Introduction to Game Boy Hacking

Pepijn de Vos

July 2017

Contents

1 Introduction **2**

1.1 About me 2

1.2 The plan 2

2 Game Boy hardware **4**

3 Resources **5**

4 Installing the tools **5**

5 Assembly Basics **6**

6 A wild glitch appeared **7**

6.1 Old Man Glitch 7

6.2 What is your name? 9

6.3 Who is that Pokémon?! 10

6.4 Your own Adventure 11

7 The Debugger **11**

8 Hello world! **15**

9 Conclusion **18**

1 Introduction

In this workshop I will teach you how to get started reading and writing code for the Nintendo Game Boy. First we will go over the hardware and the assembly language, and from there start to inspect some code and make changes to it.

1.1 About me

I'm basically a software developer turned electrical engineer, doing a lot of crazy projects along the way.

My journey with Game Boy hacking started when I connected the Game Link port to an Arduino to spoof Pokémon trades. This ended up on Hackaday, where someone suggested to trade across the internet. I quickly dismissed the idea as impossible due to latency. Then I dug around in annotated Pokémon dissassembly and did it anyway.

After poking at the Game Boy from the comfort of my Arduino, I decided to get my hands dirty and implement Pokémon Go on the Game Boy. First using Pokémon Red with an Arduino and GPS module. Then in Pokémon Crystal by directly controlling an SPI sensor with the Game Boy.

Having gotten comfortable with Game Boy assembly, I wrote a small paint program from scratch, to draw graphics for a game I have yet to make. Probably I'll keep on Yak-shaving like that.

Then when I needed to do a project for a digital hardware course at the University of Twente, we implemented the graphics chip of the Game Boy in VHDL, driving a VGA monitor. I also wrote a simple presentation framework to run our project presentation on our own emulator.

1.2 The plan

The above projects may sound extremely hard to you, and some of them were. But the truth is that when I started on Bill's Arduino, I knew nearly nothing about game development or assembly programming.

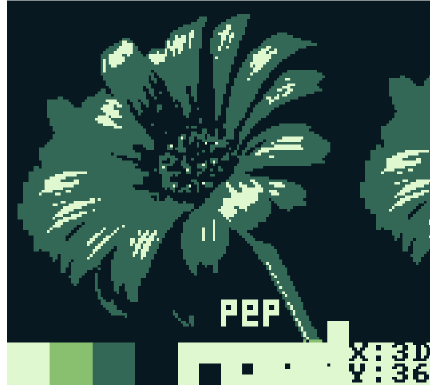

Figure 1: Game Boy Paint

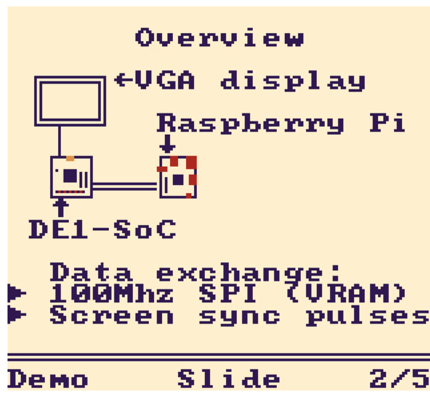

Figure 2: Project presentation

I simply started reading documentation and source code, making small changes, learning as I went. So what I'll attempt to do is to recreate a turbo-charged version of my learning process. What we'll roughly cover:

1. Basic understanding of the platform
2. Useful resources and documentation
3. Browsing disassembled and annotated Game Boy ROMs
4. Getting familiar with the debugger
5. Writing your first assembly code

Using this knowledge, we will

1. Inspect famous glitches in Pokémon Red
2. Discover a new glitch
3. Make silly changes to the game
4. Write your own "Hello World!" program
5. Mess around until tired and sleepy

2 Game Boy hardware

I assume you all know what a Game Boy is. Below are some factoids I copied from Pandoc. Of course every single metric listed has funny particularities and edge-cases, but we'll get to those as needed.

CPU	8-bit (Similar to the Z80 processor)
Clock Speed	4.194304MHz (4.295454MHz for SGB, max. 8.4MHz for CGB)
Work RAM	8K Byte (32K Byte for CGB)
Video RAM	8K Byte (16K Byte for CGB)
Screen Size	2.6"
Resolution	160x144 (20x18 tiles)
Max sprites	Max 40 per screen, 10 per line
Sprite sizes	8x8 or 8x16
Palettes	1x4 BG, 2x3 OBJ (for CGB: 8x4 BG, 8x3 OBJ)
Colors	4 grayshades (32768 colors for CGB)
Horiz Sync	9198 KHz (9420 KHz for SGB)
Vert Sync	59.73 Hz (61.17 Hz for SGB)
Sound	4 channels with stereo sound
Power	DC6V 0.7W (DC3V 0.7W for GB Pocket, DC3V 0.6W for CGB)

3 Resources

The nice thing about the Game Boy platform is that there is just so much documentation about everything. I will give you links and descriptions of some particularly helpful ones.

If you have an hour to spare, definitely watch The Ultimate Game Boy Talk by Michael Steil. He does an amazing job at describing the history of the Game Boy, as well as going into great detail about the hardware.

The document that you will use most comes in several variations and names such as Pandoc, gbspec, and Game Boy CPU Manual. It is the bible of Game Boy technology. It contains everything ranging from detailed hardware descriptions, to IO register maps, to assembly instructions.

Since all Game Boy games were originally written in assembly language, some lovely people on the internet decided to disassemble and annotate Pokémon Red for your reading pleasure. This provides a great way to become familiar with the inner workings of the game and the platform it runs on, as well as an easy avenue into hacking the game. It is also a good reference of working code and collection of development tools. Some of these folks also gather on IRC in #pret to answer my stupid questions.

When you want to write games from scratch, Pokered is not ideal because it has so much going on. A better starting place is the GameBoy Assembly Language Primer. It contains useful boilerplate code, hardware defines, utility routines, and a sprite font.

4 Installing the tools

To get started, we need 4 things:

- The Pokered source code
- RGBDS, a compiler for Game Boy assembly
- BGB, a Game Boy emulator with a good debugger
- A good text editor

First we will need to clone Pokered using git. Doing so will later allow us to easily search the source code using git grep.

Please refer to their installation instructions on how to clone the project, install RGBDS, and build the ROMs.

For running the ROMs, in theory any emulator would work, but BGB has the best debugger, which we'll be using extensively. Unfortunately it is written for Windows, but it runs excellent on Wine. Please download and install BGB from their website.

5 Assembly Basics

Assembly is the most simple and verbose programming language imaginable. It directly translates "mnemonics" into CPU instructions. A mnemonic is basically a short abbreviation for a CPU instruction. The Game Boy has very few CPU instructions, which make programming it even more simple and more verbose.

Open up the Game Boy CPU manual, and skim trough section 3, Game Boy command overview. Pay special attention to 8-bit loads and jumps.

The "ld" mnemonic is the most common in code, and copies bytes between registers and memory locations. It is especially common to load data in the **A**ccumulator register, as a lot of instructions only work on that register. 16-bit addresses will frequently be loaded in the HL register for similar reasons. (HL has a special load-and-increment instruction, allowing compact pointer loops) Run git grep ld in your Pokered folder to see how it is used.

Note especially the different addressing modes. You can do direct loads, where you pass it a register or a literal number. You can also do indirect loads, where you wrap a number or register in brackets to use it as the address to load from/to. Some modes also support adding a number and a register, but not all combinations are possible.

The way to do loops, if statements, and functions is by using jumps and labels. A jump instruction takes an address in the code, and jumps to that position. Some jumps check a condition before jumping. There is JP "jump" and JR "jump relative". They work the same, but JR is slightly more compact and faster.

As their arguments they optionally take a condition such as Z "zero", NZ "non-zro", C "carry" and NC "no carry". These flags are set by instructions such as CP "compare", and most logic and arithmetic instructions.

Instead of an absolute or relative address, usually a label is passed as the argument. The compiler will substitute this for the actual address, saving you some headaches. A label is a word at the start of a line that starts with a dot or ends

with a colon (double colon for global labels).

Run git grep jr, open one of the files, and see if you can understand what is going on. Pay attention to the way local dot labels are used for loops and branches, and global double colons are used for function declarations.

Before I'll let you figure out the rest by yourself using your CPU manual, I would like to briefly point out three more instructions. CALL and RET "return" are jump instructions that are used for functions. CALL saves the address it jumped from on the stack, and RET jumps back to the saved location. And finally, you'll frequently see xor a, which does an exclusive or on the "a" register with itself, essentially setting it to zero.

6 A wild glitch appeared

Now that you know a bit what Game Boy assembly generally looks like, let's try to understand some interesting things. If you've played Pokémon a lot, you probably know there are all these glitches that you can use to get rare Pokémon or do other weird things. Have you ever wondered how they actually work? Time to find out.

6.1 Old Man Glitch

One of the more famous glitches is of course the Old Man Glitch. Go ahead and read up on the cause of this bug. It is a reuse of memory combined with an oversight in the map data. Let's see how it works in the code.

Step one is to find where the player name is stored. I just took a guess and grepped for terms such as "player" ad "name". After a while I've learned that all WRAM variables are declared in a file called "wram.asm".

Searching around in that file you'll find "wPlayerName", which seems to be where your own name is stored. However, grepping for that does not immediately result in anything related to the old man. Knowing the man is supposed to be located in Viridian city, I look for files related to that map, leading me to "scripts/viridiancity.asm", which contains a comment "set up battle for Old Man", loading "BATTLE_TYPE_OLD_MAN" into "wBattleType".

We quickly find it is defined as 1, this is however a dead end, but prompts me to look for "wPlayerName" in battle-related files. Going back to grepping, we find "engine/battle/core.asm", which is a huge file containing all the core battle logic. However, searching for "wPlayerName", we quickly strike gold a good 2000 lines into the file.

```
.nonstandardbattle
        ld a, [wBattleType]
        cp BATTLE_TYPE_SAFARI
        ld a, BATTLE_MENU_TEMPLATE
        jr nz, .menuselected
        ld a, SAFARI_BATTLE_MENU_TEMPLATE
.menuselected
        ld [wTextBoxID], a
        call DisplayTextBoxID
        ld a, [wBattleType]
        dec a
        jp nz, .handleBattleMenuInput ; handle menu input if it's not the old ma
; the following happens for the old man tutorial
        ld hl, wPlayerName
        ld de, wGrassRate
        ld bc, NAME_LENGTH
        call CopyData  ; temporarily save the player name in unused space,
                       ; which is supposed to get overwritten when entering a
                       ; map with wild Pokemon. Due to an oversight, the data
                       ; may not get overwritten (cinnabar) and the infamous
                       ; Missingno. glitch can show up.
        ld hl, .oldManName
        ld de, wPlayerName
        ld bc, NAME_LENGTH
        call CopyData
```

What we see here is that "wBattleType" is loaded and compared to various types. The reason we don't see "BATTLE_TYPE_OLD_MAN" mentioned here is that the test is done using "dec a", which decrements the battle type and sets the zero flag. Why a decrement is chosen over a compare is any ones guess, but it wont be the last time you'll see obscure code to save a single byte or clock cycle.

If the battle type was 1, the zero flag is set and the relative jump is ignored. The code that follows loads "wPlayerName" and "wGrassRate" into registers and calls "CopyData", which does exactly what it says on the tin. (go ahead and find its implementation if you want. Hint: look for "CopyData::" to find the declaration) The comment helpfully mentiones that this enables the Old Man Glitch. Then another call to "CopyData" copies ".oldManName" to "wPlayerName".

We can check back in "wram.asm" to see that the variable is directly followed by "wGrassMons". Grepping further to see what happens with "wGrassMons" in Cinnabar Island is left for you to explore.

Figure 3: First, what is your name?

6.2 What is your name?

Now that we understand how this glitch works, can we find other ways to catch different Pokémon by surfing around on Cinnabar? While what I will describe below is currently written on Bulbapedia, that is the case because I put it there after I discovered this new glitch while working on TCPoke.

It is really not that hard, we just need to find places that write to "wGrassMons" or an address just before it. Since I was neck-deep in the link trading code, "wLinkEnemyTrainerName" seemed very promising. It does not take very long to find that indeed "engine/cable_club.asm" writes to it.

```
.findStartOfEnemyNameLoop
        ld a, [hli]
        and a
        jr z, .findStartOfEnemyNameLoop
        cp SERIAL_PREAMBLE_BYTE
        jr z, .findStartOfEnemyNameLoop
        cp SERIAL_NO_DATA_BYTE
        jr z, .findStartOfEnemyNameLoop
        dec hl
        ld de, wLinkEnemyTrainerName
        ld c, NAME_LENGTH
.copyEnemyNameLoop
        ld a, [hli]
        cp SERIAL_NO_DATA_BYTE
        jr z, .copyEnemyNameLoop
        ld [de], a
        inc de
        dec c
```

```
jr nz, .copyEnemyNameLoop
```

What we see here is a loop in the code that parses the data that is received over the link cable. After receiving the seeds for the random number generator, it chomps off a few preamble bytes, and then does a loop to copy "NAME_LENGTH" bytes to "wLinkEnemyTrainerName".

A few things to note here are the use of HLI, which is the special load-and-increment instruction I mentioned earlier. While the **DE**stination register is incremented manually. It also does a check for "SERIAL_NO_DATA_BYTE", in which case it just skips to the next byte. "and a" is another of those common patterns, that is essentially just setting the "Z" flag if "a" is zero.

Using this information, it is possible to catch arbitrary Pokémon based on the name of your link cable opponent, which means you can use many different names in the same save. To fully execute the glitch, it is needed to exit the Cable Club without resetting the Game Boy. The can be done using the Cable Club escape glitch, but I'll leave it to you to see how that works.

6.3 Who is that Pokémon?!

As we saw earlier, "wGrassMons" determines which Pokémon can be found in a certain area. What if we changed the map data so that we can find every Pokémon we want? Let's do it!

I just browsed around the repository a bit, until I found a folder called "data/wild-Pokemon". Inside it I found neatly organized files for every route with data that exactly fits the "wGrassMons" memory. All that is needed is to change all the "PIDGEY" and "RATATA" in "route1.asm" into any Pokémon defined in "constants/pokemon_constants.asm".

Grepping back from there quickly leads to "engine/overworld/wild_mons.asm" which copies the correct data to "wGrassRate" and "wGrassMons". Further grepping would reveal where this function is used, and is again left for you to explore.

Anyway, after changing a few routes like this, running make should produce a new ROM with new Pokémon, as per your changes. Congratulations! You wrote your first Game Boy hack! What are you going to change next?

6.4 Your own Adventure

Now that I've shown you how you can grep your way through the source code and make small changes, it's up to you what to do next. Bulbapedia has a whole List of glitches in Generation I for you to explore and expand upon. Or you can do whatever else you want. Make the shops sell Master Balls, change the starter Pokémon, add a new item, change some sprites, change what people say, make Magikarp learn self-destruct, anything is possible.

Using the tools that were developed for Pokered, you could in theory even disassemble other games and hack those. Except it'd be a lot harder because you would have to do it without all the names and annotations that people added to Pokered. You'd just have to use the debugger to find what you're looking for and read the code. Which brings me to...

7 The Debugger

So far we have mostly looked at assembly code where other people have given meaningful names to all the labels. Now lets see if we can make sense of another popular game that has not been disassembled. I'll be exploring Super Mario Land, but if you're feeling adventurous, pick any game you like.

After having loaded the game and doing some unavoidable "play testing", open up the debugger via the right click menu under "Other" as shown in figure 4.

Since you don't know what anything is, the only thing you can do is start at a known point, or look for known register locations. The start of the ROM is probably not a good choice, as it just contains boring setup code.

For my TCPoke project, I looked a lot at all the code that touches FF01 and FF02, the serial registers. If you want to figure out how the bleeps and bloops work, look for code that touches the audio registers, if you want to see how sprites are drawn, look at code that touches the tile data or DMA registers. If you want to see how the input is handled, look at FF00, the joypad register. If you want to see the main game loop, the VBlank interrupt is a good place to start. For games that do interesting graphics effects, such as the perspective in F1 Race, HBlank is also interesting to look at. (spoiler: it scrolls the window every line to create curves in the road)

When you open the debugger, it puts you at the start of the VBlank interrupt that happens between every frame. You can use F7 (trace) to step through the code and see the registers update on the right side.

Note that the default syntax here is slightly different than the one in Pokered.

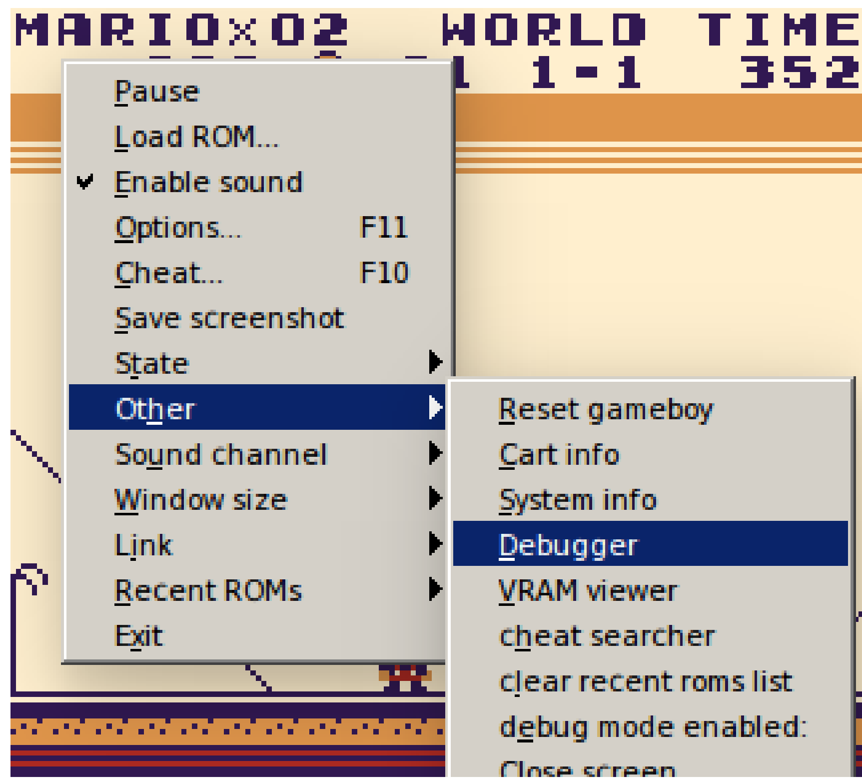

Figure 4: Open the debugger

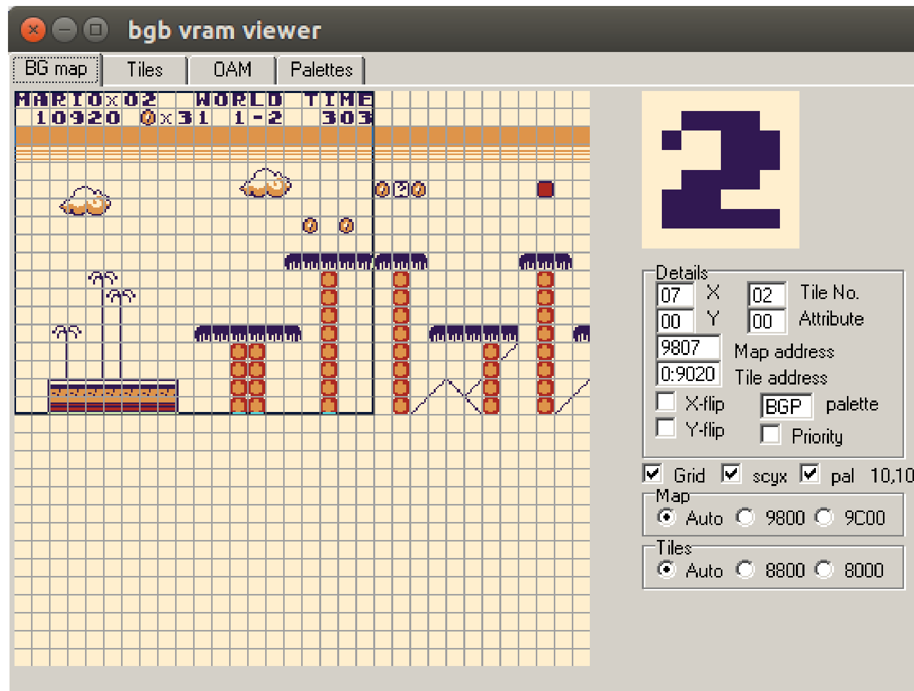

Figure 5: VRAM viewer

Where BGB uses "ldi a, (hl)", RGBDS would use "ldi a, [hl]" or "ld a, [hli]". If you want you can change the syntax in the BGB settings from "no$gmb" to "rgbds".

I thought it'd be funny to find where the game stores the lives you have and cheat a bit. Rather than browsing the endless sea of assembly (which I did for a good part of the day, believe me), lets open up the VRAM viewer (Figure 5), which can also be found in the "Other" right-click menu.

Play around a bit with the VRAM viewer, it gives you a good idea how the game is drawn. Note for example how the BG map wraps around as you walk, except the top bar. Look at the LCD status interrupt on LYC if you're curious.

Now hover your mouse over the tile that holds your lives. It'll tell you where the tile data is stored and what the map address is (9807). If we're unlucky this data is written as a larger chunk, maybe starting at 9800, but it seems we're lucky.

Hit CRTL+F and search for 9807. It returns only one location that loads the A register into the tile of interest. Hit F2 to set a breakpoint and F9 to run the game. Every time you die the breakpoint gets triggered.

At the breakpoint, you'll see that the A register contains the amount of lives you have. If you head back over to the VRAM viewer tiles tab, you'll see that tiles 0-9 have tile numbers 0-9, making the math easier. (read Pandoc on the two tile sets, and the way they are indexed. **They overlap!**)

Now we just need to backtrack where the value in A came from. A dozen lines up there is an unconditional RET that must indicate the end of another subroutine, and a few lines below is another unconditional RET that marks the end of the current routine. If you look at the stack pointer in the lower right, or trace (F7) past the RET, you can find this routine is called from the VBlank interrupt. I put another interrupt at the beginning of the routine, but ended up placing it after the early returns. (when this routine is called and the lives do not need to be drawn, it seems to return early)

After tracing trough the code a few times it became evident that DA15 contains the number of lives as binary coded decimal (4 bits per decimal digit). You can see the use of DAA to fix up the BCD number after binary addition and the use of SWAP to extract individual digits.

```
    ldh  a,[$FF9F]
    and  a
    ret  nz          ; early return if FF9F != 0
    ld   a,[$C0A3]
    or   a
    ret  z           ; early return if C0A3 == 0
    cp   a,$FF
    ld   a,[$DA15]   ; load DA15
    jr   z,.decr     ; go to .decr if C0A3 == FF
    cp   a,$99       ; DA15 is binary coded decimal!
    jr   z,.skip     ; skip everything if DA15 == 99
    push af          ; push DA15 on stack
    ld   a,$08       ; update some unknown registers
    ld   [$DFE0],a
    ldh  [$FFD3],a
    pop  af          ; restore DA15
    add  a,$01       ; add 1 to DA15
.draw
    daa              ; decimal adjust
    ld   [$DA15],a   ; store new value in DA15
    ld   a,[$DA15]   ; useless?
    ld   b,a         ; copy DA15 to b
    and  a,$0F       ; only get last BCD digit
    ld   [$9807],a   ; write last digit to tile map
    ld   a,b         ; load DA15 from b to a
    and  a,$F0       ; get first BCD digit
    swap a           ; swap upper and lower bits
```

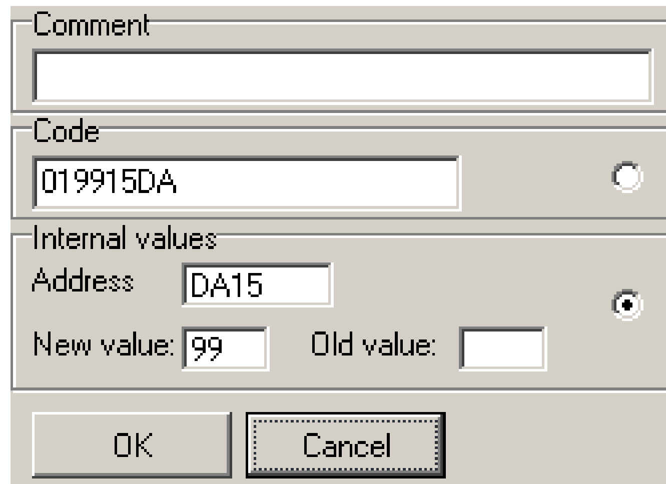

Figure 6: New cheat

```
    ld    [$9806],a ; write first digit to tile map
.skip
    xor   a
    ld    [$C0A3],a ; set C0A3 to zero
    ret             ; return
.dead
    ld    a,$39     ; update some unkown registers
    ldh   [$FFB3],a
    ld    [$C0A4],a
    jr    .skip     ; skip to the end
.decr
    and   a         ; test if DA15 is 0
    jr    z,.dead   ; you're dead, else
    sub   a,$01     ; decrement your lives
    jr    .draw     ; draw the updated values
```

Knowing this, I opened the cheat window from the window menu in the debug window. I created a new cheat that always read the value 99 from DA15, as pictured in Figure 6. The result can be seen in Figure 7.

8 Hello world!

This is the part where you start your own game and do whatever you like, and I'll run around trying to fix everything. Pairing up is highly recommended.

Figure 7: 98 lives in Super Mario Land

Alternatively, if this is all way too much, I'll take requests for improvised live-coding, while you sit back and watch me make horrible mistakes.

To start making your own game, I recommend taking the GALP code as a start, to fill in the basic details like the ROM header and setting up the screen and tile data.

Unfortunately GALP does not compile with RGBDS out of the box. To get you started, I took my paint program (that was based on GALP), and stripped it back down to a "Hello World!" game. It also includes a Makefile based on Pokered, so if RGBDS is installed correctly, you should be able to just run make to produce a main.gbc file.

The first thing you could do is obviously make it print something else. The default tile data it loads also includes some other characters, arrows and boxes. The Makefile is also capable of compiling png images to 2bpp files that can be directly included in the game as follows.

```
MyImage:
INCBIN "image.2bpp"
```

This can then be loaded somewhere in the tile data like so:

```
ld      hl, MyImage     ; the image address
ld      de, _TILE0      ; the first tile set
ld      bc,16*4         ; length (16 bytes per tile)
call    mem_Copy        ; Copy tile data to memory
```

Next, you might want to add some interactivity. Maybe you could add a simple animation or do something with the joypad.

If you look up the documentation on the OAM (Object Attribute Memory), you'll see it's not that hard to assign a sprite to it and set the X and Y coordinates. Once you figured that out, it should not be too hard to change VBlank to increment the Y position for example. You can also take a look at how my presentation framework does it.

If you look at the CPU Manual section on the joypad register FF00, you'll see it contains example code for reading the joypad. A RGBDS-compatible version of that snippet is already in the code I gave you. You can have a look at how my paint program does it. The actual tests for button presses are lines like bit PADB_START, b which set the zero flag depending on the button state, usually followed by a conditional jump.

Using these two basic features, you could already make a sprite move around when you press buttons.

From here it's up to your creativity. You could make the Star Wars opening

scene, or a whole interactive story/text-based RPG. You could make a little beatbox that just makes bleeps when you press buttons, and extend it with a basic sequencer later, or you could make a little party game where the players have to press increasingly complex button sequences such as ↑↑↓↓←→←→BA.

9 Conclusion

That's all I have. I hope you had fun and learned something.

In the unlikely event that I had time to cover everything, we did a lot. We started with a crash course 8080 assembly, followed by reading overwhelming amounts of said assembly. Then we reached a major milestone where we changed some code and ran our modified Pokemon game. Then we took off the training wheels and made a new cheat code for Super Mario Land. And finally we went completely wild with writing our own game.