### Preface

I will go over how to use a pj64 debugger to test code, find code of interest in the base game and search memory for important values. You should read my other asm tutorials first. At least up to part 4, examples and explanations. This tutorial will go fine in tandem or after that one. Otherwise this will be confusing or only marginally useful. If you are just interested in making tweaks this will also help, but I recommend reading part 1 at the very least. This is made with pj64 debugger in mind. As far as I know this has the best compatibility and most features of any debugger but this should apply to other debuggers as well.

### The program

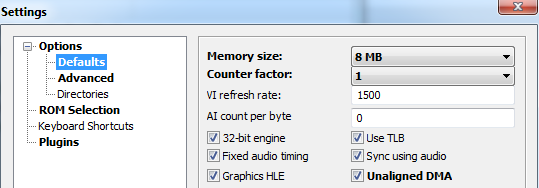
Nightly builds are the most up to date version and seem to be the only ones with rom hack compatibility (as of the last official version I downloaded). Last one I used had eeprom issues and probably still does but that is a minor issue. You can find them here:

[Nightly Builds](https://www.pj64-emu.com/nightly-builds)

It will try to get you to donate by spamming a msg on startup and may even just lock the msg on screen like some scummy ransomware. To avoid that use the password “thank you from project64” and the box will go away. Project64 is open source so if that doesn't work you can look at the source code and find the latest password. (2021 update: Download a nightly before Dec 2020 as the newer ones require a computer specific password which make them extra annoying to circumvent)

### Setting up the debugger

To use the debugger you're going to want to enable it inside the advanced options. Then you should get a tab at the top labeled debugger. As usual you should set memory to 8mb on startup. Everytime you recompile code you change the checksum and it will not remember the settings. You will have to start the game, change to 8mb, then reload (not soft reset). It seems to sometimes forget when you close and reopen as well so if you don't get past the title screen check your memory.



*My Settings*

Next you want to change the cpu core method to interpreter. This is just a more accurate method of emulation. Recompiler caches all the code in the checksum while interpreter runs the code there. This means if you overwrite something in the checksum area it will take effect with interpreter mode but not in recompiler. You will also have to write better code with interpreter. For example you will have to keep alignment in mind with interpreter but not recompiler (not exactly sure why tbh). Certain things that will crash on console will not crash on recompiler so if you want to increase compatibility you should use it. That said you should also check that it works on recompiler because that's the default emulator setting and most people playing your hack will spam you with issues that don't exist or just delete the rom.

Other settings are just up to personal preference. I usually use jabo for my plugins because those are the default and most common. It's good practice to develop with the most common plugins in mind. Of course, if you want your game to look good you should use gliden or ~~rice~~ angrylion (rice is similar to what wii VC uses so you may want to check on that but gliden and angrylion are the best to check now) plugin. Ideally you should test with all of them to make sure your hack works. Unfortunately this will limit certain effects because jabo sucks but that's just what you have to deal with if you develop with an emulator in mind. Some people will develop with the better plugins in mind, I don’t recommend it since it's worse for general users, but you will have a better looking game.

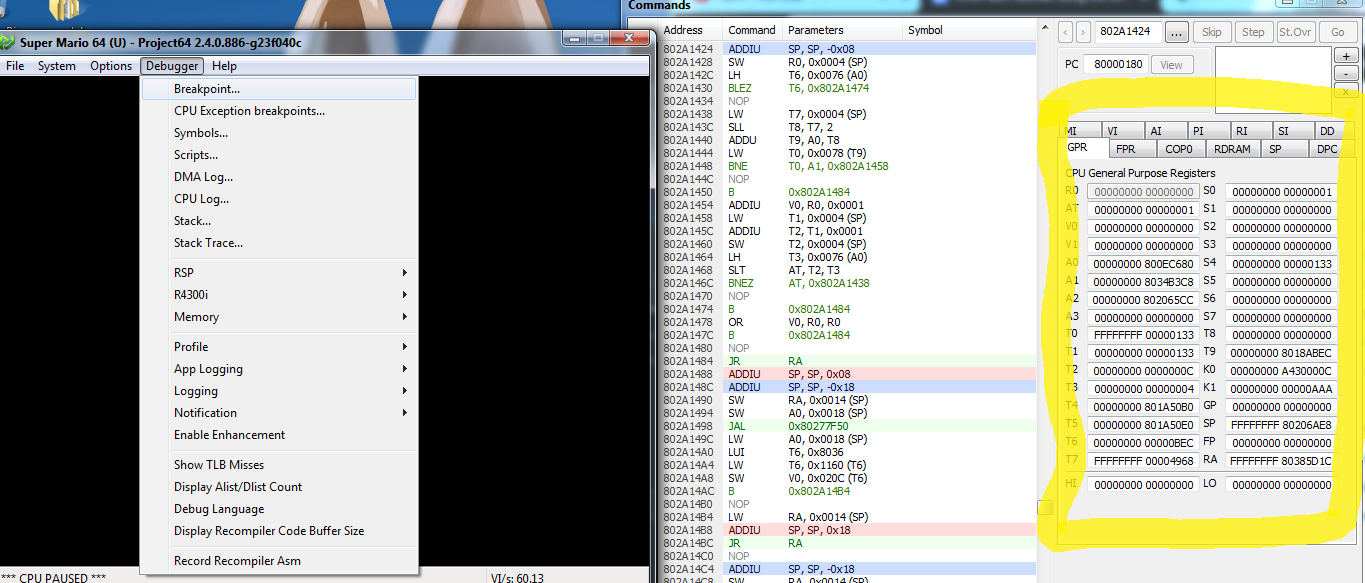
### Memory

The first things you should get used to are vmem addresses. Virtual memory just means RAM addresses. The reason its vmem instead of just RAM is that the N64 had several different types of memory you can view. There is audio, display and general memory. There is also a buffer for dma as well. I never use those and you probably will not as well. They may come up later if you're trying to do something advanced so keep them in mind, but most of the time you will be in general memory.

As you should know, general memory starts at 0x80000000 and should go for 8mb for a romhack (4mb if you are debugging vanilla). Memory is split up into segments, 0x80245000 is the start of the checksum, which is where most asm is. Important engine functions will be in 0x80370000 and forward. This is stuff like geo layout functions and processing object collision, but also OS functions as well (address prob isn't right). 0x80330000 is used for global variables (I don't think C has global/private variables but that's basically what they are), structs and other fun stuff. Earlier and later (ext memory is in hacks only) in memory is mostly used to store segments that are dma’d. So all your textures and models will be in those areas. There are other sections of memory completely unused and some used for important system effects that may only appear unused because the emulator doesn't use it. You should mostly stick to the areas I layed out above as they will be the bulk of your debugging.

### Using the debugger

Breakpoint should be the first option inside the debugger dropdown menu. This is an asm viewer, editor and console. A breakpoint basically means that when a condition is met, the game stops emulating and you can view and edit the console. You can then run the game line by line and view the console values as the code executes. This is immensefully useful for debugging.



If you're making a code and it doesn't work, this is the goto thing to test it with. Enter the RAM address of your function and then double click the line you want to test. Usually this is a branch, store or just the start. Then when the game executes the line it will break and you can see the values of all your registers on the right. General registers will be displayed as a default and floats will be in the tab labeled FPR. Then you can either hit “go” at the top to advance one frame, “step” to go forward one line or “skip” to skip the current line. Just going through the code and looking at the values should tell you a lot about what is happening.

You can also edit lines by right clicking them. There is an option to just *NOP* the line as well as editing a line. With editing, you have to type the line then hit enter. If nothing changes then your syntax was wrong. Combine this with save states to test out simple changes. For example you can *NOP* a *JAL* and see how the game reacts. You'll either learn what the *JAL* does or the game will crash. This also applies to branches as well. Editing code is a quick way to write code in without compiling and relaunching the game. It can also help test for simple errors. Common errors such as writing the wrong offset or the wrong register can be tested quickly and then you can see if there are other issues with your code. All edits are highlighted in pink and can be restored with right clicking them; if you save and load a state, they will not be able to be restored so keep track of your save states.

[Example of NOPs](https://imgur.com/7ZYEtb2)

Here I *NOP* branches to get a desired function from a pushable box. By changing where the *NOPs* are I can get a quick understanding of what each part of the code does very quickly. As mentioned earlier, edits are highlighted in pink.

Another useful thing to look at is memory. Usually you load and store things in your code. If you open the memory viewer, you can view the values you're editing. To start you should navigate to an address you're going to load or store. Then you set a breakpoint and view the value as your code executes. You can also type values into the memory viewer and see how the game changes. For example if you’re loading a value every frame, type a new value into the address and see how your code reacts. You should always pause or have a breakpoint set when typing a value in. You don't want to run the game with half the value typed in.

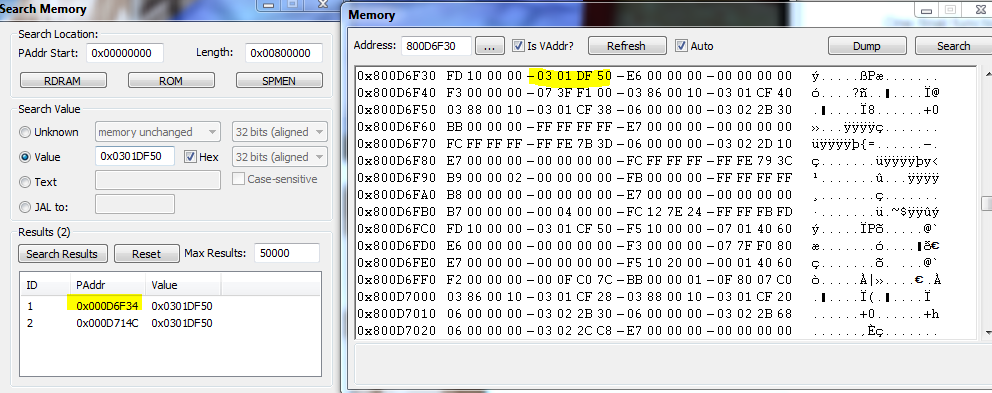
[Memory Typing Example](https://imgur.com/KInxUgv)  
Here I pause, type in a mario action of 0x00020449 into 0x8033b17c then unpause. The result is that mario gets set on fire, which is exactly what 0x00020449 does. Whatever value you put in immediately goes into effect which is useful for testing.

Another useful memory function is that you can right click a value in memory, and set a read or write breakpoint. This will jump the breakpoint window to the line that reads or writes to the specified value. You can use this to observe when structs are written to. For example, set a write breakpoint on 0x8033b1c4 to see how mario’s speed is calculated. You will find that there are several different locations depending on what mario is doing, but as you control mario and go line by line you will start to understand how it works. I recommend taking notes if you try to debug in game functions like this (with decomp now available you can also read the C code which makes it a lot easier). This is because values are stored and written to the stack a lot and many functions are called. It's very easy to get lost when there are no labels, which brings me to another debugger function, labels.

You can label certain values inside the debugger and monitor them. You can find it by going through debugger -> Symbols... This is useful for viewing struct values because it converts them to an easy to read format. It also allows you to view several different memory values at the same time without having to navigate with the tiny window back and forth. You can also double click them to bring them up in a memory view



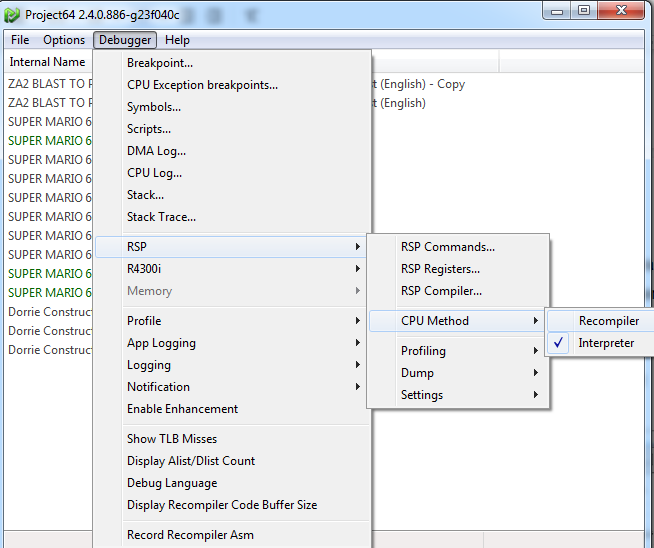
One final function to use is memory search. This does exactly what it says. Set a range (use vmem) and set a search type. Usually you want to be 32 bit aligned. This means you’re 2 byte aligned, which is the normal alignment restriction that you abide by anyway. If you double click the result it will bring up the memory viewer. I most often do this to look up textures in F3D. You can search the segmented address with 32 bit alignment and you will get every 0xFD texture assignment call used in the level.

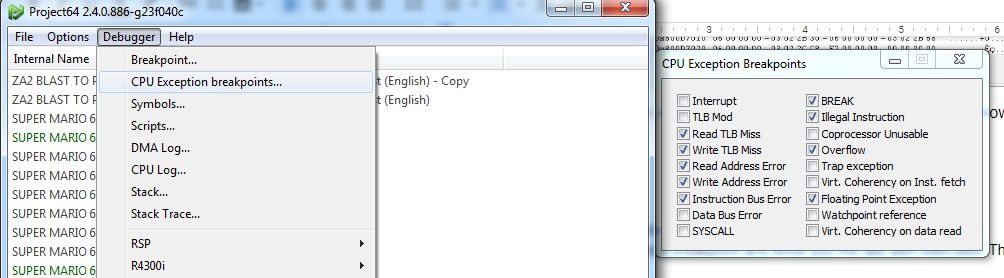


Here I can easily find the texture assignment for mario’s hat texture simply by knowing the address of that texture.

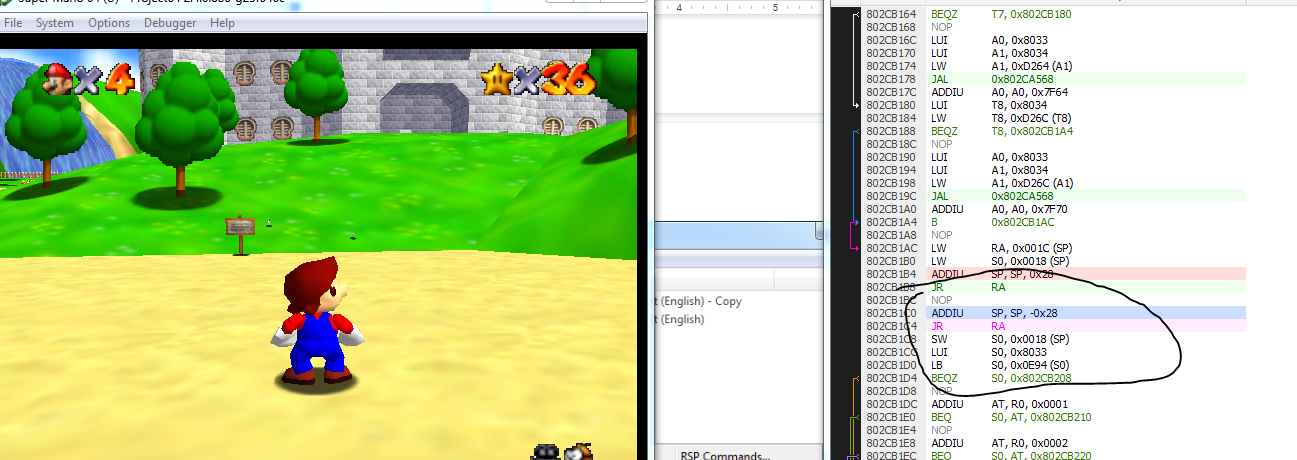
### Crash Detection

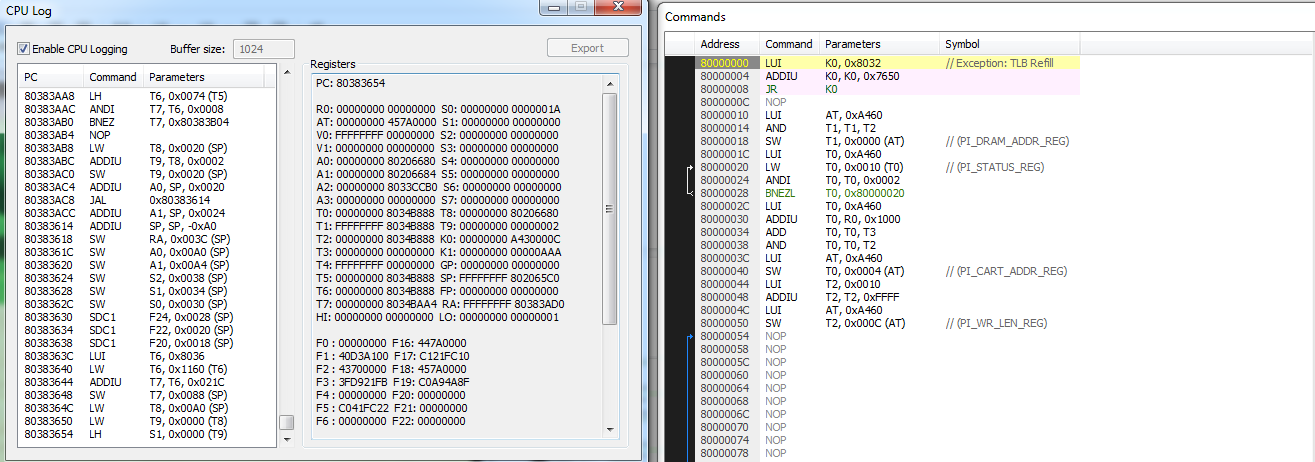
If you run the debugger on interpreter mode with cpu logging enabled, when you hit an exception, it will automatically make a breakpoint and show you the last asm executed. This is how you should set that up.



* Make sure that the cpu method is set to interpreter.
* Next use these CPU Exception breakpoints  
  
* Finally Enable CPU Log in the CPU Log option

Now whenever you generate an exception you will get a breakpoint and a CPU log of what happened. Here is an example:



Here you can see the error I have introduced. As you can see, the stack will be messed up and the exception will happen much later than the actual point where the error is.  
When I unpause the game, I get this breakpoint popup notifying me of a crash:  


You can see the crash comes from a bad load (T9 is 2 instead of a Vmem address), but you will have to go backwards in the log to find the true cause of the crash. I have set it up in a way where it is not too difficult to find it just by scrolling up the log, but sometimes the error will be larger than the 1024 buffer of the log. In this case, you will have to do advanced crash detection. Luckily as the user, you know the last place you changed. That should be the first place you check.

If you cannot detect the error from there, then the next step in detecting the crash is to go for the most likely errors. If that doesn’t work then you should attack the halfway point between your last change that interacts with the line of code, and the last executed line. The first condition may not be clear to those new to debugging so I will explain how you find what you should test.  
First you should try to isolate the crash to an object if you are editing it. Do this by just removing the object from your level. If you get a hit then it is easy to proceed from there, if not then you will have to check other changes. You will have to notice what section of code you are in. As you can see from the screenshot, we are in the 0x8038 region, which is engine code. This means that rather than code for an object, the problem is from something behind the scenes. This is usually code like behavior script parsing, geo layout parsing, collision, or level script parsing. Those would be the things I would be looking into to try and isolate the crash. Here are some tips to find each crash.

* Go to the start of the function that the crash came from and search it on the function list in hack64 or using the sm64 map file (generated from decomp, you can find a copy in the public decomp server by searching “sm64 map”).
* If you can find the function label, it will usually tell you what it is doing. Most have generic names like “GeoLayout00” or “LevelScript25”. These make it very easy to find what the error is.
* If you get a random function that makes no sense, then go back an additional function and try again. Many functions use jump tables, and if your index is outside of the array bounds it will jump to a random spot in the game. You should scroll up the cpu log and look for a JR T9 or other temp register which will indicate a jump table. These are common places for scripts to crash when you input the wrong values.

If you cannot find the crash from the low hanging fruit above, then you will have to go into the method mentioned earlier, which is to go to the halfway point. To do this, scroll up several functions in the cpu log and see if you can find a function you have changed. If you can’t, then just go towards something in the halfway point, otherwise find one between your function and the crash. What this method does is split the asm between your code and the engine code, and it allows you to see more clearly whether the crash is starting up or downwards of that point. What you want to do is set a breakpoint at that function, and see how it runs before the crash. Usually it will run several times before crashing if it is engine code, or just once. Try to debug a little what the function is doing in general, and what goes wrong when it crashes. Good use of savestates will be crucial.

So from here you will find that either the crash propagates from that function, or comes later. What that means is on the frame that the game crashes, that function will either act differently or the same as normal. If it acts differently than it is something before that function in the CPU log that is the issue. Otherwise it is something afterwards. Then you repeatedly split it half way until you can isolate the specific function that acts differently.

This method is usually only needed for the truly tricky bugs that are intermittent. The low hanging fruit I mentioned above will be the answer to most of your changes.