the ROM banks automatically, read our the game title, etc but that's for another post. You should now see more hashes appear than last time and it will take a bit longer to complete too.

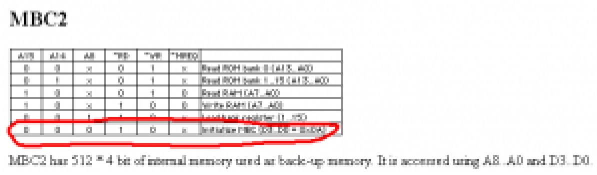
And that's it we're done. I hope you've learned how we used the Arduino and some documentation to read the Gameboy Cartridge's ROM and how the ROM banks work. In my next part, I'll show you how we can read the RAM so we can read those game saves and hopefully even load our game saves to the cartridge if it ever gets erased. The GBCartRead project page can be found here which will contain the latest version of my Python script and Arduino code.
Edit (21/03/11): GBCartRead has been updated to v1.1 which can automatically detect the number of ROM banks so you don't need to change anything on the Arduino side. It will also print the Game Title, MBC type, ROM size and RAM size in the Python script.

I hope you've gotten a feel of how we can access the Gameboy Cartridge in Part 1 and now it's time for the real part that I was looking forward to: Reading the RAM of the cartridge. Before we start I recommend you read Part 1: Reading the ROM.



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  F1 FF F0 F0 F0 F0 FF FF F0 F0 F0 F0 FF FF F0 F0  ñÿððððÿÿððððÿÿðð
00000010  F0 F0 FF FF F0 F0 F0 F0 FF FF F0 F0 F0 F0 FF FF  ððÿÿððððÿÿððððÿÿ
00000020  F0 F0 F0 F0 FF FF F0 F0 F0 F0 FF FF F0 F0 F0 F0  ððððÿÿððððÿÿðððð
00000030  FF FF F0 F0 F0 F0 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ÿÿððððððððþ÷óøðð
00000040  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000050  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000060  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000070  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000080  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000090  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
000000A0  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
000000B0  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
000000C0  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
000000D0  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
000000E0  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
000000F0  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000100  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000110  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000120  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000130  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000140  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000150  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000160  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000170  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000180  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000190  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
000001A0  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
000001B0  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
000001C0  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 FF FF  ððþ÷óøððððþ÷óøÿÿ
000001D0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000001E0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000001F0  FF FF FF FF FF FF FF FF FF FF F0 F0 F0 F0 F2 FF  ÿÿÿÿÿÿÿÿÿÿððððòÿ
```

In this tutorial I'll still be using my F1RACE game that uses MBC2 which is actually simpler compared to others because some of the other MBC's use RAM banking; which is just like ROM banking as you should already know if you read Part 1. I'll also cover another MBC to show how RAM banking works.



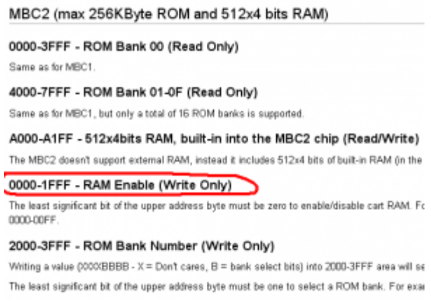
MBC2

A15	A14	A8	*RD	*WR	*MREQ	
0	0	x	0	1	x	Read ROM bank 0 (A13..A0)
0	1	x	0	1	x	Read ROM bank 1..15 (A13..A0)
1	0	x	0	1	0	Read RAM (A7..A0)
1	0	x	1	0	0	Write RAM (A7..A0)
0	0	1	1	0	x	Load bank register (1..15)
0	0	0	1	0	x	Initialise MBC (D3..D0 = 0x0A)

MBC2 has 512 * 4 bit of internal memory used as back-up memory. It is accessed using A8..A0 and D3..D0.

When we want to access the SRAM for reading or writing we firstly have to enable the SRAM, this is done by setting RD/WR and sending a specific command to the MBC. The command is called "initialise

MBC" which is found on the VBG website and is given to us as 0x0A, this translates to data pins D1 & D3 should be on (00001010). The other thing to take note of is that RD needs to be set to 1 (off) and WR to be set to 0 (on) when we give the 0x0A command.



MBC2 (max 256KByte ROM and 512x4 bits RAM)

0000-3FFF - ROM Bank 00 (Read Only)

Same as for MBC1.

4000-7FFF - ROM Bank 01-0F (Read Only)

Same as for MBC1, but only a total of 16 ROM banks is supported.

A000-A1FF - 512x4bits RAM, built-in into the MBC2 chip (Read/Write)

The MBC2 doesn't support external RAM, instead it includes 512x4 bits of built-in RAM (in the

0000-1FFF - RAM Enable (Write Only)

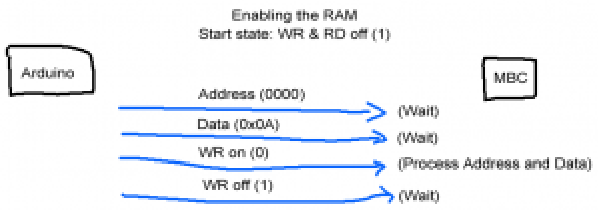
The least significant bit of the upper address byte must be zero to enable/disable cart RAM. Fo
0000-00FF.

2000-3FFF - ROM Bank Number (Write Only)

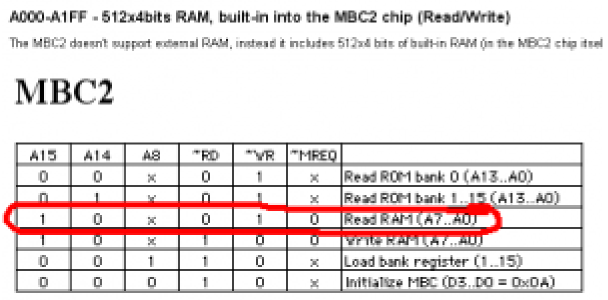
Writing a value (XXXXBBBB - X = Don't cares, B = bank select bits) into 2000-3FFF area will se

The least significant bit of the upper address byte must be one to select a ROM bank. For exa

But before we give the "initialise MBC" command we need to know where to write this command to. We hop over to GbdevWiki website and find the address range to enable the RAM: "RAM Enable (Write only)" for MBC2 is 0000-1FFF but they recommend 0000-00FF, so we'll just use 0000. If you don't initialise the MBC correctly and then try to read the RAM you will either get the same value when reading all the RAM or get random values both of which I have experienced.



What I have found is that we can actually leave RD and WR as 1 (off), specify the address (0000) and data (0x0A) then turn WR as (0) on and then turn it back off (1). Basically what is happening is we set up the address and data but the MBC doesn't do anything with it until we turn WR on (0) and that's when it processes the address and data which makes sense now. This is something to keep in mind because before in Part 1 I wasn't exactly doing it this way (I was turning on WR/RD before we set up the address/data which worked fine for reading the ROM).



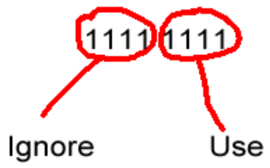
A000-A1FF - 512x4bits RAM, built-in into the MBC2 chip (Read/Write)

The MBC2 doesn't support external RAM, instead it includes 512x4 bits of built-in RAM (in the MBC2 chip itsel

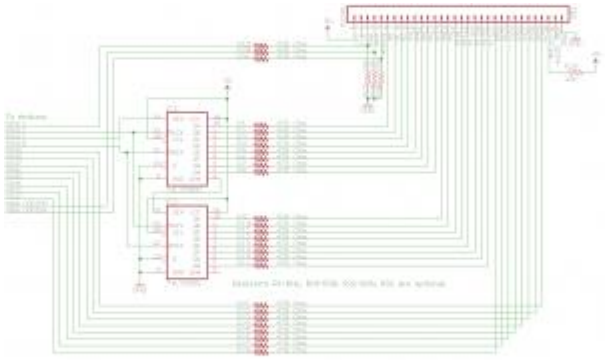
MBC2

A15	A14	A8	*RD	*WR	*MREQ	
0	0	x	0	1	x	Read ROM bank 0 (A13..A0)
0	1	x	0	1	x	Read ROM bank 1..15 (A13..A0)
1	0	x	0	1	0	Read RAM (A7..A0)
1	0	x	1	0	0	Write RAM (A7..A0)
0	0	1	1	0	x	Load bank register (1..15)
0	0	0	1	0	x	Initialize MBC (D3..D0 = 0x0A)

The next step is to find out which address the RAM is at and the values for RD and WR, it looks like it’s at A000-A1FF which would give us 512 values and RD is 0 and WR is 1.



They mention that there are only 4 bits to each value. This is different to how we are used to; normally values would have 8 bits which makes that address value have a byte. So in fact, 512x4bits will really only give us 256bytes of real data. They mention that the 4 bits that data is stored on is the lower 4 bits of the data pins which is D0-D3, so we can ignore the upper 4 bits.



Also we now need to use the CS_RAM or MREQ pin when reading the RAM, it’s the pin after the RD. Our schematic has been updated above to reflex this new pin, same wire up like the WR and RD pins are. When we read the RAM the MREQ pin needs to be 0 (on).

The last step is to disable the RAM, if this isn’t done then you may lose your save data if you disconnect the cartridge from the power. Looking at the addresses website, we disable the RAM by writing 0x00 to the same address we enabled the RAM at: 0000. After this we are free to remove the cartridge or turn the power off.

Read MBC2 RAM

Now for the code.

?
```
int latchPin = 10;
int dataPin = 11;
int clockPin = 12;
int rdPin = A5;
int wrPin = 13;
int mreqPin = A4;

void setup() {
  Serial.begin(57600);
  pinMode(latchPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  pinMode(dataPin, OUTPUT);
  pinMode(rdPin, OUTPUT);
  pinMode(wrPin, OUTPUT);
  pinMode(mreqPin, OUTPUT);
  for (int l = 2; l <= 9; l++) {
    pinMode(l, INPUT);
  }
}
```

Just the assignment of pins and our standard setup like last time. Note that our new MREQ pin is assigned to Analog Pin 4 on the Arduino. a new we just have a for loop this time for the input pins.

?

```
void loop() {

  // Wait for serial input
  while (Serial.available() <= 0) {
    delay(200);
  }
  Serial.flush();
```

On to the loop, I have added a wait until the Arduino receives some serial input before it starts so this lets us change cartridges without needing to power off or reset the Arduino. The serial input will come from our Python script.

?

```
  // MBC2 Fix (unknown why this fixes it, maybe has to read ROM before RAM?)
  digitalWrite(wrPin, HIGH); // WR off
  digitalWrite(latchPin, LOW);
  shiftOut(dataPin, clockPin, MSBFIRST, (0x0134 >> 8));
  shiftOut(dataPin, clockPin, MSBFIRST, (0x0134 & 0xFF));
  digitalWrite(latchPin, HIGH);
```

Now this next part is something that I spent a few days trying to figure out, after a lot of testing I came up with the above code. For some unknown reason if you don't perform the above and you have a MBC2 cartridge then it won't read the first few values correctly. Maybe MBC requires you to read the ROM before the RAM, who knows. A MBC5 cartridge works fine without it.

?

```
  Serial.println("START");
  // Turn everything off
  digitalWrite(rdPin, HIGH); // RD off
  digitalWrite(wrPin, HIGH); // WR off
  digitalWrite(mreqPin, HIGH); // MREQ off
```

We have START command so our Python script knows when to start saving data as the RAM. As per our new findings, we turn everything off before we start doing anything.

?

```
  unsigned int addr = 0; // Now we really need to use unsigned
```

Remember how in the last part I said we would be better off using an unsigned int even though a "normal" int would work? Now we really need an unsigned int because RAM reading starts at 0xA000 which is 40,960 in decimal and if we used the "normal" int it would overflow to -24,576.

?

```
  // Shift out 0x0000
  digitalWrite(latchPin, LOW);
  shiftOut(dataPin, clockPin, MSBFIRST, (addr >> 8));
  shiftOut(dataPin, clockPin, MSBFIRST, (addr & 0xFF));
  digitalWrite(latchPin, HIGH);
  delay(1);

  // Set D0-D7 pins as outputs
  for (int l = 2; l <= 9; l++) {
    pinMode(l, OUTPUT);
  }
```

Using our shift registers we shift out 0x0000 and set out data pins to outputs.

?

```
  // Initialise MBC: 0x0A
  digitalWrite(3, HIGH);
  digitalWrite(5, HIGH);
  digitalWrite(wrPin, LOW); // WR on
  digitalWrite(wrPin, HIGH); // WR off
```

Now we initialise the MBC so we can read the RAM, by turning on pins 3 and 5 on the Arduino we are really turning on D1 and D3 on the gameboy cartridge and that gives us 0x0A (00001010). WR is set to low (on) so the MBC processes our request and we set it back to high (off)

?

```
  // Turn outputs off and change back to inputs
  for (int l = 2; l <= 9; l++) {
    digitalWrite(l, LOW);
    pinMode(l, INPUT);
  }
```

We just turn off the outputs and set the data pins back to inputs.

?
```
  // Read RAM (512 addresses)
  for (addr = 0xA000; addr <= 0xA1FF; addr++) {
    digitalWrite(latchPin, LOW);
    shiftOut(dataPin, clockPin, MSBFIRST, (addr >> 8));
    shiftOut(dataPin, clockPin, MSBFIRST, (addr & 0xFF));
    digitalWrite(latchPin, HIGH);
    delayMicroseconds(50);
```

This part should look similar to how we read the ROM, we have changed the for loop to start at 0xA000 and end at 0xA1FF for the 512 values in MBC2. Another way to say it would be 512 nibbles as the MBC2 RAM only is stored on 4 bits – a nibble is half a byte.

?
```
    // Tell MBC to process our RAM request
    digitalWrite(mreqPin, LOW); //MREQ on
    digitalWrite(rdPin, LOW); // RD on
```

By setting MREQ and RD pins to low (on) we are telling the MBC to process our RAM request.

?
```
    byte bval = 0xF0; // Upper 4 bits always high (11110000)
    for (int z = 5; z >= 2; z--) { // Read only 4 lower bits
      if (digitalRead(z) == HIGH) {
        bitWrite(bval, (z-2), HIGH);
      }
    }
```

Now we can read the data pins. We remember that there are 4 bits (not 8) to each of those values and that the data is in the lower 4 bits, so we need to change our reading code. Firstly set the upper 4 bits to high by assigning bval as 0xF0 which would give us 1111 0000 meaning that the lower 4 bits are unchanged. Next we only read data pins D0-D3 by changing our start value to 5 in the for loop.

?
```
    // Done reading this part of RAM
    digitalWrite(mreqPin, HIGH); //MREQ off
    digitalWrite(rdPin, HIGH); // RD off

    Serial.println(bval, DEC);
  }
  Serial.println("END");
```

After we have read the data pins, we set MREQ and RD back to high (off) and the rest is the same as before, we print the value as a decimal and print END to tell our Python script to stop.

?
```
  // Disable RAM
  addr = 0;
  digitalWrite(latchPin, LOW);
  shiftOut(dataPin, clockPin, MSBFIRST, (addr >> 8));
  shiftOut(dataPin, clockPin, MSBFIRST, (addr & 0xFF));
  digitalWrite(latchPin, HIGH);
  delay(1);

  // Set D0-D7 pins as outputs
  for (int l = 2; l <= 9; l++) {
    pinMode(l, OUTPUT);
    digitalWrite(l, LOW);
  }
```

It's time to disable the RAM, as before we will be writing to the address 0x0000 and we our data pins to outputs and set them all to 0 as we will be writing 0x00.

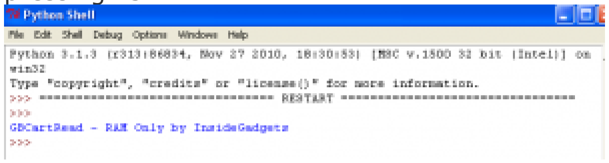
?

```
  // Tell MBC to process our request
  digitalWrite(wrPin, LOW); // WR 0
  digitalWrite(wrPin, HIGH); // WR 1

  // Set pins back to inputs
  for (int l = 2; l <= 9; l++) {
    pinMode(l, INPUT);
  }
}
```

We now set WR to low (on) and then high (off) for the MBC to process our request and then set the data pins back to inputs.

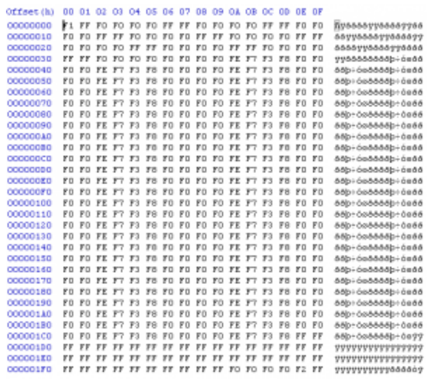
Test the F1RACE save game

Firstly download iG_GBCartRead_Read_RAM_Only and then upload the iG_GBCartRead_Read_RAM_Only_Arduino.pde file to the Arduino and run the iG_GBCartRead_Read_RAM_Only_Script.py Python script either by double clicking it or editing and pressing F5.



```
Python 3.1.3 (r313:86834, Nov 27 2010, 18:30:53) [MSC v.1500 32 bit (Intel)] on
win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
GBCartRead - RAM Only by InsideGadgets
>>>
```

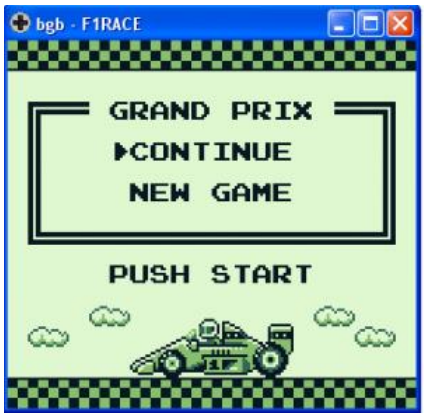
It should only take a few seconds and we'll have a file called mbc2save.sav created.



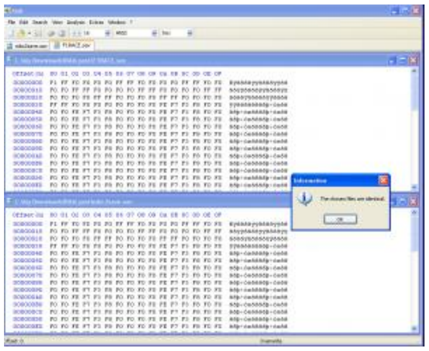
```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  F1 FF F0 F0 F0 F0 FF FF F0 F0 F0 F0 FF FF F0 F0  ñÿððððÿÿððððÿÿðð
00000010  F0 F0 FF FF F0 F0 F0 F0 FF FF F0 F0 F0 F0 FF FF  ððÿÿððððÿÿððððÿÿ
00000020  F0 F0 F0 F0 FF FF F0 F0 F0 F0 FF FF F0 F0 F0 F0  ððððÿÿððððÿÿðððð
00000030  FF FF F0 F0 F0 F0 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ÿÿððððððððþ÷óøðð
00000040  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000050  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000060  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000070  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000080  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000090  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
000000A0  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
000000B0  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
000000C0  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
000000D0  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
000000E0  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
000000F0  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000100  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000110  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000120  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000130  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000140  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000150  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000160  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000170  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000180  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
00000190  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
000001A0  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
000001B0  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 F0 F0  ððþ÷óøððððþ÷óøðð
000001C0  F0 F0 FE F7 F3 F8 F0 F0 F0 F0 FE F7 F3 F8 FF FF  ððþ÷óøððððþ÷óøÿÿ
000001D0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000001E0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000001F0  FF FF FF FF FF FF FF FF FF FF F0 F0 F0 F0 F2 FF  ÿÿÿÿÿÿÿÿÿÿððððòÿ
```

We'll open that up with our hex editor, looks like there is data in there.

Now we can actually test that we have the complete F1RACE save game by loading our F1RACE ROM into the emulator, you did dump your ROM right? :). Copy mbc2save.sav to F1RACE.sav and then load up BGB (a gameboy emulator) and BGB should find the save file.



Yes, it works!

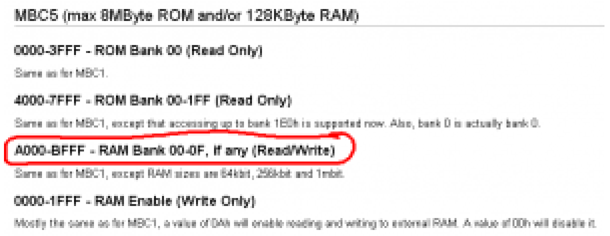


The chosen files are identical.

Another way to check is to delete F1RACE.sav, then load F1RACE in BGB again except this time it will make it's own save game. Now we can compare our mb2save.sav with it. Note that this might not work all the time because some emulators don't make the exact same file as the Gameboy but you will see some similarities. For F1RACE it does and they are exactly the same.

Read MBC5 RAM using RAM banking

Now we'll modify our code to use RAM banking on a MBC5 cartridge which has 32Kbytes – my cartridge of choice is Pokemon Red because it contains data outside the first 8Kbytes when you start a new game so you can easily verify your save with the emulators save.



MBC5 (max 8MByte ROM and/or 128KByte RAM)

0000-3FFF - ROM Bank 00 (Read Only)

Same as for MBC1.

4000-7FFF - ROM Bank 00-1FF (Read Only)

Same as for MBC1, except that accessing up to bank 1E0h is supported now. Also, bank 0 is actually bank 0.

A000-BFFF - RAM Bank 00-0F, if any (Read/Write)

Same as for MBC1, except RAM sizes are 64kbit, 256kbit and 1mbit.

0000-1FFF - RAM Enable (Write Only)

Mostly the same as for MBC1, a value of 0Ah will enable reading and writing to external RAM. A value of 00h will disable it.

The RAM is grouped in 8K blocks with 4 banks. The GbdevWiki website says that the start address for MBC5 is still A000 but the end address changes to BFFF to fit the 8Kbytes (BFFF – A000 = 8,191 + 1 for starting at A000). Once we have read the first 8Kbytes block of RAM just fine but we need to be able to switch to the next 8K and so on. We do this by writing to the address 0x4000 and using the data pins to specify the bank to read which should sound familiar to you.

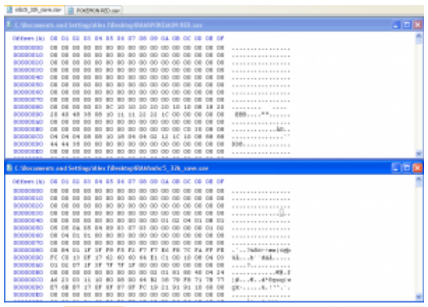
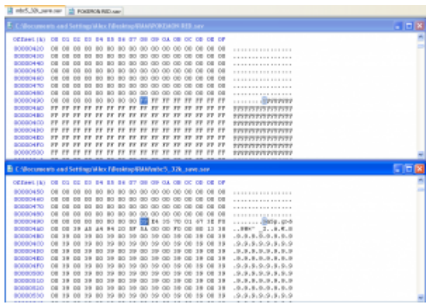
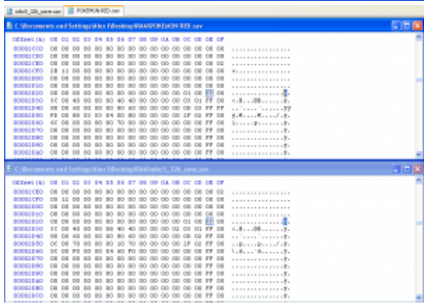
?

```
...

  // Initialise MBC: 0x0A
  digitalWrite(3, HIGH);
  digitalWrite(5, HIGH);
  digitalWrite(wrPin, LOW); // WR on
  digitalWrite(wrPin, HIGH); // WR off

  // Turn outputs off and change back to inputs
  for (int l = 2; l <= 9; l++) {
    digitalWrite(l, LOW);
    pinMode(l, INPUT);
  }

  // Switch banks (32Kbytes)
for (int bank = 0; bank <= 3; bank++) {

// Shift out 0x4000
digitalWrite(latchPin, LOW);
shiftOut(dataPin, clockPin, MSBFIRST, (0x4000 >> 8));
shiftOut(dataPin, clockPin, MSBFIRST, (0x4000 & 0xFF));
digitalWrite(latchPin, HIGH);
delayMicroseconds(50);
```

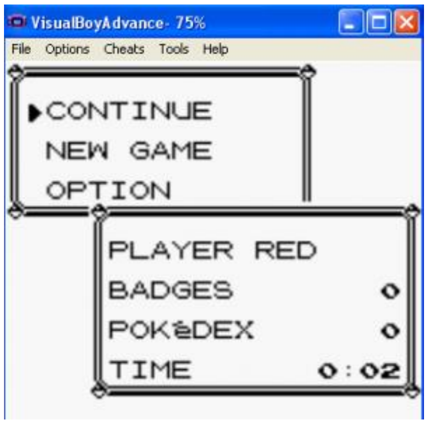
```
// Set D0-D7 pins as outputs
for (int l = 2; l <= 9; l++) {
pinMode(l, OUTPUT);
}

if (bank & 1) {
digitalWrite(2, HIGH);
}
if (bank & 2) {
digitalWrite(3, HIGH);
}

// Tell MBC to process our RAM bank request
digitalWrite(wrPin, LOW); // WR 0
digitalWrite(wrPin, HIGH); // WR 1

// Turn outputs off and change back to inputs
for (int l = 2; l <= 9; l++) {
digitalWrite(l, LOW);
pinMode(l, INPUT);
}

    // Read RAM
    for (addr = 0xA000; addr <= 0xBFFF; addr++) {
      digitalWrite(latchPin, LOW);
      shiftOut(dataPin, clockPin, MSBFIRST, (addr >> 8));
      shiftOut(dataPin, clockPin, MSBFIRST, (addr & 0xFF));
      digitalWrite(latchPin, HIGH);
      delayMicroseconds(50);

...
```

There isn't too much of a code change as all that's needed is a for loop to happen 4 times. We shift out to the 0x4000 address, set our data pins as outputs and by using the AND (&) we can determine which data pins to turn on. We enable (0) and then disable (1) the WR pin so the MBC can process our request then set our data pins back to inputs. The only other thing that changes is to set our end address in the reading RAM loop is set to BFFF and that's it.

Download iG_GBCartRead_Read_32K_RAM_Only1 and then upload the iG_GBCartRead_Read_32K_RAM_Only_Arduino.pde file to the Arduino and run the iG_GBCartRead_Read_32K_RAM_Only_Script.py Python script. After a few seconds and we'll have a file called mbc5_32k_save.sav created.

Now let's compare this file with the VBA emulator version which is at the same state as the Gameboy (I chose the VBA emulator because BGB appears to write random data when it could be just padded with FF to make it easier to distinguish real data from null data). It might not all match but there are specific sections that do so we know we have a good save.



Last test load it up in the emulator and yes it also works too!

And that's it, now we are able to read the RAM of our cartridges. In my next part I'll show you how we can write to the RAM so then we can restore our game saves, it's pretty simple actually and we can use the same code for reading just swap values of RD and WR then set the data pins to outputs but that's for next time.

So what good is reading the RAM if we can't write back to it? This is what we'll cover in Part 3: Write to RAM.

Writing to the RAM is quite similar to reading the RAM except instead of reading the data pins you write to them. This part won't be as large as our previous parts because we're really just re-using ourReading the RAM code. Let's jump right to the code.

?

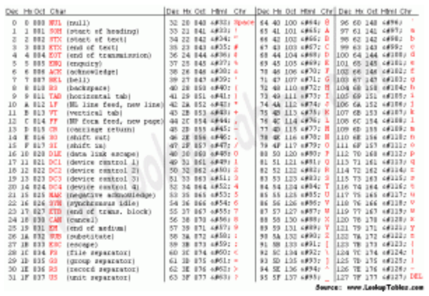
```
...

  // Initialise MBC: 0x0A
 digitalWrite(3, HIGH);
 digitalWrite(5, HIGH);
 digitalWrite(wrPin, LOW); // WR on
 digitalWrite(wrPin, HIGH); // WR off

 // Write RAM (512 addresses)
 for (addr = 0xA000; addr <= 0xA1FF; addr++) {
   digitalWrite(latchPin, LOW);
   shiftOut(dataPin, clockPin, MSBFIRST, (addr >> 8));
   shiftOut(dataPin, clockPin, MSBFIRST, (addr & 0xFF));
   digitalWrite(latchPin, HIGH);
   delayMicroseconds(50);
```

All code up to the end of the shifting out the address is the same as in Part 2.

?

```
// Tell MBC to process our RAM request
digitalWrite(mreqPin, LOW); // MREQ on
digitalWrite(wrPin, LOW); // WR on

// Wait for serial input
while (Serial.available() <= 0) {
  delay(1);
}

// Decode input
byte bval = 0;
if (Serial.available() > 0) {
  char c = Serial.read();
  bval = (int) c;
}
```

We put MREQ to 0 as we've done in reading the RAM but this time we put the WR pin as 0 (on) to tell the MBC we will be writing to the RAM. Next we wait until something is received from our Python script. The Python script will read our save file byte by byte and send 1 byte at a time over to the Arduino. Next if there is 1 byte of serial data available (because we just use the if statement once) we store it into our character variable and convert that character into it's integer value.



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

(ASCII table used from http://www.asciitable.com)

What I haven't mentioned before is the ASCII table, this table shows the number that represents each character. If we sent over the character "A", the integer value of that would be 65.

?
```
// Read the bits in the received character and turn on the
// corresponding D0-D7 pins
for (int z = 9; z >= 2; z--) {
  if (bitRead(bval, z-2) == HIGH) {
    digitalWrite(z, HIGH);
  }
  else {
    digitalWrite(z, LOW);
  }
}
Serial.println("."); // Send something back to update progress
```

Now we take that integer value 65 (01000001) and read which bits are on and which ones are off by using bitRead; previously we used the AND (&) function but bitRead is simpler. For example, bitRead(65, 7) would correspond to the left most bit (most significant) and would read as 0. Next we turn on the corresponding data pin if that bit was read as HIGH (1) otherwise we set that data pin to LOW (0). Lastly we just print something back to our Python program so we can get a sense of the progress.

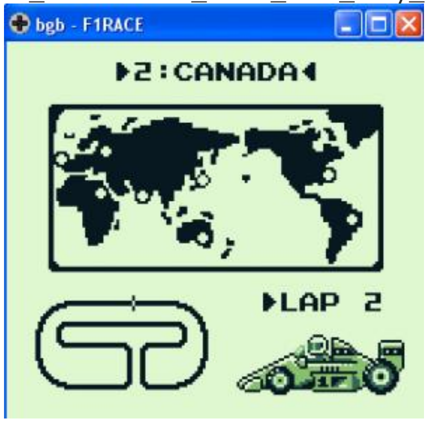
?
```
  // Done writing this part of RAM
   digitalWrite(mreqPin, HIGH); // MREQ off
   digitalWrite(wrPin, HIGH); // WR off

 }
 Serial.println("END");

 // Disable RAM
 addr = 0;
 digitalWrite(latchPin, LOW);
 shiftOut(dataPin, clockPin, MSBFIRST, (addr >> 8));
 shiftOut(dataPin, clockPin, MSBFIRST, (addr & 0xFF));
 digitalWrite(latchPin, HIGH);
 delay(1);
```

. . .

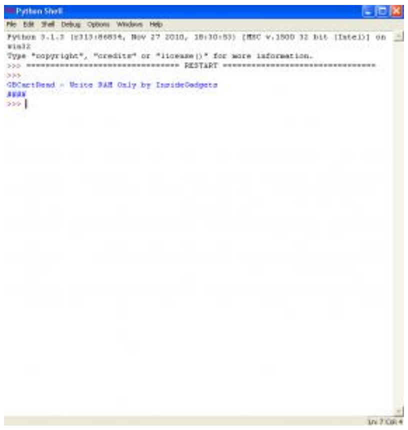
The last thing to do is set MREQ back off (1) and set WR off too (1) and then it's the same code as Reading the RAM, just to disable the RAM.

Restore the F1RACE save game

Firstly download iG_GBCartRead_Write_RAM_Only and then upload the iG_GBCartRead_Write_RAM_Only_Arduino.pde file to the Arduino.

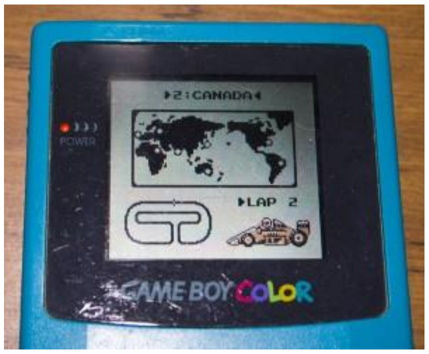


Make sure you have a F1RACE.sav (or change this). I've just loaded my save in the BGB emulator to show I'm up to the track "2. Canada".



```
Python 3.1.3 (r313:86834, Nov 27 2010, 18:30:53) [MSC v.1500 32 bit (Intel)] on
win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
GBCartRead - Write RAM Only by InsideGadgets
####
>>> 
```

Now run the iG_GBCartRead_Write_RAM_Only_Script.py Python script either by double clicking it or editing and pressing F5.The Python script will open F1RACE.sav and send this save byte by byte to the Arduino. After a few seconds it will show some hashes.



Remove the cartridge, insert into the Gameboy and give it a test, it works!

So that's it, we now can do everything to the Gameboy Cartridge that we need, read the ROM, read the RAM and write to the RAM 🙂