## Preface

All objects in sm64 have a structure that determines how they behave. Accessing and editing the values in this struct will allow full control over its properties. This tutorial will assume you understand hexadecimal and ram/rom addresses. A basic understanding of behavior scripts is helpful as well but not necessary.  
This tutorial will only cover asm, which may not be necessary to understand for some people with the advent of decompilation. Please see the decomp header for more info.

### The obj struct

You can find a reference with all the values in the obj struct here:

[sm64 structs](https://hack64.net/wiki/doku.php?id=super_mario_64:structs)

You will notice that each variable listed will have a value type, an offset and a description of what its used for. Value type is just what type of number the variable is. These can be either *float, u16, u32, u8, s16,* or *struct.* For now don’t worry about structs or floats. The *u* and *s* stand for unsigned and signed, and the number after them is the length in bits. This means 16=2 bytes and 32=4 bytes. They can also be referred to as halfs (16 bits or 2 bytes) and words (32 bits or 4 bytes). The descriptions represent what the variable does, but most variables will not behave as described in isolation, most will only work in tandem with another function.

An example is speed (0xAC, 0xB0, 0xB4). Just setting speed alone is not enough to make an object move. The object must also execute a function that processes the speed value.

There are others that will be written to. For example the angle/distance to mario values get written to under certain conditions. Others will always be active; for example position will always update to the written value.

You will have to experiment with these values to determine how each works, but doing so is relatively easy. All you need to do is access the beginning of the struct, then edit the values you want and observe the results. Throughout the tutorial I will explain how some work in examples as well.

### Using the struct

First thing to do when you want to edit an object is you need to execute a function. In game this is done with a behavior script. A behavior script is exactly as it sounds, it is a high level script that tells the game how an object behaves. There are many behavior script tutorials online already so if you don't have a basic understanding of them you should read/watch them. I will not go over them in depth but just explain them as I use certain commands. For now we are only interested in this one.

08 00 00 00

0c 00 00 00 xx xx xx xx

09 00 00 00

Basically this executes a function every frame. The x’s represent the ram address of the function you want to execute. There's generally no restriction on where this is but most of memory is managed by the game where specific regions have specific purposes. The space in the rom from 0x1202000 to 0x1210000 is the easiest place to put code that will not overwrite anything else. This will load into 0x80400000 plus your offset from 0x1200000.

Now onto the actual code itself. To begin with, you have to access the object struct. The game stores a pointer to the beginning of the object struct at the address 0x80331160. If you are using a function executed using the 0c behavior script cmd shown above this is the best way to access the object struct.

Asm can only fit two bytes of an immediate at a time, so loading this value will take two instructions. You should almost always use the following code to load this value.

* LUI T0, 0x8036 // load upper immediate value of 0x8036 into the variable T0
* LW T0, 0x1160 (T0) //load the word stored at the offset 0x1160 from the value at T0

As you can see, first we put an upper value into T0. This means the first 2 of four bytes in a word get put into T0. You can represent this by 0x8036 0000. Next we load the value at an offset of 0x1160 from T0. This means you go to the value of T0, offset by 0x1160 then load the word there. This can be written mathematically with this.

* T0 = value at [ T0 (0x8036 0000) + 0x0000 1160]

This means in memory we will load the value stored at 0x8036 1160, which is the current object.

Now with this we can edit the struct. If you remember earlier, I mentioned that the struct is made up of variables with a value type and offset. To read a variable, you simply load the value offset from the current object which you have loaded above. Here is an example on how to load the y rotation.

First load current object:

* LUI T0, 0x8036
* LW T0, 0x1160 (T0)

Next load y rotation from T0

* LW T1, 0xD4 (T0)

0xD4 is the variable that holds the y rotation of the object. If I wanted to edit the rotation now all you would have to do is edit T1 and then store T1 back into the object struct. Here is an example:

//continuing from above code

* ADDIU T1,T1, 0x100
* SW T1, 0xD4 (T0)

So I add 0x100 to T1 then store it back into 0xD4. This will increase the rotation by 0x100. This is the general method used for all object editing, just use the specific offset for the specific variable you want to change.

### More on asm

If you want to know more on how to use asm you can reference this data sheet:

[asm short reference](http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html)  
[Full Reference (has some more complex cmds not covered here)](https://hack64.net/wiki/doku.php?id=r4300)

I will explain some nuances of n64 and asm code here.

First your variables are all temporary. That means you should not expect anything you load to stay in memory unless you store it somewhere. Your temporary variables have registers from T0 to T9. So any value you want to load should go in these first. Functions generally take a0 to a3 as arguments and return v0 and v1 with v0 being the much more common one. There are also S0 to S7 which are saved registers (only S0 and S1 are ever used in game) and there is AT which seems to just be a temporary value (most often used for loading a value used for arithmetic or unit conversion). Some registers are reserved and should not be used, these are FP, GP, K0 and K1, which are the frame pointer, global pointer and kernel values. The last register is the stack pointer (SP), which I will go over in the next tutorial. You can just use these however you want without worrying about them usually. My advice is to use the T registers for normal use and then use S0-S7 for a variable you don't want overwritten.

You can use any register for any data you want in your own function but later when you learn about game functions you should keep this in mind.

Next you should know when to use different variations of *ADD/SUB* cmds. Put an *I* at the end when you're using an immediate. An immediate is just a number. So above when I added 0x100 to T1 I used *ADDIU* because I added an immediate instead of two variables. The *U* at the end is for unsigned. This allows overflow. If you overflow using an *ADD* or *ADDI* cmd you generate an exception which is not good.

Cmds generally have a format of:

*cmd* **result**, **target**, immediate/argument

*Cmd* is what you're going to do with the registers, or basically the operation, in terms of math this is just add, subtract, multiply. It can also be bitwise logic. OR, XOR, AND and bit shifts. I will explain how these work as I use them in examples. **Result** is the register the result of the operation you do gets stored to. **Target** is generally the register your operation acts on, though the value inside **target** will not change. For add this means register that gets added to. Sub means that register gets subtracted from etc. Immediate/argument is the number/register that you operate on the target with. This will be the amount you add/subtract from the target for example.

Next is on words, halfs and bytes. A word is just a term for four bytes. When a cmd says *LW* (load word), it loads 4 bytes starting at the address. Half is two bytes and a byte is one byte. Any of the load and store cmds can be changed for these values. *LW* can also be *LH*, load half and *LB* can be load byte. They will load that many bytes starting from the offset you give it. If you use the smaller loading cmds then you have the option to load either signed or unsigned.

In N64 code, any number above 7fff ffff is considered negative. If you load a number above this value, it is negative. This is applied to offsets, adding and loading numbers. So for example, if you want to load 8033 B170, you would have to account for the negativeness of the second half. This means when you add B17c to 8033, it actually subtracts one, because it's considered negative. To properly load the value you should offset by one. Ex.

* LUI T0, 0x8034
* LW T0, 0xb170 (T0)

Here you can see that even though I load 8033 b170 I use 8034. This applies in general to all numbers. If you're loading a number and you want it to be unsigned, you can put a u at the end. For example, *LH* loads a half, and *LHU* loads a half unsigned. This means that loading 8000 will be negative in the first cmd but positive with the next.

If you are loading a half or a byte, the sign still remains the same. This means if the uppermost bit you load is 1, it is negative. For a half, this means above 7fff is negative and for a byte, above 7f is negative.

### Examples

These will all be under the assumption that your code is running every frame.

Ex. 1: rotating an object

* LUI T0, 0x8036
* LW T0, 0x1160 (T0)
* LW T1, 0xD4 (T0)
* ADDIU T2,T1,0x50
* SW T2, 0xD4 (T0)
* JR RA
* NOP

This code first loads the current object into T0. Then it loads the y rotation from 0xD4 inside the object struct. Next it adds 0x50 and puts that value into T2. Finally it stores T2 into 0xD4 of the object struct. To end the code it uses the *JR RA* cmd. This returns to where the function was called from. You don't have to understand that just yet. Just know that you always have to end a function in *JR RA* and put a *NOP* (no operation) afterwards.

The *NOP* is there because jumping in n64 code has a delay slot. I will explain that in part 2 where conditionals will be explained.

Ex. 2: flickering a model

* LUI T0, 0x8036
* LW T0, 0x1160 (T0)
* LH T1, 0x2 (T0)
* XORI T2, T1, 0x2
* JR RA
* SH T2, 0x2 (T0)

First I load the current object. Next I load the graph flags. The graph flags determine how the object is displayed. It can either be visible, invisible or billboarded. This code uses the bitwise cmd *XORI* which basically means if the bits are equal, make it zero. If not then make it 1. This is good for a flicker effect since it will change the second bit between 0 and 1 every frame. You can flicker any other one and it will have a different effect, try it out yourself. I end the code with *JR RA* and then store the cmd in the next line. This works because the line after *JR RA* still executes before the code ends.

Ex. 3: sprinkler type of rotation

* LUI T0, 0x8036
* LW T0, 0x1160 (T0)
* LW T1, 0xD4 (T0)
* ADDIU T2,T1,0x50
* XOR T3, T2, T1
* ANDI T4, T3, 0x4000
* XOR T2, T4, T2
* JR RA
* SW T2, 0xD4 (T0)

As usual we load the object struct and our variable. Next we increase it by 0x50 and then using the *XOR*, we can find out which bits are different with this cmd. We filter those bits with *ANDI* to get a result that basically means t4 is 0x4000 when T2 is just hitting 0x4000.

Now the result is that T4 is 0x4000 when our rotation is 0x4000 and zero when it is not. Next we *XOR* our rotation with T4 and basically it will set T2 to zero if T2 is 0x4000 or it will do nothing. This means we will rotate the obj until 0x4000 and then it will go down to zero, like a sprinkler.

Ex. 4: back and forth rotation

* LUI T0, 0x8036
* LW T0, 0x1160 (T0)
* LW T5, 0x154 (T0)
* XORI T7,T5,0x80
* SLT AT, T7, T5
* LW T9, 0x14c (T0)
* XOR T9, T9, AT
* SW T9 0x14c (T0)
* ORI T8, R0, 0x200
* ORI T7, R0, 0x100
* MULTU T9, T8
* MFLO T8
* SUBU T7, T7, T8
* LW T1, 0xD4 (T0)
* ADDU T1, T1, T7
* JR RA
* SW T1 0xD4 (T0)

As usual I begin by loading the obj struct. Next I load the timer, 0x154. The timer goes up by 1 every frame and resets when the action changes. The action is 0x14c. So what I do is *XORI* time with 0x80 and compare that with the normal time. *SLT* means set on less than. This means *AT* will be 0 if T7 is greater than or equal to T5 and will be 1 when T7 is less than T5. As it stands, T7 equals T5 plus 0x80 until T5 hits 0x80. At that point T5 is 0x80 and T7 is zero. This means we have AT equal 1 after 0x80 frames.

Next I load the action and *XOR* it with AT. This means every 0x80 frames the action will Swap between 0 and 1. This works because changing the action will reset the timer.

Next I load 0x200 to t8 and 0x100 to T7. I multiply T9, the current action by t8 and subtract t8 from T7. What this means is when the action is 1, T7 will be negative 0x100. When the action is zero, T7 will be 0x100. Then I add T7 to the current rotation and store it like in previous examples. Even though *ADDU* has unsigned after it, adding a number that's negative will still overflow and subtract. This is actually how the processor handles subtraction. It just adds a negative. *SUB* is just an easier way to write it but the assembler will always compile it as adding a negative.

### Compiling

I recommend you compile asm using simple armips gui. Download it here:

[download](https://hack64.net/printthread.php?tid=4)

Using it is easy enough. Just write your code and save it as a text file. One last thing before you compile will be to determine a location for your code. You Can do this by prefacing it with:

* .orga 0x[rom location]

You need to know how to convert a ram address to a rom address to use asm effectively. For a sm64 rom hack using the importer patch (any hack made with sm64 editor or rom manager) I usually put my asm between 0x1200000 and 0x1210000 which gets converted to ram at 0x80400000 plus the offset from 0x1200000. For example, if you start at 0x1203000, then your ram address is 0x80403000.

Otherwise most ram for asm will be between 0x802450000 and 0x80330000.

To get a rom address from that, just subtract 0x802450000 from the ram value. An example is 0x8029edcc will be in rom at 0x59DCC.

To get a ram address from an empty spot in ram, just take your rom address and add 0x80245000. If it’s over 0x80330000 then its not loaded into asm so you should then look to using space in 0x1200000 to 0x1210000.

### If you know which section of ram your code will be in, then you can use ram addresses along with a ".headersize 0x[address]" directive to tell the compiler where to put it in the rom. If I were to put it in the 0x8040xxxx region, I would use a headersize of 0x80400000.

### Decompilation

When I first wrote this tutorial, the decompilation of sm64 was still in progress. As of now it has been released. This basically means all the source code is available and is easily (mostly, some is still really bad) readable. You can find the GitHub repo here:  
[GIT](https://github.com/n64decomp/sm64)

As it stands, there are not many tools for it, but I believe decomp will become the new standard very soon. This does not mean asm is obsolete, as it will still be very useful for debugging and for working on compiled roms, but it is no longer the only option. These tutorials will still give you a general understanding of how the game works and how the code should be written, the code will simply be a different language.