ROMhacking Int. 104

Compression Scheme RE

(Dynamic and Static approach)

By Bunkai

(Last update: August 1st 2024)

Index

Introduction & Tools

** Warning: the Int. means Intermediate, this is not a beginner topic. However, the intention of the document is to be a reference for beginners, and a reminder for veterans.*

** RE stands for Reverse Engineering, so expect a lot of that.*

For this document/tutorial, you will need to have the following tools:

Game's ROM: Marmalade Boy (Japan).sgb with CRC 32: 0F3FF7DA

The Hex Editor of your choice. The screenshots are from crystal tile, but ImHex and HxD work too

A console emulator. Screenshots for this document are from BGB and Mesen2.

Your code editor of choice if you want to program your tools. Tools in this doc are code in python.

A lot of patience, effort, and a place to write your notes to review them when needed.

Preface

This document attempts to be a learning reference to 'attack' compression in a game, from start to finish, with all the (minimum required) theory followed by practical examples. All of the steps and explanations are applied to an actual project with actual screenshots of the project.

References will be given when required, you can use them to dive deeper. But rest assured, you will only need the tools used in each step to follow along. You can just read this paper without any extra material if your only goal is to learn about the process.

You must know that each section of this document can be read standalone. If you want to read only one section, don't be afraid to do so, however, you will benefit from reading all.

** Note: The actual extraction/reinsertion from/to the ROM won't be detailed here. If you need a hint for that, just mind the offset to extract or reinsert bytes and copy them in or out.*

The end goal is to understand and be able to use and export the knowledge to do this process, not to parrot. Remember: Curiosity leads to knowledge, be curious

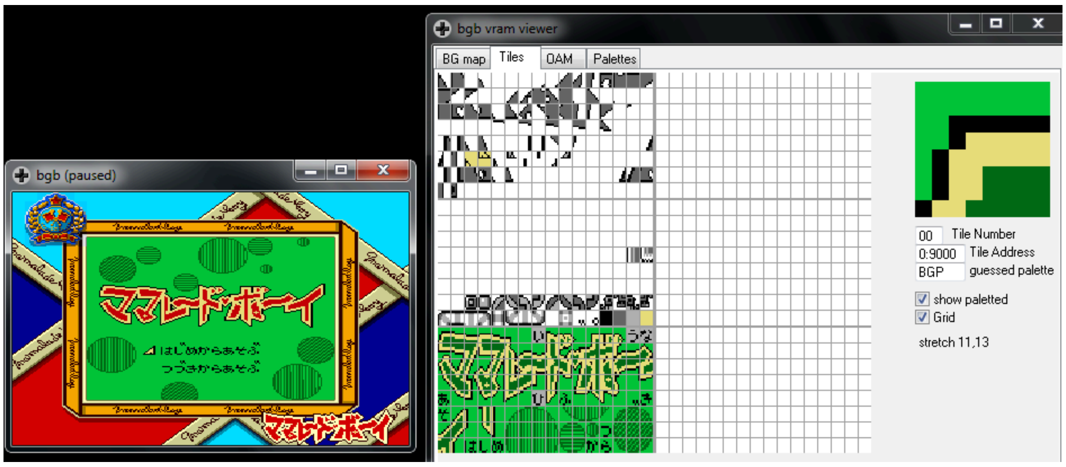
Static Analysis

I am going to follow, and copy, the great explanations from abridgewater in his: *Rom Hacking 201: A non-ASM approach to compressed data*, which you can find here: https://www.romhacking.net/documents/878/
However, I will be applying them to the “marmalade boy (japan).sgb” game in order to learn and “break” his compression routine. (I will also add some aid from the use of a, dynamic, debugger in some parts for explanation purposes)

* Investigating the Title Screen

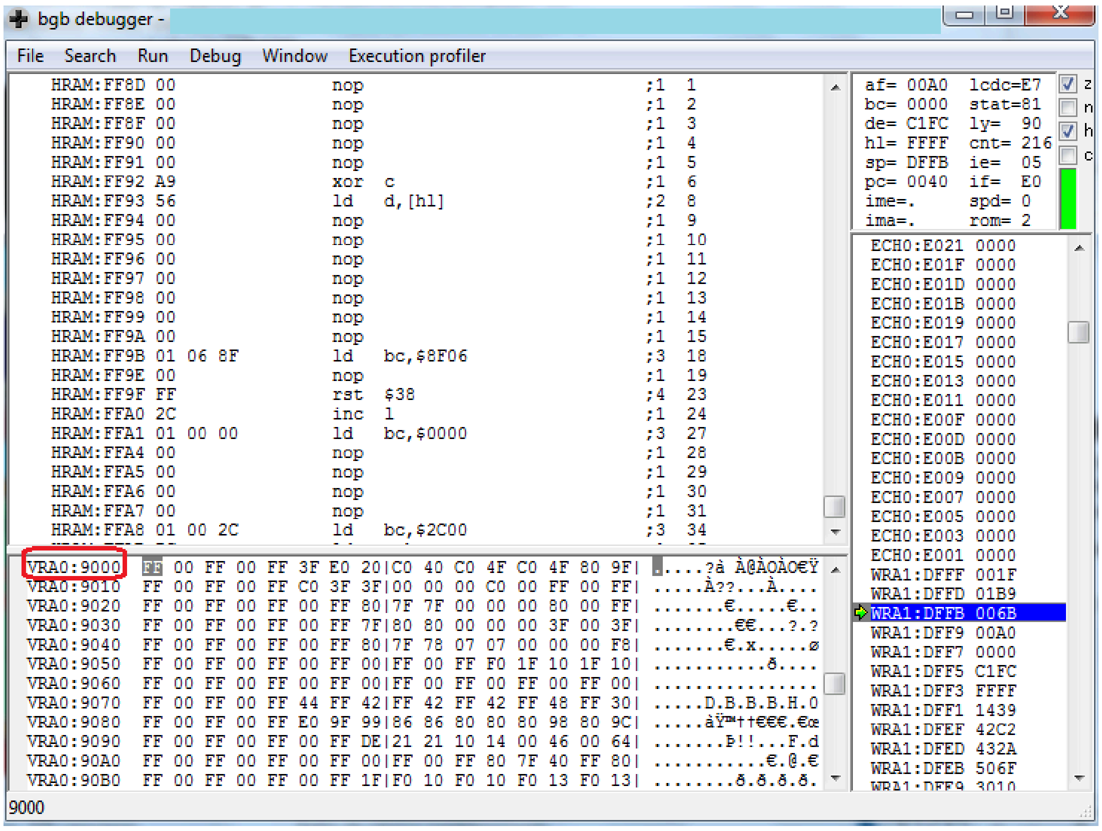
Step one, find our decompressed data. Start the ROM in an emulator that has debugging support, get to the title screen, and pause the emulation. Next, use the tile viewer to find the address of some of the decompressed tiles in VRAM.

Here you can see an example of one Tile and its address 0:9000



```
00      Tile Number
0:9000  Tile Address
BGP     guessed palette
[✓] show paletted
[✓] Grid
stretch 11,13
```

Now, point your memory viewer at that (0:9000) address of the V-RAM to see the decompressed data in arrays of bytes. See below for a reference:



```
HRAM:FF8D 00            nop                     ;1  1
HRAM:FF8E 00            nop                     ;1  2
HRAM:FF8F 00            nop                     ;1  3
HRAM:FF90 00            nop                     ;1  4
HRAM:FF91 00            nop                     ;1  5
HRAM:FF92 A9            xor  c                  ;1  6
HRAM:FF93 56            ld   d,[hl]             ;2  8
HRAM:FF94 00            nop                     ;1  9
HRAM:FF95 00            nop                     ;1  10
HRAM:FF96 00            nop                     ;1  11
HRAM:FF97 00            nop                     ;1  12
HRAM:FF98 00            nop                     ;1  13
HRAM:FF99 00            nop                     ;1  14
HRAM:FF9A 00            nop                     ;1  15
HRAM:FF9B 01 06 8F      ld   bc,$8F06           ;3  18
HRAM:FF9E 00            nop                     ;1  19
HRAM:FF9F FF            rst  $38                ;4  23
HRAM:FFA0 2C            inc  l                  ;1  24
HRAM:FFA1 01 00 00      ld   bc,$0000           ;3  27
HRAM:FFA4 00            nop                     ;1  28
HRAM:FFA5 00            nop                     ;1  29
HRAM:FFA6 00            nop                     ;1  30
HRAM:FFA7 00            nop                     ;1  31
HRAM:FFA8 01 00 2C      ld   bc,$2C00           ;3  34
```

```
VRA0:9000  FF 00 FF 00 FF 3F E0 20|C0 40 C0 4F C0 4F 80 9F| .....?à À@ÀOÀO€Ÿ
VRA0:9010  FF 00 FF 00 FF C0 3F 3F|00 00 00 C0 00 FF 00 FF| .....À??...À....
VRA0:9020  FF 00 FF 00 FF 00 FF 80|7F 7F 00 00 00 80 00 FF| .......€.....€..
VRA0:9030  FF 00 FF 00 FF 00 FF 7F|80 80 00 00 00 3F 00 3F| ........€€...?.?
VRA0:9040  FF 00 FF 00 FF 00 FF 80|7F 78 07 07 00 00 00 F8| .......€.x.....ø
VRA0:9050  FF 00 FF 00 FF 00 FF 00|FF 00 FF F0 1F 10 1F 10| ...........ð....
VRA0:9060  FF 00 FF 00 FF 00 FF 00|FF 00 FF 00 FF 00 FF 00| ................
VRA0:9070  FF 00 FF 00 FF 44 FF 42|FF 42 FF 42 FF 48 FF 30| .....D.B.B.B.H.0
VRA0:9080  FF 00 FF 00 FF E0 9F 99|86 86 80 80 80 98 80 9C| .....àŸ™††€€€.€œ
VRA0:9090  FF 00 FF 00 FF 00 FF DE|21 21 10 14 00 46 00 64| .......Þ!!...F.d
VRA0:90A0  FF 00 FF 00 FF 00 FF 00|FF 00 FF 80 7F 40 FF 80| ...........€.@.€
VRA0:90B0  FF 00 FF 00 FF 00 FF 1F|F0 10 F0 10 F0 13 F0 13| ........ð.ð.ð.ð.
```

```
af= 00A0   lcdc=E7   [✓] z
bc= 0000   stat=81   [ ] n
de= C1FC   ly=  90   [✓] h
hl= FFFF   cnt= 216  [ ] c
sp= DFFB   ie=  05
pc= 0040   if=  E0
ime=.      spd= 0
ima=.      rom= 2
```

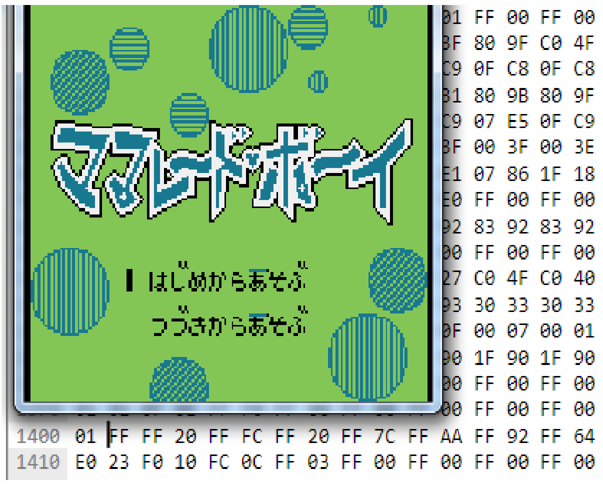
```
ECHO:E021  0000
ECHO:E01F  0000
ECHO:E01D  0000
ECHO:E01B  0000
ECHO:E019  0000
ECHO:E017  0000
ECHO:E015  0000
ECHO:E013  0000
ECHO:E011  0000
ECHO:E00F  0000
ECHO:E00D  0000
ECHO:E00B  0000
ECHO:E009  0000
ECHO:E007  0000
ECHO:E005  0000
ECHO:E003  0000
ECHO:E001  0000
WRA1:DFFF  001F
WRA1:DFFD  01B9
WRA1:DFFB  006B
WRA1:DFF9  00A0
WRA1:DFF7  0000
WRA1:DFF5  C1FC
WRA1:DFF3  FFFF
WRA1:DFF1  1439
WRA1:DFEF  42C2
WRA1:DFED  432A
WRA1:DFEB  506F
WRA1:DFE9  3010
```

```
9000
```

That very same RAM block is copied below for quick reference:

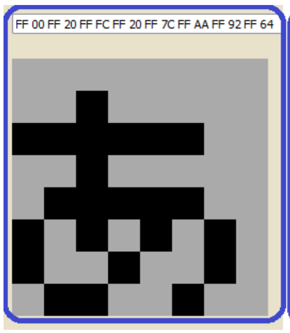
```
                    00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
--------------------------------------------------------------------------------

VRA0:9000: FF 00 FF 00 FF 3F E0 20 C0 40 C0 4F C0 4F 80 9F
VRA0:9010: FF 00 FF 00 FF C0 3F 3F 00 00 00 C0 00 FF 00 FF
VRA0:9020: FF 00 FF 00 FF 00 FF 80 7F 7F 00 00 00 80 00 FF
VRA0:9030: FF 00 FF 00 FF 00 FF 7F 80 80 00 00 00 3F 00 3F
VRA0:9040: FF 00 FF 00 FF 00 FF 80 7F 78 07 07 00 00 00 F8
VRA0:9050: FF 00 FF 00 FF 00 FF 00 FF 00 FF F0 1F 10 1F 10
VRA0:9060: FF 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF 00 FF 00
VRA0:9070: FF 00 FF 00 FF 44 FF 42 FF 42 FF 42 FF 48 FF 30
VRA0:9080: FF 00 FF 00 FF E0 9F 99 86 86 80 80 80 98 80 9C
VRA0:9090: FF 00 FF 00 FF 00 FF DE 21 21 10 14 00 46 00 64
VRA0:90A0: FF 00 FF 00 FF 00 FF 00 FF 00 FF 80 7F 40 FF 80
VRA0:90B0: FF 00 FF 00 FF 00 FF 1F F0 10 F0 10 F0
```

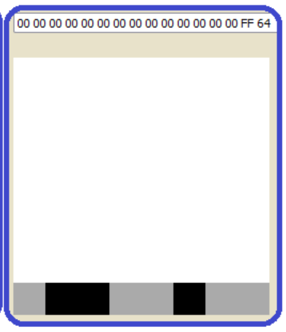
Pro Tip for this game: In mesen2's real time debugger, if you go to the VRAM $1400 address. You can do some screen edits like the one in the image below, with the あ tile edited.



```
01 FF 00 FF 00
3F 80 9F C0 4F
C9 0F C8 0F C8
31 80 9B 80 9F
C9 07 E5 0F C9
3F 00 3F 00 3E
E1 07 86 1F 18
E0 FF 00 FF 00
92 83 92 83 92
00 FF 00 FF 00
27 C0 4F C0 40
93 30 33 30 33
0F 00 07 00 01
90 1F 90 1F 90
00 FF 00 FF 00
00 FF 00 FF 00
1400 01 FF FF 20 FF FC FF 20 FF 7C FF AA FF 92 FF 64
1410 E0 23 F0 10 FC 0C FF 03 FF 00 FF 00 FF 00 FF 00
```

Explanation of this: take the あ bytes from the screenshot ($1400: FF 00 FF 20 FF FC FF 20 FF 7C FF AA FF 92 FF 64) to https://www.huderlem.com/demos/gameboy2bpp.html


FF 00 FF 20 FF FC FF 20 FF 7C FF AA FF 92 FF 64


00 00 00 00 00 00 00 00 00 00 00 00 00 00 FF 64

The changes done here are explained as follows, each 16 bytes (a full tile). If you separate that in blocks of 2:

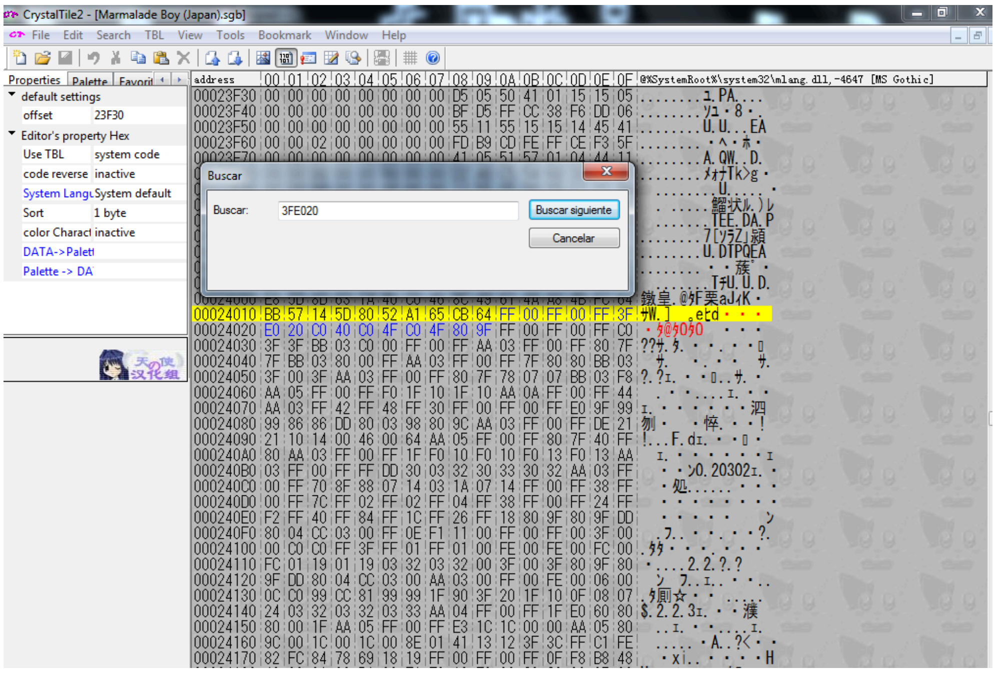
"FF 64" each of these pairs is a row of the tile being the 2nd half "64" the tile palette and the 1st half "FF" the drawn part of the raw.

This also suggests that the font could be the same used in the script.
(As proved later, this is not the case)

Back to our analisis now, this RAM block has a very regular pattern in the beginning, (it’s very common and repeated):

9000: ‘FF 00 FF 00 FF’
So, we'll skip ahead to a significant break in the pattern: At $9005, "3F E0 20"
We want this pattern to make a relative search in the ROM so we can find the related data.

Tip: You will find false positives, just remember to keep searching until you find the correct match. Searched match displayed below:


CrystalTile2 - [Marmalade Boy (Japan).sgb]

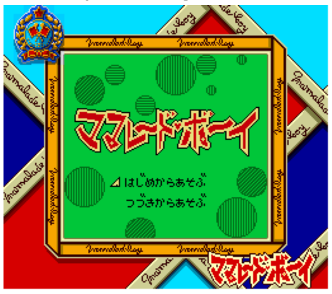
offset	23F30
Use TBL	system code
code reverse	inactive
System Langu	System default
Sort	1 byte
color Charact	inactive

Buscar: 3FE020

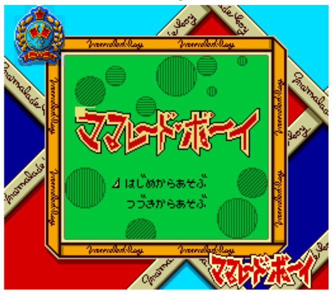
```
address  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00023F30 00 00 00 00 00 00 00 00 D5 05 50 41 01 15 15 05
00023F40 00 00 00 00 00 00 00 00 BF D5 FF CC 38 F6 DD 06
00023F50 00 00 00 00 00 00 00 00 55 11 55 15 15 14 45 41
00023F60 00 00 02 00 00 00 00 00 FD B9 CD FE FF CE F3 5F
00023F70 00 00 00 00 00 00 00 00 41 05 51 57 01 04 44 11
00024010 BB 57 14 5D 80 52 A1 65 CB 64 FF 00 FF 00 FF 3F
00024020 E0 20 C0 40 C0 4F C0 4F 80 9F FF 00 FF 00 FF C0
00024030 3F 3F BB 03 C0 00 FF 00 FF AA 03 FF 00 FF 80 7F
00024040 7F BB 03 80 00 FF AA 03 FF 00 FF 7F 80 80 BB 03
00024050 3F 00 3F AA 03 FF 00 FF 80 7F 78 07 07 BB 03 F8
00024060 AA 05 FF 00 FF F0 1F 10 1F 10 AA 0A FF 00 FF 44
00024070 AA 03 FF 42 FF 48 FF 30 FF 00 FF 00 FF E0 9F 99
00024080 99 86 86 DD 80 03 98 80 9C AA 03 FF 00 FF DE 21
00024090 21 10 14 00 46 00 64 AA 05 FF 00 FF 80 7F 40 FF
000240A0 80 AA 03 FF 00 FF 1F F0 10 F0 10 F0 13 F0 13 AA
000240B0 03 FF 00 FF FF DD 30 03 32 30 33 30 32 AA 03 FF
000240C0 00 FF 70 8F 88 07 14 03 1A 07 14 FF 00 FF 38 FF
000240D0 00 FF 7C FF 02 FF 02 FF 04 FF 38 FF 00 FF 24 FF
000240E0 F2 FF 40 FF 84 FF 1C FF 26 FF 18 80 9F 80 9F DD
000240F0 80 04 CC 03 00 FF 0E F1 11 00 FF 00 FF 00 3F 00
00024100 00 C0 C0 FF 3F FF 01 FF 01 00 FE 00 FE 00 FC 00
00024110 FC 01 19 01 19 03 32 03 32 00 3F 00 3F 80 9F 80
00024120 9F DD 80 04 CC 03 00 AA 03 00 FF 00 FE 00 06 00
00024130 0C C0 99 CC 81 99 99 1F 90 3F 20 1F 10 0F 08 07
00024140 24 03 32 03 32 03 33 AA 04 FF 00 FF 1F E0 60 80
00024150 80 00 1F AA 05 FF 00 FF E3 1C 1C 00 00 AA 05 80
00024160 9C 00 1C 00 1C 00 8E 01 41 13 12 3F 3C FF C1 FE
00024170 82 FC 84 78 69 18 19 FF 00 FF 00 FF 0F F8 B8 48
```

Jackpot! We note that the pattern from VRAM $9005, "0e f1 3f c0" appears here at $2401A as "FF 00 FF 00 FF 3F E0 20 C0 40 C0 4F C0 4F 80 9F", and the remaining of the line matches too, further confirming that this is the data that we're looking for. (If the full block is shown, everything is in plain, non compressed. If it shows tidbits with other hexes in-between then you have probably found it but it is compressed or have some extra unrelated data for the game’s code to work)

Besides, you can do a corruption test of that address (this means editing those hexes, and saving the file to run it in the emulator and see if the image displayed changes). Since the display changes, it proves to be the right location too.



Pantalla original



Pantalla corrupta

We'll now grab a bit of this ROM data for deeper analysis. Copied below for quick reference:

```
        00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
-----------------------------------------------------------------------
24010:                                 FF 00 FF 00 FF 3F
24020: E0 20 C0 40 C0 4F C0 4F 80 9F FF 00 FF 00 FF C0
24030: 3F 3F BB 03 C0 00 FF 00 FF AA 03 FF 00 FF 80 7F
24040: 7F BB 03 80 00 FF AA 03 FF 00 FF 7F 80 80 BB 03
24050: 3F 00 3F AA 03 FF 00 FF 80 7F 78 07 07 BB 03 F8
24060: AA 05 FF 00 FF F0 1F 10 1F 10 AA 0A FF 00 FF 44
24070: AA 03 FF 42 FF 48 FF 30 FF 00 FF 00 FF E0 9F 99
24080: 99 86 86 DD 80 03 98 80 9C AA 03 FF 00 FF DE 21
24090: 21 10 14 00 46 00 64 AA 05 FF 00 FF 80 7F 40 FF
240A0: 80 AA 03 FF 00 FF 1F F0 10 F0 10 F0 13 F0 13 AA
240B0: 03 FF 00 FF FF DD 30 03 32 30 33 30 32 AA 03 FF
```

At this point we have a chunk of decompressed data and a couple of places where the decompressed data matches up with data in the ROM, meaning that we've found approximately where the compressed data sits. We don't yet know any details of the compression scheme other than that it includes literal data at times, and we don't know where the compressed data starts or where it ends.

To find either end of the compressed data, we want to look forward or backwards through the ROM until we find the compressed data that corresponds with the start or the end of the decompressed data. This, in turn, requires that we work out at least the basics of the overall compression scheme.

And the compressed data includes literal data, so what we do is to line it up with the decompressed data and look for patterns. What's in the compressed data that's not in the decompressed data, and what's in the decompressed data that's not in the compressed data? Our starting point is the first match we found, and we work our way forward from there.

```
VRAM $9000:  FF 00 FF 00 FF 3F E0 20 C0 40 C0 4F C0 4F 80 9F
ROM  $2401A: FF 00 FF 00 FF 3F E0 20 C0 40 C0 4F C0 4F 80 9F
```
> These two are equal so we can't take any extra info from the comparison

```
VRAM $9010:  FF 00 FF 00 FF C0 3F 3F 00 00 00 C0 00 FF 00 FF
ROM  $2402A: FF 00 FF 00 FF C0 3F 3F BB 03 C0 00 FF 00 FF AA
```
> From these two, we can see a pattern where 'BB 03' means take the first Byte and repeat three times '00 00 00'. no control code at the end, just the next byte to display

```
VRAM $9020:  FF 00 FF 00 FF 00 FF 80 7F 7F 00 00 00 80 00 FF
ROM  $2403A: 03 FF 00 FF 80 7F 7F BB 03 80 00
```

> We can see a similar pattern again here (taking the 'AA' from the previous row), we can see 'AA 03 FF 00' results in 'FF 00 FF 00 FF 00' where the 03 again marks how many times we repeat the bytes, and we could guess that the AA means it's a two Bytes string this time 'FF 00'

```
VRAM $9030:  FF 00 FF 00 FF 00 FF 7F 80 80 00 00 00 3F 00 3F
ROM  $2403B: FF AA 03 FF 00 FF 7F 80 80 BB 03 3F 00 3F
```
> we can see the 'AA 03 FF 00' and the 'BB 03' patterns again here.

```
VRAM $9040:  FF 00 FF 00 FF 00 FF 80 7F 78 07 07 00 00 00 F8
ROM  $24049: AA 03 FF 00 FF 80 7F 78 07 07 BB 03 F8
```
> we can see the 'AA 03 FF 00' and the 'BB 03' patterns again here.

```
VRAM $9050: FF 00 FF 00 FF 00 FF 00 FF 00 FF F0 1F 10 1F 10
ROM $2404D: AA 05 FF 00 FF F0 1F 10 1F 10
```
> we can see the 'AA 03 FF 00' but now with more bytes repeated, which change it to 'AA 05 FF 00' further confirming our previous guesses

We can now code this pattern into a routine to test. Using the rom hex block as the input file, should give the vram hex block as the output

From here on, the process is to keep referencing the pattern while also looking for how does the decompressor know when to stop decompressing? There are two ways for the decompressor to know when to stop. Either it knows how long the decompressed data will be and stops when it fills that space, or there is a special end marker that it looks for in the compressed data.

The end marker is the more likely of the two, and the only space in the encoding scheme that it can fit is as a back reference with a particular value. This is an easy enough thing to determine: Walk forward through the compressed data until you reach the end. If it just stops without any unusual backreference (An item in a regular expression equivalent to the text matched by an earlier pattern in the expression), it's length-limited, and the length needs to be found. If there /is/ an unusual backreference, then that's the end marker.

[To be brutally honest, I didn’t find this one through static, I got “lucky” because I had the routine disassembled from an old attempt to find the text (that I had archived until I could use it). I just didn’t notice this was for the images until recognizing the same patterns in the static analysis. However, I will copy the reference of the end of this ROM’s block below for those who want to do it]

ROM $246A0: B6 92 6D 24 DB 49 B6 92 6D 24 DB 48 B7 81 7E 25
ROM $246B0: DA 49 B6 93 6C 23 DC 47 B8 8F 70 3F C0 FF 00 **EE**
ROM $246C0: 80 80 DD C0 04 A0 E0 90 F0 88 F8 88 F8 84 FC 82

After all the chewed info, we should be able to recreate the decompression routine. To test your tool at the beginning, use both (ROM and V-RAM) blocks you’ve been analyzing in the previous steps. Extract the block from the rom and decompress running your code, check if the result matches with the vram block to see the accuracy until you have your tool ironed.

The process to create the tools will be talked about later in a future section. But, this one about the static analysis can be considered as finished.

Dynamic Analysis

I am going to follow, and copy, the great explanations from YasaSheep in her: *ROM Hacking 202 - Reverse engineering with the debugger*, which you can find here: https://www.romhacking.net/documents/877/

However, I will be applying them to the “marmalade boy (japan).sgb” game in order to learn and “break” his compression routine.

I want to add here my whole gratitude to Phonymike for all his help documenting the ASM routine of the Marmalade game. (many of those code comments are done by him)

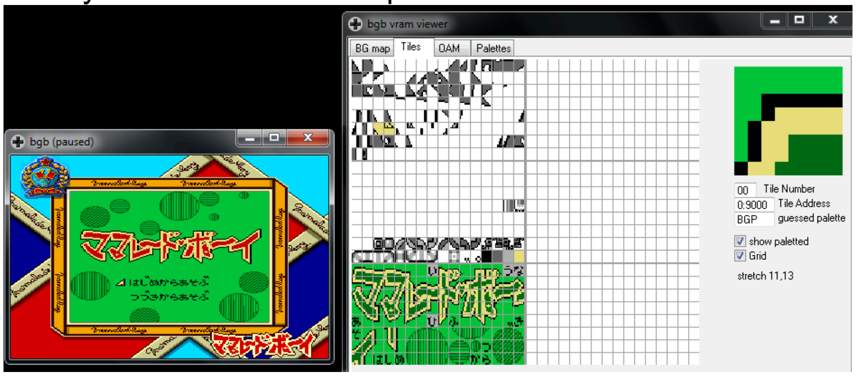
* Note: This 202 doc. uses the previous one as a base to find (have) the first addresses and a few routine logistics. However, in order to make both methods as standalone and complete as possible, I will reuse/copy the first page from the Static method (Because it’s the same beginning in both cases, the difference starts when you branch out between analyzing the data in the hex values or the code in the debugger.)

* Extra note: This method was done at first following the code to find the text, we recognized the routine as a compression form but it obviously wasn’t the text; As a result, the screenshots of this method weren’t saved, only the ASM and comments were archived for future use. It hasn’t been used again until I recognized the patterns in the static analysis (like I mentioned before). For those reasons, the process is mostly recreated and won’t throw as many tips as one would write if it was done from scratch; but, I believe it will be “good enough” to explain the method. Besides, it gives a nice way to have a method comparison with the same data, and makes the “paper” most complete.

* Investigating the Title Screen

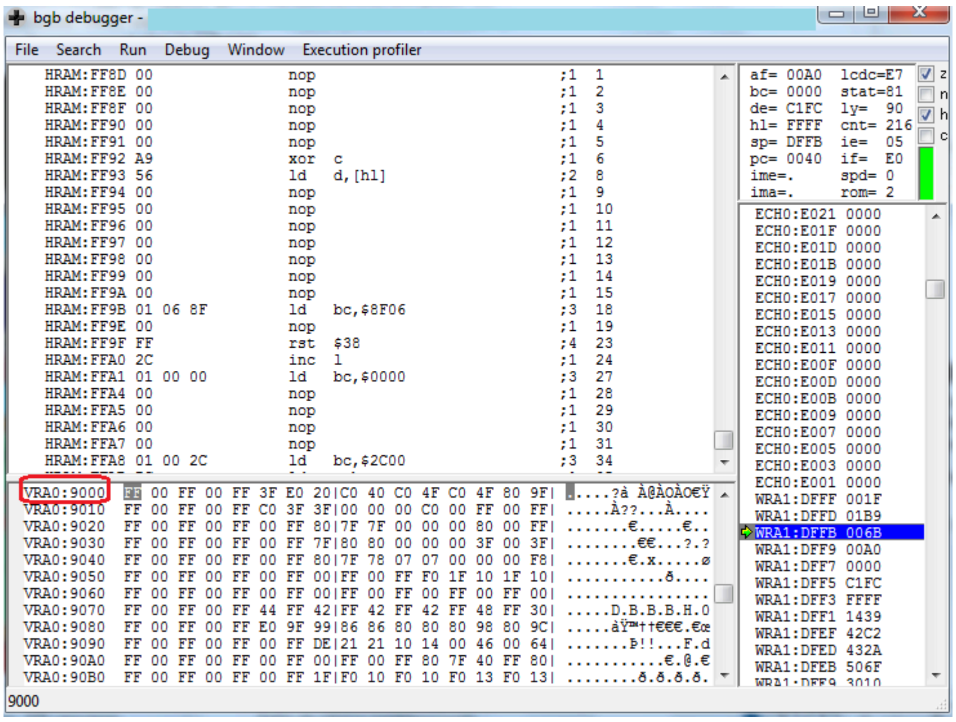
Step one, find our decompressed data. Start the ROM in an emulator that has debugging support, get to the title screen, and pause the emulation. Next, use the tile viewer to find the address of some of the decompressed tiles in VRAM.

Here you can see an example of one Tile and its address 0:9000



00 Tile Number
0:9000 Tile Address
BGP guessed palette
☑ show paletted
☑ Grid
stretch 11,13

With that, point your memory viewer at that (0:9000) address of the V-RAM to see the decompressed data in arrays of bytes. See below for a reference:



```
HRAM:FF8D 00                nop                   ;1  1
HRAM:FF8E 00                nop                   ;1  2
HRAM:FF8F 00                nop                   ;1  3
HRAM:FF90 00                nop                   ;1  4
HRAM:FF91 00                nop                   ;1  5
HRAM:FF92 A9                xor  c                ;1  6
HRAM:FF93 56                ld   d,[hl]           ;2  8
HRAM:FF94 00                nop                   ;1  9
HRAM:FF95 00                nop                   ;1  10
HRAM:FF96 00                nop                   ;1  11
HRAM:FF97 00                nop                   ;1  12
HRAM:FF98 00                nop                   ;1  13
HRAM:FF99 00                nop                   ;1  14
HRAM:FF9A 00                nop                   ;1  15
HRAM:FF9B 01 06 8F          ld   bc,$8F06         ;3  18
HRAM:FF9E 00                nop                   ;1  19
HRAM:FF9F FF                rst  $38              ;4  23
HRAM:FFA0 2C                inc  l                ;1  24
HRAM:FFA1 01 00 00          ld   bc,$0000         ;3  27
HRAM:FFA4 00                nop                   ;1  28
HRAM:FFA5 00                nop                   ;1  29
HRAM:FFA6 00                nop                   ;1  30
HRAM:FFA7 00                nop                   ;1  31
HRAM:FFA8 01 00 2C          ld   bc,$2C00         ;3  34
```

```
af= 00A0   lcdc=E7
bc= 0000   stat=81
de= C1FC   ly=  90
hl= FFFF   cnt= 216
sp= DFFB   ie=  05
pc= 0040   if=  E0
ime=.      spd= 0
ima=.      rom= 2
```

z [✓] n [ ] h [✓] c [ ]

```
ECHO:E021 0000
ECHO:E01F 0000
ECHO:E01D 0000
ECHO:E01B 0000
ECHO:E019 0000
ECHO:E017 0000
ECHO:E015 0000
ECHO:E013 0000
ECHO:E011 0000
ECHO:E00F 0000
ECHO:E00D 0000
ECHO:E00B 0000
ECHO:E009 0000
ECHO:E007 0000
ECHO:E005 0000
ECHO:E003 0000
ECHO:E001 0000
WRA1:DFFF 001F
WRA1:DFFD 01B9
WRA1:DFFB 006B
WRA1:DFF9 00A0
WRA1:DFF7 0000
WRA1:DFF5 C1FC
WRA1:DFF3 FFFF
WRA1:DFF1 1439
WRA1:DFEF 42C2
WRA1:DFED 432A
WRA1:DFEB 506F
```

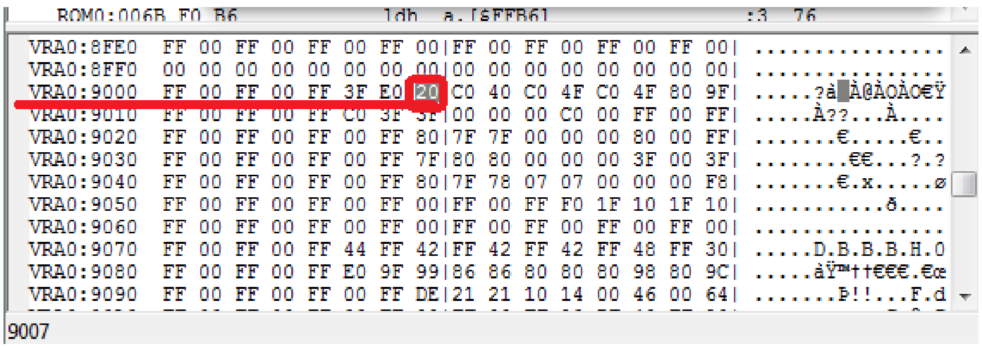
```
VRA0:9000  FF 00 FF 00 FF 3F E0 20|C0 40 C0 4F C0 4F 80 9F| .....?à À@ÀOÀO€Ÿ
VRA0:9010  FF 00 FF 00 FF C0 3F 3F|00 00 00 C0 00 FF 00 FF| .....À??...À....
VRA0:9020  FF 00 FF 00 FF 00 FF 80|7F 7F 00 00 00 80 00 FF| .......€.....€..
VRA0:9030  FF 00 FF 00 FF 00 FF 7F|80 80 00 00 00 3F 00 3F| ........€€...?.?
VRA0:9040  FF 00 FF 00 FF 00 FF 80|7F 78 07 07 00 00 00 F8| .......€.x.....ø
VRA0:9050  FF 00 FF 00 FF 00 FF 00|FF 00 FF F0 1F 10 1F 10| ...........ð....
VRA0:9060  FF 00 FF 00 FF 00 FF 00|FF 00 FF 00 FF 00 FF 00| ................
VRA0:9070  FF 00 FF 00 FF 44 FF 42|FF 42 FF 42 FF 48 FF 30| .....D.B.B.B.H.0
VRA0:9080  FF 00 FF 00 FF E0 9F 99|86 86 80 80 80 98 80 9C| .....àŸ™††€€€.€œ
VRA0:9090  FF 00 FF 00 FF 00 FF DE|21 21 10 14 00 46 00 64| .......Þ!!...F.d
VRA0:90A0  FF 00 FF 00 FF 00 FF 00|FF 00 FF 80 7F 40 FF 80| ...........€.@.€
VRA0:90B0  FF 00 FF 00 FF 00 FF 1F|F0 10 F0 10 F0 13 F0 13| ........ð.ð.ð.ð.
```

9000

You need to read those values until you find one that is unique enough so it stops when watched, during the execution running, but not so common to stop at every instruction.

I went with that ‘20’ in the VRAM Address $9007.

VRAM0: 9000   FF 00 FF 00 FF 3F E0 20 C0 40 C0 4F C0 4F 80 9F



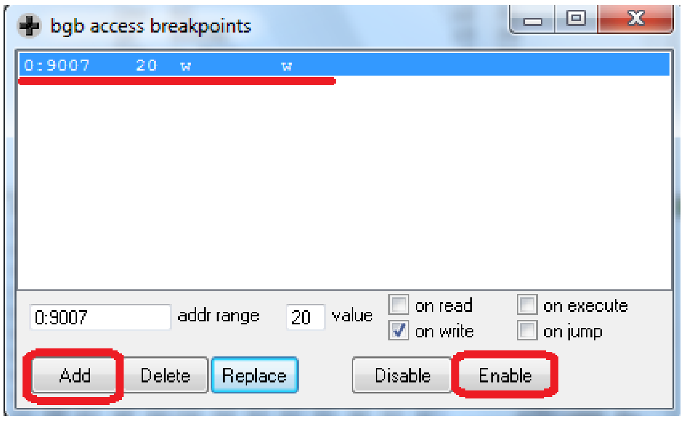
```
ROM0:006B F0 B6             ldh  a,[$FFB6]        ;3  76

VRA0:8FE0  FF 00 FF 00 FF 00 FF 00|FF 00 FF 00 FF 00 FF 00| ................
VRA0:8FF0  00 00 00 00 00 00 00 00|00 00 00 00 00 00 00 00| ................
VRA0:9000  FF 00 FF 00 FF 3F E0 20|C0 40 C0 4F C0 4F 80 9F| .....?à À@ÀOÀO€Ÿ
VRA0:9010  FF 00 FF 00 FF C0 3F 3F|00 00 00 C0 00 FF 00 FF| .....À??...À....
VRA0:9020  FF 00 FF 00 FF 00 FF 80|7F 7F 00 00 00 80 00 FF| .......€.....€..
VRA0:9030  FF 00 FF 00 FF 00 FF 7F|80 80 00 00 00 3F 00 3F| ........€€...?.?
VRA0:9040  FF 00 FF 00 FF 00 FF 80|7F 78 07 07 00 00 00 F8| .......€.x.....ø
VRA0:9050  FF 00 FF 00 FF 00 FF 00|FF 00 FF F0 1F 10 1F 10| ...........ð....
VRA0:9060  FF 00 FF 00 FF 00 FF 00|FF 00 FF 00 FF 00 FF 00| ................
VRA0:9070  FF 00 FF 00 FF 44 FF 42|FF 42 FF 42 FF 48 FF 30| .....D.B.B.B.H.0
VRA0:9080  FF 00 FF 00 FF E0 9F 99|86 86 80 80 80 98 80 9C| .....àŸ™††€€€.€œ
VRA0:9090  FF 00 FF 00 FF 00 FF DE|21 21 10 14 00 46 00 64| .......Þ!!...F.d
```

9007

Now make a breakpoint in the debugger, so it tells you when that happens.

- right-click and select “Set access breakpoint”

You want to see when that address is written on with that value, so make sure “on write” is the only checkbox. For now. every other checkbox will make it stop for things you are not interested in, which will be useless effort.

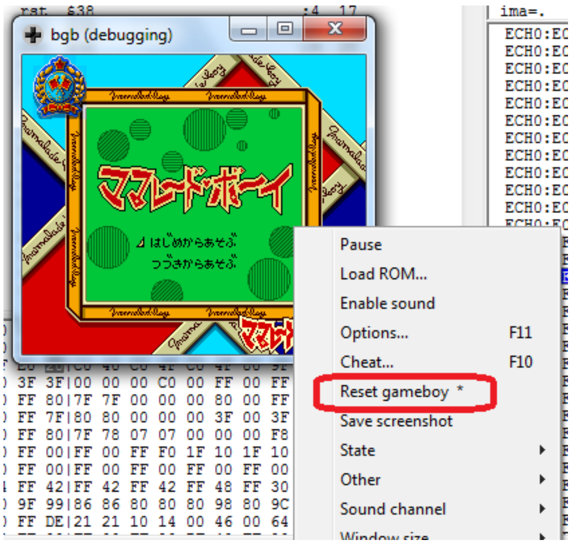


```
0:9007    20  w      w
```

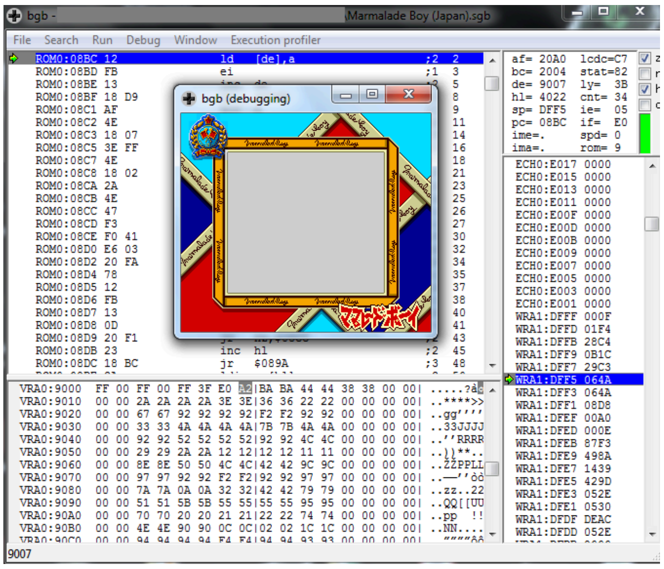
0:9007 addr range, 20 value, on read [ ], on write [✓], on execute [ ], on jump [ ]

The debugger doesn’t have a rewind, so you want to reset the emulation to make the game go back and redo the operations until it reaches the breakpoint.

Right-click the game screen and select “Reset gameboy



After that, the game will load and pause right before the value is updated to the one we are expecting. So, don’t panic. You can see the screenshot of that moment below:


bgb - Marmalade Boy (Japan).sgb
File Search Run Debug Window Execution profiler
bgb (debugging)
ママレードボーイ

```
ROM0:08BC 12          ld   [de],a        ;2  2
ROM0:08BD FB          ei                 ;1  3
ROM0:08BE 13                                 5
ROM0:08BF 18 D9                              8
ROM0:08C1 AF                                 9
ROM0:08C2 4E                                11
ROM0:08C3 18 07                             14
ROM0:08C5 3E FF                             16
ROM0:08C7 4E                                18
ROM0:08C8 18 02                             21
ROM0:08CA 2A                                23
ROM0:08CB 4E                                25
ROM0:08CC 47                                26
ROM0:08CD F3                                27
ROM0:08CE F0 41                             30
ROM0:08D0 E6 03                             32
ROM0:08D2 20 FA                             34
ROM0:08D4 78                                35
ROM0:08D5 12                                37
ROM0:08D6 FB                                38
ROM0:08D7 13                                40
ROM0:08D8 0D                                41
ROM0:08D9 20 F1                         ;2  43
ROM0:08DB 23          inc  hl            ;2  45
ROM0:08DC 18 BC       jr   $089A         ;3  48

VRA0:9000  FF 00 FF 00 FF 3F E0 A2|BA BA 44 44 38 38 00 00|  .....?à
VRA0:9010  00 00 2A 2A 2A 2A 3E 3E|36 36 22 22 00 00 00 00|  ..****>>
VRA0:9020  00 00 67 67 92 92 92 92|F2 F2 92 92 00 00 00 00|  ..gg''''
VRA0:9030  00 00 33 33 4A 4A 4A 4A|7B 7B 4A 4A 00 00 00 00|  ..33JJJJ
VRA0:9040  00 00 92 92 52 52 52 52|92 92 4C 4C 00 00 00 00|  ..''RRRR
VRA0:9050  00 00 29 29 2A 2A 12 12|12 12 11 11 00 00 00 00|  ..))**..
VRA0:9060  00 00 8E 8E 50 50 4C 4C|42 42 9C 9C 00 00 00 00|  ..ŽŽPPLL
VRA0:9070  00 00 97 97 92 92 F2 F2|92 92 97 97 00 00 00 00|  ..——''òò
VRA0:9080  00 00 7A 7A 0A 0A 32 32|42 42 79 79 00 00 00 00|  ..zz..22
VRA0:9090  00 00 51 51 5B 5B 55 55|55 55 95 95 00 00 00 00|  ..QQ[[UU
VRA0:90A0  00 00 70 70 20 20 21 21|22 22 74 74 00 00 00 00|  ..pp  !!
VRA0:90B0  00 00 4E 4E 90 90 0C 0C|02 02 1C 1C 00 00 00 00|  ..NN....
9007
```

```
af= 20A0   lcdc=C7
bc= 2004   stat=82
de= 9007   ly=  3B
hl= 4022   cnt= 34
sp= DFF5   ie=  05
pc= 08BC   if=  E0
ime=.      spd= 0
ima=.      rom= 9

ECH0:E017 0000
ECH0:E015 0000
ECH0:E013 0000
ECH0:E011 0000
ECH0:E00F 0000
ECH0:E00D 0000
ECH0:E00B 0000
ECH0:E009 0000
ECH0:E007 0000
ECH0:E005 0000
ECH0:E003 0000
ECH0:E001 0000
WRA1:DFFF 000F
WRA1:DFFD 01F4
WRA1:DFFB 28C4
WRA1:DFF9 0B1C
WRA1:DFF7 29C3
WRA1:DFF5 064A
WRA1:DFF3 064A
WRA1:DFF1 08D8
WRA1:DFEF 00A0
WRA1:DFED 000E
WRA1:DFEB 87F3
WRA1:DFE9 498A
WRA1:DFE7 1439
WRA1:DFE5 429D
WRA1:DFE3 052E
WRA1:DFE1 0530
WRA1:DFDF DEAC
WRA1:DFDD 052E
```

The values displayed in the emu are:

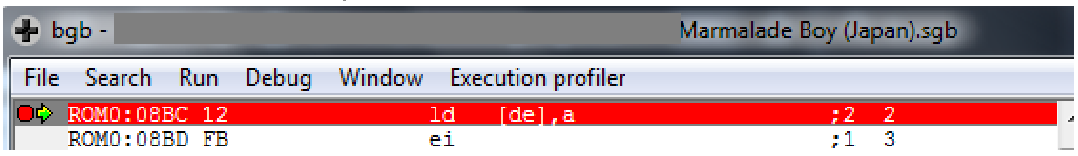
ROM0:08BC 12 ...

VRA0:9000
FF 00 FF 00 FF 3F E0 A2 BA BA 44 44 38 38 00

This is the operation that writes the value to the memory position we asked the emulator to watch. **ROM0:08BC 12 ld [de],a**

This operation is "Load A into the address pointed to by DE". Essentially, as expected, it writes A (which contains 0x20).

Tip: you can set a Breakpoint here then disable it from the "Debug → breakpoints" menu in order to save it as a bookmark). Just double click on it, the line will be highlighted in red to show the breakpoint.

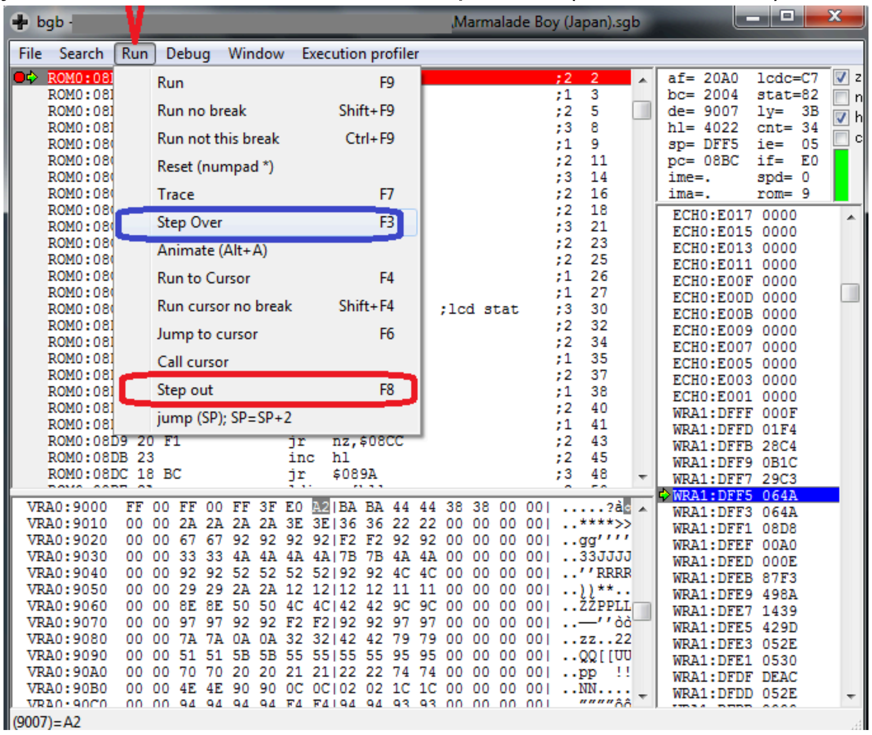

bgb - [redacted] Marmalade Boy (Japan).sgb
File Search Run Debug Window Execution profiler

```
ROM0:08BC 12           ld   [de],a          ;2  2
ROM0:08BD FB           ei                   ;1  3
```

However, typically the function we end up in from doing this isn't the one we need, so we want to go higher. BGB has a nice function for this!

Select "Run → Step out" (circled in red) to go up to the calling function. We will end up on the address after the call to this function.

If for any reason that doesn't work or you want to read all instructions from that function, you can do a Select "Run → Step over" (circled in blue)



```
ROM0:08D9 20 F1        jr   nz,$08CC        ;2  43
ROM0:08DB 23           inc  hl              ;2  45
ROM0:08DC 18 BC        jr   $089A           ;3  48
```

```
af= 20A0   lcdc=C7
bc= 2004   stat=82
de= 9007   ly=  3B
hl= 4022   cnt= 34
sp= DFF5   ie=  05
pc= 08BC   if=  E0
ime=.      spd= 0
ima=.      rom= 9
```

```
ECH0:E017 0000
ECH0:E015 0000
ECH0:E013 0000
ECH0:E011 0000
ECH0:E00F 0000
ECH0:E00D 0000
ECH0:E00B 0000
ECH0:E009 0000
ECH0:E007 0000
ECH0:E005 0000
ECH0:E003 0000
ECH0:E001 0000
WRA1:DFFF 000F
WRA1:DFFD 01F4
WRA1:DFFB 28C4
WRA1:DFF9 0B1C
WRA1:DFF7 29C3
WRA1:DFF5 064A
WRA1:DFF3 064A
WRA1:DFF1 08D8
WRA1:DFEF 00A0
WRA1:DFED 000E
WRA1:DFEB 87F3
WRA1:DFE9 498A
WRA1:DFE7 1439
WRA1:DFE5 429D
WRA1:DFE3 052E
WRA1:DFE1 0530
WRA1:DFDF DEAC
WRA1:DFDD 052E
```

```
VRA0:9000  FF 00 FF 00 FF 3F E0 A2|BA BA 44 44 38 38 00 00|  .....?à
VRA0:9010  00 00 2A 2A 2A 2A 3E 3E|36 36 22 22 00 00 00 00|  ..****>>
VRA0:9020  00 00 67 67 92 92 92 92|F2 F2 92 92 00 00 00 00|  ..gg''''
VRA0:9030  00 00 33 33 4A 4A 4A 4A|7B 7B 4A 4A 00 00 00 00|  ..33JJJJ
VRA0:9040  00 00 92 92 52 52 52 52|92 92 4C 4C 00 00 00 00|  ..''RRRR
VRA0:9050  00 00 29 29 2A 2A 12 12|12 12 11 11 00 00 00 00|  ..))**..
VRA0:9060  00 00 8E 8E 50 50 4C 4C|42 42 9C 9C 00 00 00 00|  ..ŽŽPPLL
VRA0:9070  00 00 97 97 92 92 F2 F2|92 92 97 97 00 00 00 00|  ..——''òò
VRA0:9080  00 00 7A 7A 0A 0A 32 32|42 42 79 79 00 00 00 00|  ..zz..22
VRA0:9090  00 00 51 51 5B 5B 55 55|55 55 95 95 00 00 00 00|  ..QQ[[UU
VRA0:90A0  00 00 70 70 20 20 21 21|22 22 74 74 00 00 00 00|  ..pp  !!
VRA0:90B0  00 00 4E 4E 90 90 0C 0C|02 02 1C 1C 00 00 00 00|  ..NN....
```

(9007)=A2

If we step over, we want to find the jump that makes the routine go back to its caller, if we step over, we'll land at the same place. If we did it right, for our example, you will be right at **ROM0:089A** which, lucky for this project, is the beginning of our compression routine.

(hl) = data pointer
ROM0:089A 2A ldi a,(hl)   ;get next byte, then inc data pointer

* Note: In many cases, that routine could be a subroutine or part of the patterns in the compression routine. For those cases you will have to spend more time with “step into” and “trace” until you go back to the main compression routine. Meanwhile you can comment and write up every operation the routine does, to use that info later for creating your own decompression tool.

In any case, from this point on, you set a breakpoint here (restart the emulation), and start to analyze the whole routine:

The important thing to do is watch the registers and how they change as well as what memory is being accessed. These are the byte changes that we want to document and know, since this is what the routine does (to decompress) and where (to what is being decompressed or loaded)

- To keep the same sample pattern for the whole document, as a way to compare and find consistency through the shared data between sections, I will copy the `BB` pattern below with comments.

```
; Pattern found: BB 05 - will write 00 five times
ROM0:089B FE BB cp a,BB
ROM0:089D 28 22 jr z,08C1              ;write value 00, next byte is count
; …
; After the jump, the pattern is expanded into what the code does with the data
;BB routine
ROM0:08C1 AF xor a             ;a = 00
ROM0:08C2 4E ld c,(hl)         ;c = count
ROM0:08C3 18 07 jr 08CC                ;wait for H-Blank and write byte
```

A bit of extra explanation here to know what we want to look for in a more generic approach:

- First, we locate the jump, which is the place where the subroutine for the pattern will be “called”

- Second, we note what the code does right before, because that’s the data it prepares and the data that wants to work with, in the subroutine.

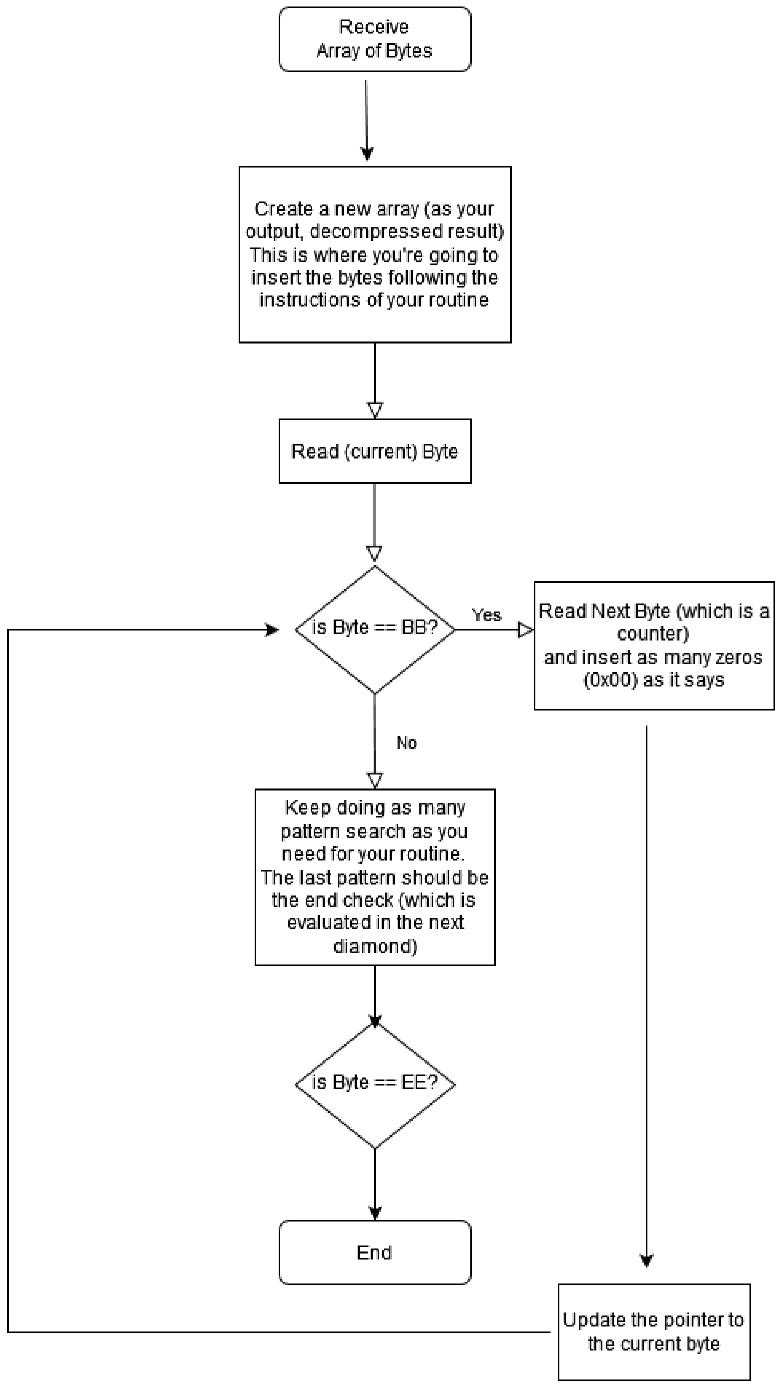
- Third and last, we follow what the subroutine does by reading its operations with the received data, and try to guess what it can mean at a higher level (this last bit is probably the harder and the one you need to comment and practice the most).

Creating the tools - Decompression

This section uses the data from the previous analysis. The complexity here resides on how good your analysis is and which method you used, but the idea is the same: Create a script to launch that routine at will without the game emulator running. This means that you are going to take each step of the compression algorithm and make it into code in your coding language of choice. I will be using python here, but you can do the same with any other.

Below, the main flow is explain but only the first pattern is focused on, so you have a general idea without being overwhelmed, and at the same time, you have one pattern to check against:

The flowchart based on the Static Analysis is shown below:



Receive Array of Bytes
Create a new array (as your output, decompressed result) This is where you're going to insert the bytes following the instructions of your routine
Read (current) Byte
is Byte == BB?
Yes
Read Next Byte (which is a counter) and insert as many zeros (0x00) as it says
No
Keep doing as many pattern search as you need for your routine. The last pattern should be the end check (which is evaluated in the next diamond)
is Byte == EE?
End
Update the pointer to the current byte

Using this method, you need to develop your own flowchart. Hence why I am showing you one for this pattern, as example

VRAM $9010: FF 00 FF 00 FF C0 3F 3F 00 00 00 C0 00 FF 00 FF

ROM $2402A: FF 00 FF 00 FF C0 3F 3F BB 03 C0 00 FF 00 FF AA

> From these two, we can see a pattern where 'BB 03' means take the first Byte and repeat three times '00 00 00'. no control code at the end, just the next byte to display

The Idea and code based on the Dynamic Analysis is shown below:

```
BB 05 - will write 00 five times
ROM0:089B FE BB cp a,BB
ROM0:089D 28 22 jr z,08C1                 ;write value 00, next byte is count

ROM0:08C1 AF xor a              ;a = 00
ROM0:08C2 4E ld c,(hl)          ;c = count
ROM0:08C3 18 07 jr 08CC                   ;wait for H-Blank and write byte
```

Do the other patterns until reaching the end's control code and finish

* Note: You have seen the approach depending on which analysis method you want. As for Pros and Cons:

- With the static method, you will have to somehow make yourself flowcharts of how the data evolves and why.

- With the dynamic method you will have to port what the code does from one language to another.

If you followed the previous 'BB' patterns and made them into python, you will have something like this

```
def decompress(data):
    output = []            # create an output file
    i = 0
    while i < len(data):          # loop through the compressed  file and
        byte = data[i]
        i += 1
        if byte == 0xBB:     # check your patterns
            count = data[i]
            i += 1
            output.extend([0x00] * count)    # Insert your bytes accordingly

# The code for the other patterns that would follow is omitted.
# This is to help visualize and highlight some key points at the end of the routine
# Those keys are written (and coded) below

        elif byte == 0xEE:   # Control code, Nobody wants an endless loop
            break
        elif byte == 0x99:   # Be careful with overlooking some control patterns.
            value = data[i]        # They might be scarce, so pay attention.
            i += 1
            output.append(value)
        else:                      # Don't forget to copy the bytes that belong to no
            output.append(byte)        # pattern.   They are part of the file too.
    return output
```

Creating the tools - (Re)compression

This part of the document might be the one closest to dev than RE. However you can still do some steps of the analyzing methods to create and check your tool.

Warning: In contrast to the decompression method, you might have "limit cases" that won't be seen until you test the tool. The compression method has to be "exactly the same" not better and not worse, any difference will make the game read the inserted (compressed) image wrong.

- You will need a flowchart, based on the patterns you used for decompression, but this time you have to code the tool to create the control bytes instead of finding them. See next page if you want to see an example for this project

- You have to take into account what can be considered a pattern and what must be passed as a plain Byte (one that is copied ‘as is’).

- You must not forget unique Bytes. An example in our marmalade routine would be the '0x99'. When you decompressed those, you probably saw them repeated and passed them as non important, but now you must be careful because if you see a '0x99', you have to insert an extra '0x99' to indicate to the game's decompression routine not to take it as a pattern warning for the next byte read.

```
# Check unique Bytes inside the loop through the file.
if input_data[i] in [0xBB, 0xCC, 0xDD, 0xAA, 0x99, 0xEE]:
    compressed.extend([0x99, input_data[i]])
else:
    compressed.append(input_data[i])
```

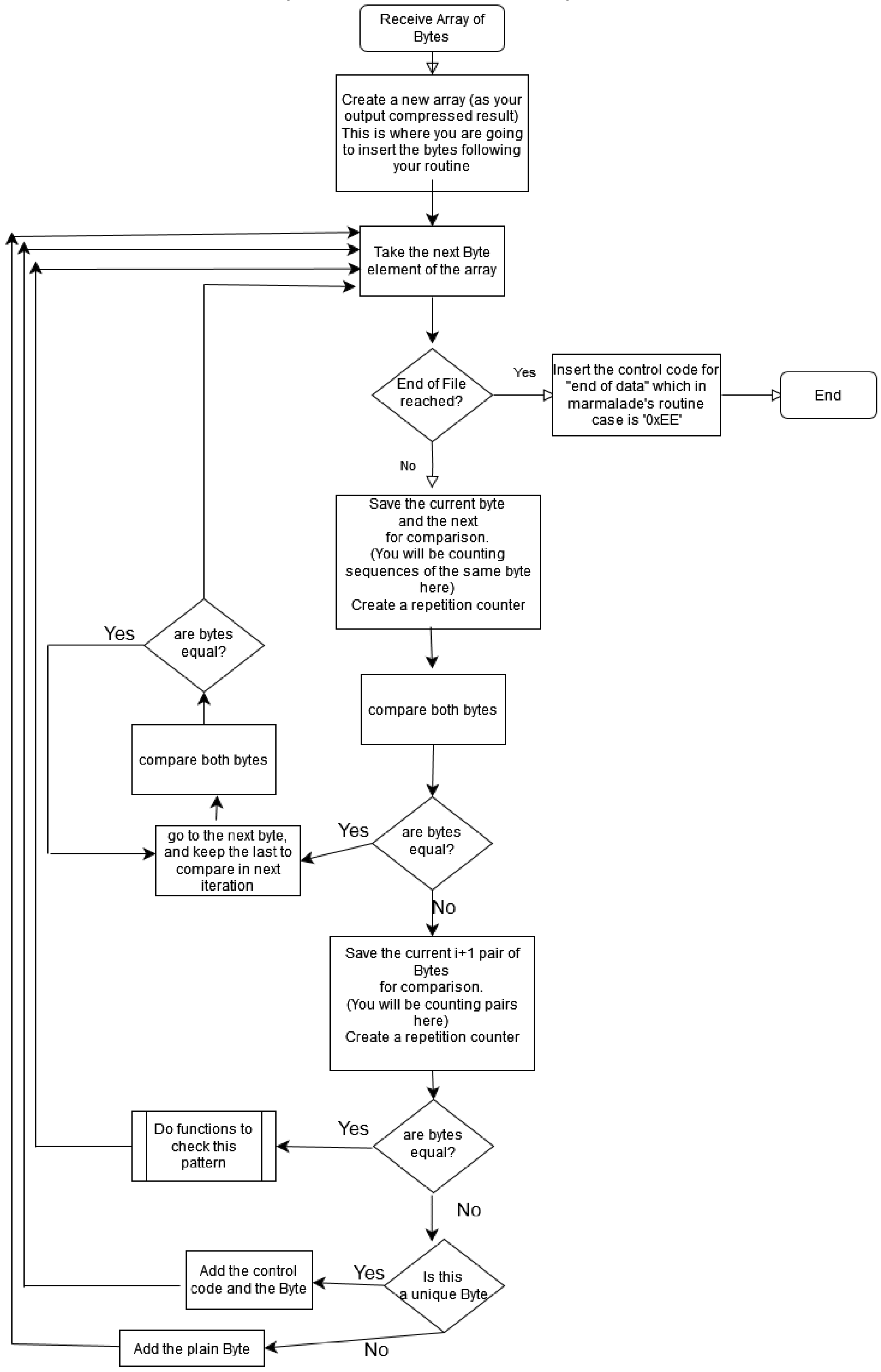
- Remember to add the "END" control code, in our marmalade boy's routine was '0xEE', but in some cases can be the length of the image file at the beginning or at the end of the file. If you don't do it, you may find errors like endless files or RAM overloads.

```
# After looping through the elements of your array/file, before the exit
compressed.append(0xEE)  # Add the “END” control code
return compressed
```

- Your compression routine and its notes will come handy to develop a counterpart.

* Hint: You can use the original compressed image from the rom to compare with your compressed one side by side to find errors in the algorithm you have coded.

Flowchart, based on the patterns used for decompression:


Receive Array of Bytes
Create a new array (as your output compressed result) This is where you are going to insert the bytes following your routine
Take the next Byte element of the array
End of File reached?
Yes
Insert the control code for "end of data" which in marmalade's routine case is '0xEE'
End
No
Save the current byte and the next for comparison. (You will be counting sequences of the same byte here) Create a repetition counter
compare both bytes
are bytes equal?
Yes
go to the next byte, and keep the last to compare in next iteration
compare both bytes
are bytes equal?
Yes
No
Save the current i+1 pair of Bytes for comparison. (You will be counting pairs here) Create a repetition counter
are bytes equal?
Yes
Do functions to check this pattern
No
Is this a unique Byte
Yes
Add the control code and the Byte
No
Add the plain Byte

Snippet of the code for beginning of the routine and for the “Same-Byte” Sequence pattern finding:

```
def compress(input_data):
    compressed = bytearray()
    i = 0
    while i < len(input_data):
    # Same-Byte Sequence pattern finding
```

** Note: It uses a counter for 3 because 2 wouldn't save any space. Example: '00 00' would become 'BB 02', both cases require 2 bytes. [This can also be considered a special or limit case since some routines just transform the '00 00' into 'BB 02' either way].*
Flowchart example for this compression routine

```
    if i + 1 < len(input_data):
    count = 1
    while i + count < len(input_data) and input_data[i] == input_data[i + count]:
        count += 1

    if count >= 3: # A sequence has been found

# The code for the other patterns that would follow is omitted.
# This is to help visualize and highlight some key points at the end of the routine
# Those keys are written (and coded) below

    # Check unique Bytes inside the loop through the file

    if input_data[i] in [0xBB, 0xCC, 0xDD, 0xAA, 0x99, 0xEE]:
        compressed.extend([0x99, input_data[i]])
    else:
        compressed.append(input_data[i])

    # After looping through the elements of your array/file, before the exit

    compressed.append(0xEE)  # Add the “END” control code
    return compressed
```

Credits and Special Thanks

- This doc (design and all images) was made (written) and redacted by Bunkai.

- First of all, thanks to the devs of the game, this doc wouldn't exist without it.

- Thanks to Abridgewater and YasaSheep for their phenomenal documents on the Compression Data Analisis (and their help when I asked), without them this doc might not exist or would have taken way longer.

- Thanks to Phonymike for his help on explaining and commenting most of the ASM routine to me. This was done two years ago, but as you see, it was well worth it.

- Thanks to 343 for coding the compression and decompression tools, and for virtually being a partner in those stages. All the python code written in this doc is his,
My own was a work in progress dropped when he beat me to do it.

Hopefully this paper helps many people out there, so thanks to you, readers for reading and sharing my words.