### **Preface**

This tutorial will go over the basic hacking knowledge required for asm, and also for other small modifications including: making tweaks, custom model importing and behavior scripts.

### **Hexadecimal**

Hexadecimal is a number system that has a base of 16. The numbers go from 0x0-0xF instead of 0-10 like in normal decimal. They are also prefixed with 0x to differentiate them from decimal numbers. A byte of data, which is 255 in decimal will be 0xFF in hexadecimal. In general, this will be the smallest amount of data used when discussing mario hacking, and coding in general.

1 byte, 0xFF can also be represented as 8 bits. A bit is just one binary piece of data, a 0 or 1. Binary will also be very important to understand for asm coding. Understanding converting between these three number systems will be integral to becoming a good asm programmer.

An easy way to think of binary in terms of hexadecimal is that each hex number is 4 bits.

* 0x0 = 0000 = 0
* 0x1 = 0001 = 1
* 0x2 = 0010 = 2
* 0x3 = 0011 = 3
* 0x4 = 0100 = 4
* 0x5 = 0101 = 5
* 0x6 = 0110 = 6
* 0x7 = 0111 = 7
* 0x8 = 1000 = 8
* 0x9 = 1001 = 9
* 0xA = 1010 = 10
* 0xB = 1011 = 11
* 0xC = 1100 = 12
* 0xD = 1101 = 13
* 0xE = 1110 = 14
* 0xF = 1111 = 15

You can extend this to higher numbers by just finding the value from 0 to 15 for each number, then expanding the powers.

* 0x2F is 16\*2 + 15 = 47 in decimal and 0010 1111 in binary
* 0x200 is 256 \* 2 = 512 in decimal and 0010 0000 0000 in binary
* 0x1303 is 4,096\*1 + 256 \* 3 + 16 \* 0 + 3 = 4,867 in decimal and 0001 0011 0000 0011 in binary

There are tons of online resources for learning hex and binary as well as many converters. It's best to learn by doing some examples.

### **RAM**

RAM (also known as random access memory) is the memory the game uses while running to store/use data. When the console starts, data is copied from the ROM (read only memory) to RAM. RAM is managed by the game by separating it into segments called banks. RAM banks in sm64 go from 0x0 to 0x1f and the game will usually refer to a piece of data's location by its bank offset rather than its location in RAM. To find something in RAM you will have to know where a bank starts and the data's offset from the bank start.

RAM starts at 0x80000000 and goes for as much memory you use. For N64 this is 4mb or 8mb depending on whether you have an expansion pack plugged in or on an emulator, it reflects the settings. Almost all sm64 rom hacks use 8mb. Bank addresses start with the bank number, then the next 3 bytes are the offset. So 0x0E 67 FD 20 is located in bank 0x0E and has an offset of 0x67 FD 20.

Ex. 1 Find an address in RAM

* 0x13 00 0F DC

This data is in bank 0x13. If we know where bank 0x13 starts we can find the ram location but I will make something up for this example.

* Bank 0x13 start = 0x80040200 (just an ex. Not the real location)
* Our data offset = 0x00 0F DC
* Our data in ram = 0x80040200 + 0x0FDC = 0x800411DC

Now we can find our data by looking in RAM.

If you know where the data started to be copied from, you can also find the location in the ROM.

Ex. 2 Find object in ROM

* 0x13 00 0F DC
* Bank 0x13 was copied from 0x219E00 in ROM
* Our data in ROM is at 0x219E00 + 0x0FDC = 0x21ADDC

Now if we want to change data at 0x13000FDC we can type it in a hex editor at that ROM location.

### **Behaviors and Banks**

Behavior scripts are a high level script system used to set up objects for use in sm64. They are made up of 4 byte aligned cmds that tell the game how to process a function. The first byte is the type of cmd, the rest are the parameters of it. For example, the “begin behavior” cmd starts with 0x00, then has 3 bytes that are for parameters. 0x00 09 00 00 is a common one used to start behaviors for solid objects.

Banks 0x0 to 0xF are used to store model data, this includes the triangles, textures and the microcode used to process them. If you want to edit a texture or a triangle, you will have to find the bank in that range, its offset and convert it to a ROM address.

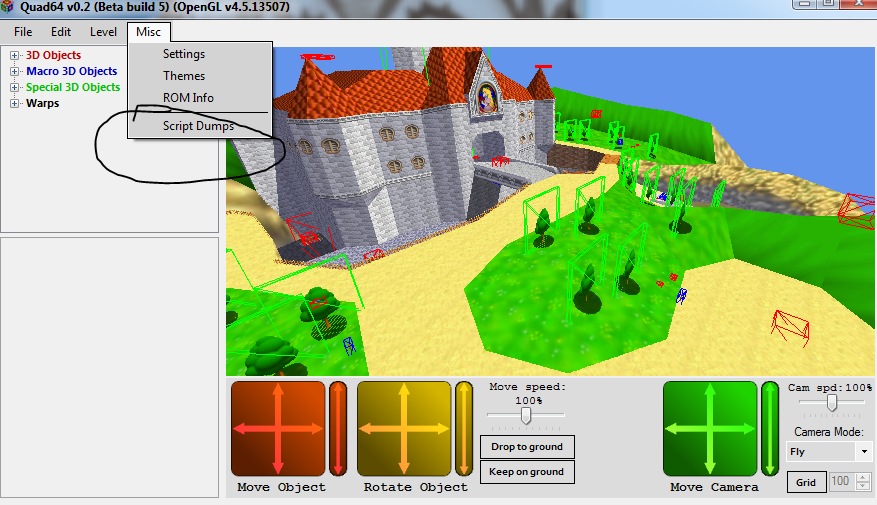
Behaviors are located in bank 0x13. If you want to edit them you should use the method described in example 2.

For any of these, you can also type the direct RAM address to reference the data or use the bank + offset method. I recommend you use the bank method because that is how the data is loaded and it will always have the same bank+offset but not always the same RAM address.

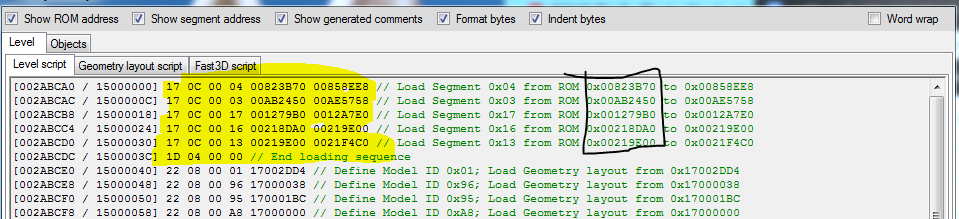
### **Finding bank starts**

The best tool to find where a bank starts is quad64 script dumps. This tool will dump the loading procedure the game uses and display it in an easy to read format. Quad64 is very useful so you should download it if you plan on doing any hacking.

After you open quad64 up, load your level and goto the script dumps tab at the top toolbar.



On the level script page of the popup, you are looking for a line that begins with 17.

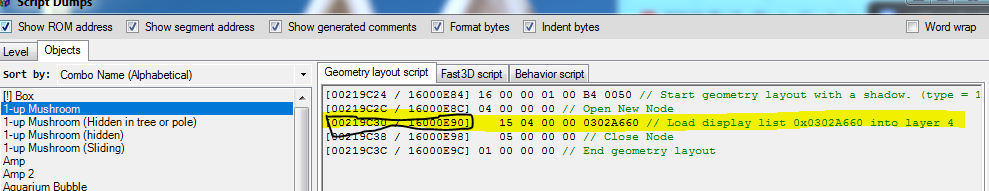


17 is a cmd that loads data from ROM to RAM. Next to this line there should be a comment in green that says something like “load segment 0x13 from 0x219e00 to 0x21f4c0”. This tells us the bank start that you can use for finding a ROM address for your data. Segment is the bank number, and then the hex numbers following it are the range. This is the best way to find where banks start and calculate a bank address for your object. Not only because you can guarantee which bank your data is in, but also because banks do not load the same data for every level.

### **Custom model importing**

Custom model importing is just a method of using bank offsets to find free space, import over it and tell the game that you put a model there. You should use the quad64 script dumps to find the start of RAM banks and then calculate your particular offsets for the data you import.

RAM offset refers to the bank + offset you start your import. ROM offset is the location you are importing over in the ROM. You should always find your ROM offset then calculate your RAM offset from that. Pointer is where you tell the game where you are importing your data to in the ROM. This will go in a geo layout. If you don't know what that is, don’t worry about it. There is a tab on the script dumps called objects. From there find your object, click it and you'll see the geo layout of the object on the right.



The 15 or 13 cmd is where you want to put the pointer. Specifically the last 4 bytes of that line shown. In the above picture it would be 0x219c34. The pointer in the model importer always refers to the spot in the ROM your pointer goes. If there are multiple 15/13 cmds then that means there are multiple models. Finding out which is which will just be experimentation.

Collision pointer will be the location in ROM of the 0x2A behavior script cmd. You need to find the behavior, calculate the ROM location as earlier explained and then find the line of the 0x2A cmd. There should also be a tab for behaviors in quad64 script dumps so you can use the same method you used for pointers.

### **Finding a place to import**

When you want to make anything custom you have to find a place to put your data. An extended rom has basically everything past 0x1200000 as free space. What you need to do is goto that space and then just scroll down until you find a free spot occupied by 0x00s or 0x01s, which is just data paddings. Past 0x1210000 is generally used as space for importing custom levels. 0x1200000 to 0x1210000 is used for asm, do not import a model there.

There is also free space located intermittently before 0x1200000 but usually no more than 0x1000 bytes but it never hurts to look. This is usually good space to fit a small custom model. A good spot to find these will be at the end of bank 0x4 or end of bank 0x3. If you see free space but it is not loaded, then you can just edit the 0x17 cmds so that your data is included in the range. If you increase it too much your game might crash so be careful.

Next you need to see where the spot you have is loaded into RAM. You can find this from the 0x17 cmds that tell your bank starts or from knowing that sections asm start location. This will mostly come from knowledge on how the game is laid out but there is a ROM map online that gives details of what goes where.

[ROM map](https://hack64.net/wiki/doku.php?id=super_mario_64:rom_memory_map)

[Bank addresses](https://hack64.net/wiki/doku.php?id=super_mario_64:list_of_segments)

You just cross reference the ROM address with the segment loading commands, which are best found using quad64 script dumps. If you still do not know or think it's not in RAM at all then there are ways to do custom loading, but it is too advanced for this tutorial.

**Step by step for custom model imports**

If you are still confused, you can use this step by step process.

First decide if you want your model to be level specific or if you want it to be used in every level.

Single level custom object:

* Find the level script of your level
* Goto the 0x22/21 cmd section
* Add a new one at the end, just copy/paste everything a line down or replace a 0x10 cmd if there are any.
* Your 0x22/21 will load a model. If you want a geo layout use a 0x22, if you don't know what that is use 0x21.
* 21 (22) 08 00 xx aa aa aa aa
* xx is your model ID, aa is the address.
* If you're using a geo layout its the address of your geo layout. If you aren't, it's the place your pointer will write to.
* Write down the ROM location of the a’s. That is your pointer.
* Next find your model import. This will be bank 0xE. Rom manager uses multiple bank 0xEs. Make sure you're in the correct one.
* Scroll down until you see free space, that is your ROM offset
* Calculate the RAM offset using the start of bank 0xE
* Find collision pointer using method described in above tut if necessary
* Import
* Find the end of your import. This will be indicated by more padding
* Goto your bank 0xE 0x17 cmd and extend the range so that it ends where your import ends.
* Double check by seeing if your chosen model ID displays in quad64.
* Note: Rom Manager does not respect custom imports in bank 0xE so everytime you import a level model you will have to repeat this process. Use a different bank (a custom bank 0x7 load or 0x5 maybe) or wait until your level is done before doing this.

Some common errors.

* P. Nothing displays
* A. Bad pointer/model ID. Check your 0x22/21 cmd
* P. Model is distorted
* A. Wrong RAM address/not all of your model is loaded. Check 0x17 cmd and your pointer
* P. Game crashes when I get near object but visuals are fine
* A. Bad collision pointer
* P. Entire level graphics are weird when I look towards object
* A. Bad RAM address
* P. I have no idea what any of this means.
* A. Read this tutorial again, go through the example yourself and read the wiki on level scripts. Then look at your script dumps and try again. Repeat as necessary.

To do it for a global object, just do the same process but change your import location to something that is in bank 0x3 or 0x4 (or make your own bank). You can extend them the same way you extended bank 0xE, just be careful to not go too far and overwrite other stuff.

### **Conclusion**

This may all seem confusing in just text but it will become easier if you actually calculate some offsets. I recommend you use windows calculator and put it into programming mode by pressing alt+3 and then you can use it to convert from decimal to binary to hexadecimal and calculate offsets. You should start with trying to do behavior offsets since you can double check if you're right with script dumps and then you will get the hang of it quickly. This knowledge is required if you want to do anything outside of what current hacking tools allow, which is not much compared to what is possible.