RomHacking 101

How to make a proper Script Dump

(Last update 23 / Apr / 2022)

Index

Introduction & Tools:

This document will guide you towards building a proper script dumper from scratch, using a GBC game as an example. Before you begin, you may want to read http://www.romhacking.net/start/ first. However many of the techniques, processes and knowledge involved may be (and is probably) useful in other game consoles/hardware.

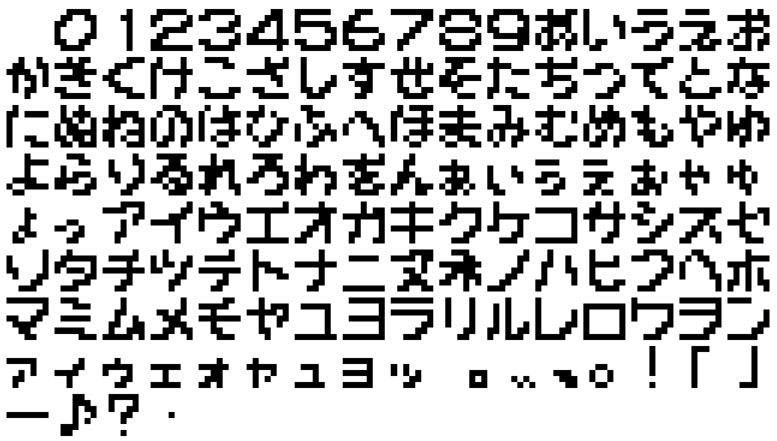
For this document/tutorial, you will need to have the following tools.

Windhex: https://www.romhacking.net/utilities/291/
Notepad++: https://notepad-plus-plus.org/
A hex calculator, any will do.
Game's ROM: Kikansha Thomas - Sodor-tou no Nakamatachi (J) [C][!]
BGB emulator: https://bgb.bircd.org/
Cartographer: https://www.romhacking.net/utilities/647/

Preface:

ROMhacking is like being a detective. You have to find answers through determination and by not giving up. You will find clues along the way, but each ROMhack is an individual journey with many obstacles to overcome. The end result is not that of following instructions, but of carving your own path.

Step 01, Creating/Obtaining a table file (.tbl)



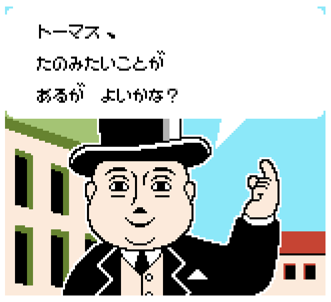
[1]

This document is based on a chat, and the chat started with a snapshot of the font [1] which provided the order for the table, however most of the document explanations don’t take that into consideration in order to better demonstrate the entire approach. Besides, you could use Monkey Moore which is the same procedure but automated.

Open the BGB emulator and load the game’s ROM.

If you pause the game when any sentence is displayed and open the VRAM viewer: [Mouse Right click, other → VRAM viewer], you will be able to see the tiles’ IDs (used in the ROM as hex strings).
Using that, you do the relative search with a text string encoded as hex (which will be explained next).

The first sentence in the game is [2],



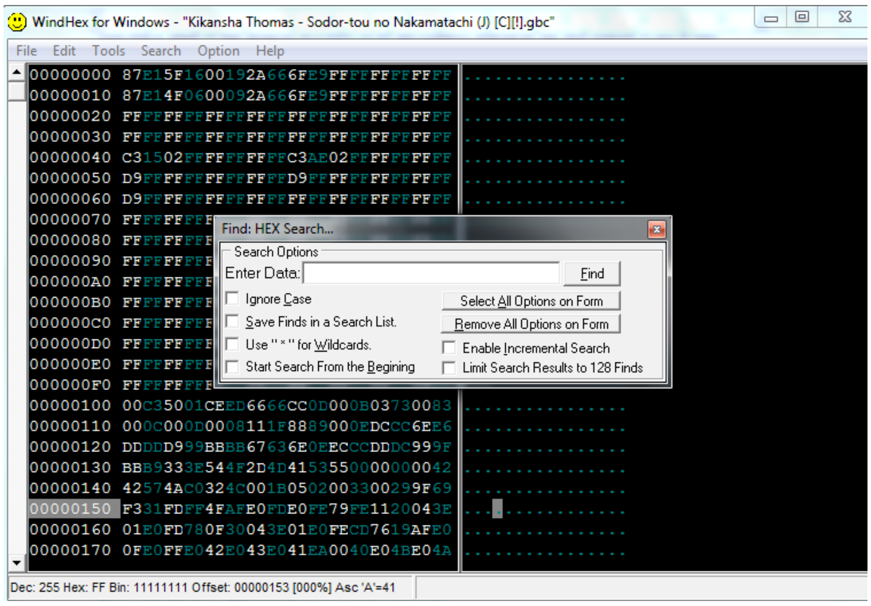
トーマス, たのみたいことが あるがよいかな?

[2]

Since you already have the font (order) [1]. Assuming it starts at 00, when you search for the word トーマス (by characters) the result will be トーマス=55 80 60 4E.
[You could also try a Monkey Moore search of the word for the same goal]
Keep in mind that some games have data structures describing the font characters more directly where this assumption doesn't work!

Anyways, back to our project, it’s time to do the relative search/or the hex search to see if these values are used in the game.

Building the hex string for the whole first sentence you have:



WindHex for Windows - "Kikansha Thomas - Sodor-tou no Nakamatachi (J) [C][!].gbc"
File Edit Tools Search Option Help

```
00000000 87E15F1600192A666FE9FFFFFFFFFFFF  ................
00000010 87E14F0600092A666FE9FFFFFFFFFFFF  ................
00000020 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF  ................
00000030 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF  ................
00000040 C31502FFFFFFFFFFC3AE02FFFFFFFFFF  ................
00000050 D9FFFFFFFFFFFFFFD9FFFFFFFFFFFFFF  ................
00000060 D9FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF  ................
00000070 FFFFFFFF
00000080 FFFFFFFF
00000090 FFFFFFFF
000000A0 FFFFFFFF
000000B0 FFFFFFFF
000000C0 FFFFFFFF
000000D0 FFFFFFFF
000000E0 FFFFFFFF
000000F0 FFFFFFFF
00000100 00C35001CEED6666CC0D000B03730083  ................
00000110 000C000D0008111F8889000EDCCC6EE6  ................
00000120 DDDDD999BBBB67636E0EECCCDDDC999F  ................
00000130 BBB9333E544F2D4D415355000000000042  ................
00000140 42574AC0324C001B05020033300299F69  ................
00000150 F331FDFF4FAFE0FDE0FE79FE1120043E  ................
00000160 01E0FD780F30043E01E0FECD7619AFE0  ................
00000170 0FE0FFE042E043E041EA0040E04BE04A  ................
```

Find: HEX Search...

Search Options

Enter Data: [ ] Find

- [ ] Ignore Case
- [ ] Save Finds in a Search List.
- [ ] Use " * " for Wildcards.
- [ ] Start Search From the Begining
- [ ] Enable Incremental Search
- [ ] Limit Search Results to 128 Finds

Select All Options on Form | Remove All Options on Form

Dec: 255 Hex: FF Bin: 11111111 Offset: 00000153 [000%] Asc 'A'=41

[3]

トーマス,=55 80 60 4E 7B
たのみたいことが = 1A 23 2A 1A 0C 14 1E

Those are the hex values from the font that you will be looking for; in the hex editor (Windhex). [Keep in mind you will need a hex editor with .tbl (and Japanese) support if you prefer something other than Windhex]

Go to Search → Hex Search, or press Ctrl+H [3], and type the hex string you made previously (トーマス,=55 80 60 4E 7B)
Here you are trying to find the text string (in hex form) inside the ROM, and that's why you're using those numbers. [The programs leads you to the address 49A52]

Mind you トーマス can be used more than 1 time, that's why you need to see if the string for the following sentence たのみたいことが is nearby: (usually with a control code in-between)
[This second line is at address 49A70]

Next is creating the table file (.tbl), with that you will be able to read actual Japanese in the hex editor without doing the conversion between hex and symbols manually.

Go to Tools → Table Maker, or press Ctrl+M, and the Table maker will pop up.
[Windhex creates the table with Japanese encoding, use Shift+J to view the proper keys]

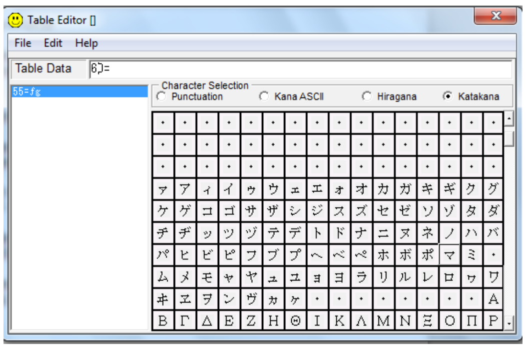


Table Data: 6]=

55=fg

Character Selection: Punctuation ( ) Kana ASCII ( ) Hiragana ( ) [✓ Katakana]

・	・	・	・	・	・	・	・	・	・	・	・	・	・	・	・
・	・	・	・	・	・	・	・	・	・	・	・	・	・	・	・
・	・	・	・	・	・	・	・	・	・	・	・	・	・	・	・
ァ	ア	ィ	イ	ゥ	ウ	ェ	エ	ォ	オ	カ	ガ	キ	ギ	ク	グ
ケ	ゲ	コ	ゴ	サ	ザ	シ	ジ	ス	ズ	セ	ゼ	ソ	ゾ	タ	ダ
チ	ヂ	ッ	ツ	ヅ	テ	デ	ト	ド	ナ	ニ	ヌ	ネ	ノ	ハ	バ
パ	ヒ	ビ	ピ	フ	ブ	プ	ヘ	ベ	ペ	ホ	ボ	ポ	マ	ミ	・
ム	メ	モ	ャ	ヤ	ュ	ユ	ョ	ヨ	ラ	リ	ル	レ	ロ	ヮ	ワ
ヰ	ヱ	ヲ	ン	ヴ	ヵ	ヶ	・	・	・	・	・	・	・	・	Α
Β	Γ	Δ	Ε	Ζ	Η	Θ	Ι	Κ	Λ	Μ	Ν	Ξ	Ο	Π	Ρ

[4]

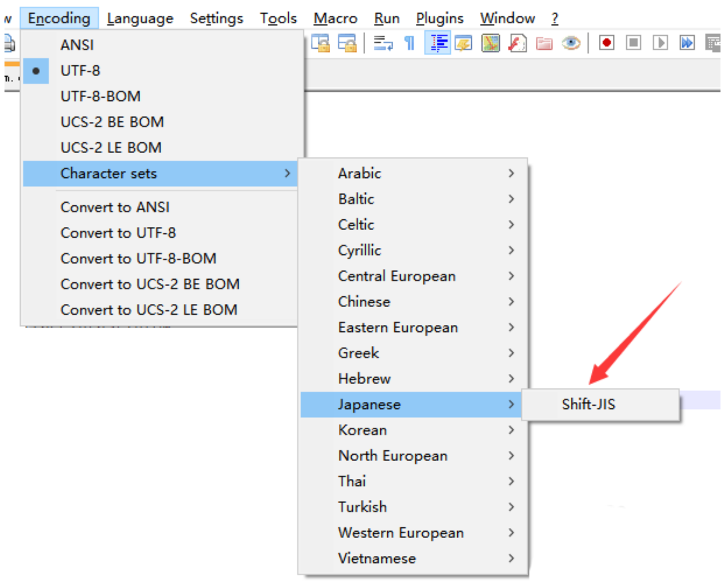
Time to fill the table with the hex and their corresponding symbols [4]

Type the hex number in the Table Data field and select the from the character selection.

[Don't be surprised if the kana doesn't show up in the table, it's still working properly, it's just a software issue.]

Start with the hex addresses that you used previously,
55= ト 80= ー 60= マ 4E= ス
After that keep going until you've made all the symbols in the font.
[In this ROM's case 132 font symbols]

You can also save it as is with the File → Save Table File, and complete it using Notepad++.
Don't forget to put the proper codification to the table file, open the tbl file in Notepad++ go to Encoding → Character set → Japanese → Shift-JIS [5]

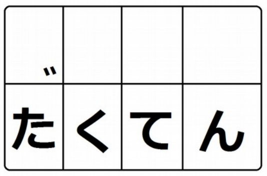


[5]

Once you finish, congratulations, you have made the Japanese table file for your game.
[In the table you have a relation of Hex Number = Symbo]

* Oh, wait! *
Looking at the rom code, they just put dakuten/handakuten with an additional byte (the hex value of Maru/Tenten) above the kana, as shown in [6].

In hex, that means が is 7A+10, or 7A+か instead of a completely new address.
[This lets them reduce the font by using compounds.]
As a result, the table requires 2 bytes for compounds (e.g. 7A10=が)
[1 for the syllable + the 1 of the dakuten]



[6]

Now each codepoint (index of a character in the table) will have 4 digits. Since a table file processes a byte stream, the bytes are in the same order that they appear in the hexdump. Which means you have to insert an additional Byte ("00") at the left of the number you had previously, so that the compound can be dumped properly.
Example: 0055=ト 0080=ー 0060=マ 004E=ス

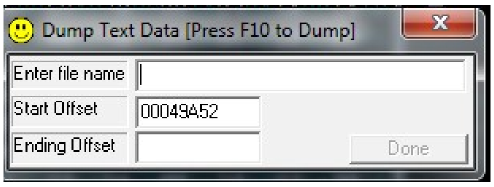
Step 02, Doing a Test Dump.

Open the ROM on Windhex and load the .tbl file (File → Open Table file).
[Make sure to use the proper encoding, Option → View Text Data as Japanese or Ctrl+D]

Go to the position of the text you found before, Search → Goto to Offset or Ctrl+G, and insert the address (offset) you found at the beginning, when you were searching the text script.
[Offset 49A52 in this case]

Can you see "トーマス、 たのみたいことが あるがよいかな" or not?
If you can, the table is correct, you can make a test dump now.

Go to Edit → Dump Text or press F10. [7]



Start Offset: 00049A52

[7]

The file name requested here is the name of the dumped text file (remember to add the .txt extension in the name), Start and Ending Offset are self-explanatory.

Since this is just a test, to decide the Ending Offset just put one beyond the end of recognizable Japanese text. For example (start & ending): 49a52 & 4b372.
[There's a terminator sequence, and you want the address of the first byte past the end of the text area.]

Step 03, Making a Proper Dump - Prologue

You now have the text but it's dirty, mixed with control codes, images, sounds, pics, etc .
It's time to reverse engineer all text strings in order to dump flawless text.

You should read https://datacrystal.romhacking.net/wiki/Pointer and https://offsecdeer.gitlab.io/post/gb-pointers/ to get ready for the next part.

So… You found the text. The question is, how does the game find the text?
It does so by using a pointer table which is what you will be looking for next.
[The table also lets you relocate text, change text lengths. and whatnot.]

How do you find the pointer table, you ask?
You have 2 text strings and you know where they start, so you can work backwards and find the table that points to all text strings.

Looking at the ROM in your hex editor, you see the first offset (beginning of the text) and, after a control code, the following offset.

[The pointer can be in various locations: in a table at the start of the text, after each text line, or even somewhere else entirely.]

Remember, in ROMhacking, you make note of what you've done, so if you find out you've done it wrong you can go back and do it better, as many times as it takes.

Now it is time to find the pointer table address.

Usually they put each pointer in a table one directly after another, so if you find two together, you can assume the following bytes are also text pointers in the game (up to a point).

[The table might contain all of the pointers, or there might be another pointer table somewhere else. After all, there is craziness like split pointer tables where the high bytes are in one table and the low bytes are in another.]

Since you already know that the text starts at 49A52 (starting offset), and the next line starts at 49A70. You can look for that pair.

[As a sanity check, you can try nearby values that look like pointers to see if they also point to suitable places in the game text.]

To help you find the table address, you will look at the same bank the text is in and use the documentation for pointers linked above. [Game Boy 2 Byte Pointers]

[Pointer tables are often contained in the same bank as the data that it points to, and usually is directly above or below the data.]

The formula to calculate the bank you are at is,
```
bankBaseAddress = (offset / 0x4000) * 0x4000
```

Time to use the hex calculator:

offset = 0x49A70

bankBaseAddress = (0x49A70 / 0x4000) * 0x4000

0x49A70 / 0x4000 is 0x12 with a remainder of 0x1A70.

You are on bank 12, and the bankBaseAddress is 48000

That means your pointers are probably between the addresses 48000 and 4BFFF in the file.

The GameBoy and GameBoy Color usually have their switchable bank "mapped" so that the CPU sees it at 0x4000-0x7FFF, then you need to take the file position, work out where it is within bank 0x12, and thus where the CPU will see it.

Switchable Bank Explanation: The CPU has a... "window", let's call it, that can show any single "bank". The CPU can change which "bank" appears in the "window" at any time. This process is called "bank switching".

[There's an extra layer to this, in that some systems don't have just one window, and sometimes there's some adjustments on a different part of the program.]
Bank 0x12 in the ROM file represents the address range 0x48000-0x4BFFF, and the start of the text (0x49A52) is within this range/bank.

Since you want to find 0x49A52 (& the pointer table), you have to:

1. Subtract the starting position of the bank from the location of the data being pointed to.
   0x49A52 – 0x48000 = 0x1A52

2. Then add 0x4000 because that is where the bank appears to the GameBoy CPU.
   0x1A52 + 0x4000=0x5A52

Since this is hexadecimal and each pair of digits represents a byte, you break them by pairs, 5A 52

Now, the GameBoy is a little endian machine, so swap the order, to get the actual value you want to search for: 52 5A.

This is the value you search in the hex editor to try and find the pointer table.

Now you do the same with the address of the next line, $49A70. Since both pointers should be in close proximity to each other if it’s the pointer table.

1. Subtract the starting position of the bank from the location of the data being pointed to.
   $49A70-$48000=$1A70

2. Then add $4000 because that is where the bank appears to the GameBoy CPU.
   $1A70+$4000=$5A70

Don’t forget it’s little endian, so do the byte swap, 70 5A

After that, write both hex values down: 525A and 705A. Go back to the hex editor, with the game ROM loaded, and find those values.

To search, you can use the shortcut Ctrl+H in Windhex.
You're looking for 52 5A to find candidates, with a nearby 70 5A to see if it's the correct one.

You don't search the full 52 5A 70 5A in case there's a gap between the two pointers for whatever reason. [There could be arbitrary data, duplicates… You need to find the correct occurrence]

If you searched for 705A, then you found no results, therefore it doesn’t exist.

You can only search for 525A now.

You have to keep searching for the next one until you recognize a pattern around it that leads you to think it is a pointer sequence (pointer table).
[e.g. like the pattern you would see if you followed the compounds of the kanas in a table.]

The first match is at 0x713D0, but it doesn’t look like a pointer table or a pointer sequence.
Next match shows a recognizable pattern: B1 5A, E2 5A. You may have found the pointer table at 0x49850.

With a general pattern to watch for, and specific examples. You can figure out the relationship between text and pointers.

It’s time to reverse the process with these pairs of bytes to find out where they point to.
This way you can see the text and its code patterns

The procedure is analogous to finding the pointers with the text, but now you’ll be finding text with the pointers instead. This procedure lets you use those memory addresses so you will be able to look for patterns.

When you finish, you will either see a "terminator" sequence just before the later pointers or you will see a length somewhere.

[Or, very unlikely, the game is using the distance between two pointers to determine the length. Even more unlikely is each entry having the same length.]

Now, back to finding the text with the pointers, the process is the previous one but reversed, you can follow the step by step next.

First, since you got the pointer from the memory, it’s in little endian and you’ll work with it in “human” order, so you will swap the pointer (B1 5A), to 5A B1

1. Subtract 0x4000, which is where the bank appears to the GameBoy CPU, in order to find the offset within the bank.
   0x5AB1 – 0x4000 = 0x1AB1

2. Then add 0x48000, which is the starting position of the bank within the ROM file.
   0x1AB1 + 0x48000 = 0x49AB1

With the address calculated, you go and see what (text) you see at 49AB1.

Once you go to that address in your hex editor, you will see the text with control codes (bytes that are not text themselves). If everything else is good, these bytes will have a pattern.
[The bytes will appear at the end of the line, at the end of the scene, etc.]

Within the hex editor, as soon as you go to the text you will recognize the following pattern
(for this game): Notice the bunch of 72s and 73s, those look like pointers. Also:

FF00 = .0 ;
FF01 = .1 ;
FF02 = .2 ;

If you look at them closely, it looks like it’s probably:

FF00 = .0 is a BR line
FF01 = .1 is a NXT line
FF02 = .2 is an END of scene

So you have a general pattern to watch for and specific examples in order to figure it out. Don’t forget to take notes as you go.

After all this hard work you can now begin the cartographer part, so that you can make a proper script dump.

Step 04, Making a Proper Dump - Cartographer

This document won't go over the information which you find in the tool's readme. However, for text fluidity, some explanations and examples will be copied, starting with the basic command to make it work.

```
cartographer rom.ext commands.txt script.txt -s/-m
```

As stated in Cartographer's readme "commands.txt contains the dumping commands (see below) that specify how to dump the text.". And that's the only file you really need to create/edit at this point.

So, again, at 0x49850 (pointer table address) you can see the offsets (pointers) of the script.

Using the values of these pointers, you know the locations for the text in the ROM. [Don't forget that they are in little endian]

The first ones are 0x49850 and 0x49A52, which suggest (means) that:

- 0x49850 is the offset of the table start pointer.
- 0x49A52 offset with the first pointer of the text.

You'll especially need those for the command.txt and the dump.

You can figure out what the others are, from playing through the scene on the emulator or real hardware and comparing that to the script.

[Remember to take copious notes as to what you're seeing, what you're figuring out, and your working hypotheses. It will save you a lot of trouble later]

Seems that going to the address pointed at by 0x49A52, the first value there is at 0x525A, and with the byte swap reverse them to not be little endian, 0x5A52.
[This will be used to calculate the base pointer for Cartographer to know where to start dumping text from]

For the cartographer's command.txt:
Cartographer will be reading this table (at 0x49850) in order to dump the text, so it needs to know how to calculate the ROM locations from the pointers listed in the table.

In order to do this, you have to calculate what's called the "base pointer", which Cartographer will add the pointers to in order to calculate the ROM address.

So, take the text's ROM address (0x49A52) and its pointer value from the pointer table (0x5A52), and subtract them to find the base pointer, like so:

0x49A52 - 0x5A52 = 0x44000.

The base pointer value for the tool is 0x44000.

Gameboy (GB/GBC) addresses are always between 4000-7FFF.
If it's the first bank which is loaded at all times it's 0000-3FFF but that's a fixed bank usually for game code.

That's the way the CPU address works. Nothing prevents the game from having its own high level pointer system that uses 0000-FFFF or even 24 bit pointers, then converts to CPU pointers later

[For other consoles, you may want to read the documents for pointers linked above or documentation specific to said consoles]

You can assume that it's a value somewhere in the bank we've been looking at, because as discussed before, "Pointers are often contained in the same bank as the data that they point to, and are usually directly above or below the data."

If you assume that the table is before the text, then the smallest value in the table has to point to just after the table, which sets a boundary for the table.

As a last resort you can scroll like a maniac until you see were the table
ends. [At the minimum, it has to be after the starting offset you already have].

Taking that into consideration, in our project the offset is
$49A4F

Almost finished, but you probably want a nice display of the text with line breaks in the dumped text. To do that you'll need to add the control codes to your table file in the pointers part for that.
Control codes that can be added to table entries (in Cartographer):

\r	Line break with commenting (//) on the next line.
\n	Line break with no commenting on the next line.

Example:
FF00=BRK\n

Another important line of Cartographer's readme is about how you should define your endtokens so the library functions properly. Translated to our project:

```
/FF02=END\n\n
```

If you followed the instructions properly, you should get this command.txt configuration. [Values from the Thomas game example.]

```
#BLOCK NAME:                Thomas (GBC)
#TYPE:                      NORMAL
#METHOD:                    POINTER_RELATIVE
#POINTER ENDIAN:            LITTLE
#POINTER TABLE START:       $49850
#POINTER TABLE STOP:        $49A4f
#POINTER SIZE:              $02
#POINTER SPACE:             $00
#ATLAS PTRS:                Yes
#BASE POINTER:              $44000
#TABLE:                     ThomasHiragana.tbl
#COMMENTS:                  No
#END BLOCK
```

Additional notes

The 101 document ends here, if you found an error and want to report it please send a private message to Bunkai in the https://www.romhacking.net/forum/

If you have a question about the process you may ask them in the forum or in the discord, using the proper channels for that. But I won’t answer any private questions about your projects.

A translation project should include: script dump, font and sprites edit, and script insertion.
This document is part of a series to cover those 3, but any of the papers can be read standalone.
If you want to check any of the others, go to:

RomHacking 101 - Script Dump: https://www.romhacking.net/documents/871/
RomHacking 102 - Font & Sprites: https://www.romhacking.net/documents/872/
RomHacking 103 - Script Insertion: https://www.romhacking.net/documents/873/

Credits:

Guidance and explanations: 343
Additional explanations: Abridgewater, YasaSheep, Bootleg Porygon, Pluvius
Document author: Bunkai
Document reviewer: YasaSheep

Special Thanks:

Phonymike for the reminder of the quotes used as a preface.
TF1945, because all this document has been extracted from a chat with people explaining everything to him in the RHDN discord server.
To the RHDN community for their wonderful support of this hobby with manuals, tools, explanations and so on.