What This Video Assumes

- That you’ve met all the assumptions from Parts 1-4 of the Video Series, as well as watched Parts 1-4.
- That you have IDA Pro 7.6. IDA Pro 7.0 will not work for what we’re about to do. The tutorial could possibly work with IDA Pro 7.7, but there’s no guarantee of the programs working for the sake of this tutorial.
- That you’ve installed the Switch Loader plugin for IDA Pro 7.6. Even if you’ve installed it from the previous video, you will have to install a specific version for IDA Pro 7.6. Download it from here: https://github.com/pgarba/SwitchIDAProLoader
- Specifically, you’ll need to install the package from Switch64_76_123.rar. No other version of the plugin will work.
- At least Atmosphere 1.2.3 or above version installed, as that’s when debugging support was added. I recommend the latest version.

Video Contents

- Zero-Level Pointers
- Corrections to the Part 4 Video Series PowerPoint
- ARM32 (AArch32) and THUMB ASM (One Slide)
- Properly Setting Up Atmosphere
- Find the Address You Want to Debug
- Attaching IDA Pro 7.6 to the Switch
- Breakpoints (Read/Write/Execute) and Examples of Each
- Using IDA Pro 7.6’s Application Patcher
- Sections That Transfer Over to Other Consoles

Zero-Level Pointers

- In Part 3 of the Video Series, I went over how to create Pointer Codes. There’s another type of unusual pointer type I forgot to go over, as I only ever encountered it once. Here’s an example:

- [MAIN+759E0000] – This is a zero-level pointer. It has no plus or minus offset out to the side.

- Here’s how you handle a zero-level pointer:

```
580F0000 759E00000
640F0000 000000000 80000000
```

- A zero-level pointer lacks a Code Type 7 in its atmosphere code. A Code Type 5 is used to store the first bracket and a Code Type 6 is used to write to the address. Simple enough.

Corrections to the Part 4 Video Series PowerPoint

- In Part 4 of the Video Series, I told you that the MOV instruction could be used with the Sn and Dn registers. That’s not true. In order to move a custom value to a float/double address, two instructions are needed instead of one. This is because MOV only works with integer hexadecimal values and not floats/doubles. So, to get the value you want into a Sn or Dn register, you use MOV to put the hexadecimal value in float/double into an empty register from a Wn or Xn register. Then, you use FMOV to put the value from the Wn or Xn register into a Sn or Dn register. Unused registers typically have 0 in them. We’ll call this the donor register.
- Here’s how we’ll transfer a value of our choosing into each register:

For Sn Registers:

```
MOV Wn, #imm16 ; Sets Donor Register Wn to #imm16
FMOV Sn, Wn ; Sets Register Sn to Donor Register Wn
```

For Dn Registers:

```
MOV Xn, #imm16 ; Sets Donor Register Xn to #imm16
FMOV Dn, Xn ; Sets Register Dn to Donor Register Xn
```

Corrections to the Part 4 Video Series PowerPoint (EXs)

- For Sn Registers:
  1. MOV W10, #0x7F800000 ; Sets Donor Register W10 to 0x7F800000 (Infinity in Float)
  2. FMOV S9, W10 ; Sets Register S9 to Donor Register W10

- For Dn Registers:
  1. MOV X1, #0x7FF0000000000000 ; Sets Donor Register Xn to 7FF0000000000000 (Inf in Double)
  2. FMOV D8, X1 ; Sets Register D8 to Donor Register X1

- If you just want to set a float or double register to 0x0, only one instruction is needed. Simply do this:

  ```
  LDR Sn, #0 ; EX = LDR S11, #0
  LDR Dn, #0 ; EX = LDR D2, #0
  ```

- I also corrected some typos in the original slide for Part 4 of the Video Series and reuploaded it.

ARM32 (AArch32) and THUMB ASM

- Fortunately, most, if not all you learned in AArch64 will carry over to this section. I'm not going over specific ARM32 or THUMB instructions, as most will be the same.
- There's only a few notable differences between AArch64, ARM32, and THUMB.
- First off, ARM32 and THUMB are only present on 32-bit games. Games like Mario Kart 8 Deluxe are 32-bit. IDA Pro will alert you if a game uses ARM32 and THUMB and not the typical AArch64. If no alert is presented, the game is AArch64.
- Secondly, there are 12 GPRs we need to remember. They're r0-r12, which I'll notate as rN. If you see anything above r12, such as r13-r15, I suggest not writing to them, as they contain stuff used by the program. Also, avoid writing to r11 or r12 unless the instruction that handles your address uses that instruction. These registers store any value from 8-bit to 32-bit. Float and Double Registers are pretty much the same as before. Here's two examples of STR in ARM32 with the differently notated registers:

```
STR r10, [r9]
or
STR r10, [r9, #0xFC]
```

- Thirdly, the values reported by ARMConverter will be different than those used for AArch64, so use the values there instead. THUMB is an abbreviated form of ARM32 and uses 2-bytes instead of 4-byte instructions.
- Finally, all of the instructions available in AArch64 may not be available in ARM32 or THUMB. For instance, I didn't see STP or LTP. However, the notation of most instructions should be the same or similar as AArch64. With what you've learned, you should be able to look up documentation on ARM32 and THUMB and understand it.

Properly Setting Up Atmosphere

- Before we start debugging our Switch games, we’ll need to change a few settings in Atmosphere.
- Navigate to SD:\atmosphere\config_templates\ and copy the system_settings.ini file to the SD:\atmosphere\config directory.
- Place these three lines anywhere in your system_settings.ini file and then save it:

```
dmnt_cheats_enabled_by_default = u8!0x0
enable_standalone_gdbstub = u!0x1
enable_htc = u!0x0
```

- For the game you’re going to debug, go to SD:\atmosphere\contents\ directory and find the folder with the Title ID of the current game you’re playing. If there isn’t such a folder, you’re set. If there is a folder with that Title ID, back it up and delete it. We do this because only one thing can hook our game at once. It can either be Atmosphere’s dmnt_cheat module or it can be Atmosphere’s gdbstub module, as to which we use the gdbstub module to debug our game.
- Also, dump the MAIN file of your game with NXDumpTool. We’re going to need it.

Find the Address You Want to Debug

- First off, use Edizon SE to find your address of choice. It can be a MAIN, HEAP, or BASE address. It doesn’t matter if the address is static or dynamic.
- Then, on the search entry for the address or on the address’s bookmark, click in the center of the right stick to enter the Memory Editor.
- On the top ribbon, you’ll see a bracket with an address in it alongside its data type. For instance, you might see [646CC20B6A] u8. u8 is 8-bit data type and 646CC20B6A is the real address of the address you found with no connection to MAIN, HEAP, or BASE. Write down the address you see in brackets.
- Quit out of the memory editor by pressing ZL+B. Then, press the minus button to clear your search results. Click L to toggle bookmarks and delete any bookmarks you might have.
- Afterwards, press L again to get out of bookmarks. Now, press ZL+B to go back into the Homebrew Menu. Press B one more time. By doing all of this, we detach dmnt_cheat from the game so we can attach gdbstub.

Attaching IDA Pro 7.6 to the Switch

- Open IDA Pro 7.6 and click “Go Work on your own”.
- Go to the top ribbon and put the cursor over ‘Debugger.’ Hover over ‘Attach’, then click “Remote GDB Debugger.” You should get a pop-up saying “The current debugger backend does not provide memory information to IDA.” Check the “Don’t display this message again” box and then click ‘OK.’
- For Hostname, type your Nintendo Switch’s IP Address. For Port, type 22225.
- Now, click “Debug Options.” Under the logging tab, check everything. Under ‘Options’ check “Show debugger breakpoint instructions” and “Use hardware temporary breakpoints.”
- Now, click “Set specific options.” Change Configuration to “ARM64 (AArch64)”. Click ‘OK’ twice.
- Now, click ‘OK’ once you get back to the hostname entry. A box should pop up. Scroll all the way down and find the ID that says ‘Application.’ If you do not find the Application and the game is running, dmnt_cheat will need to be detached. Go back to Edizon SE and click ZL+B again and then click B again. After that, right-click the box and select ‘Refresh.’ Keep doing this until the application ID pops up.
- Select ‘Application’ and click OK. If the game’s 32-bit, it’ll tell you the processor can switch between ARM and THUMB modes. This is when you’ll know to use the other fields in ARMConverter. Otherwise, use ARM64 as always. The game will now pause.
- Right-click where it says memory then click “Jump to address.” Type in MEMORY:XXXXXXXXXX, where the Xs are the real memory address you found in Edizon SE.

Attaching IDA Pro 7.6 to the Switch (Cont)

- Before we move on to breakpoints, I want to point a few things out with the IDA Pro 7.6 GUI. Up in the near-top righthand corner, there's a box that says "General registers." This contains registers ?0-?30 or r0-r12. You will also notice SP and PC. SP is the Stack Pointer Register I previously talked about. PC is the Program Counter Register, which stores the current instruction we're on. Depending on what you're doing in IDA Pro 7.6 right now, you may even see PC in the lefthand corner if the game is currently stopped on a particular instruction. r13 is another way to say SP Register and r15 is another way to say PC Register. This is why I told you not to write to them: the processor needs those to see what instructions and functions to load.

- You'll also notice some letters right next to the box with the GPRs in them. It starts with N and Z, which are the processor flags for Negative and Zero I previously told you about, alongside some other letters (flags) I didn't mention. This area would show you what conditions are being met in the case of a compare condition like CMP, where a 0 is false for the condition being met and a 1 is true for the condition being met.

Breakpoints (Read/Write/Execute)

- When you've jumped to the address you found in Edizon SE, you'll want to set a breakpoint. It's useful to set breakpoints because we can figure out what instructions write to or read from an address, even without having labels or text in the MAIN file. This lets us get over that problem from Part 4 of the Video Series. Now you'll be able to make ASM codes for all games, not just ones with labeled MAIN files.
- Whenever a breakpoint is triggered, the game freezes and a particular instruction is displayed. Basically, the program counter gets frozen to the instruction that's acting on your address, allowing you to see the wanted instruction.
- To set a breakpoint, right-click the memory address, click "Add breakpoint" and a pop-up box will appear. Check the boxes 'Enabled' and 'Hardware' in the 'Settings' box. Check the boxes 'Break' and 'Trace' in the 'Actions' box.
- Now, go to the box that says "Hardware breakpoint mode." You'll see the options 'Read', 'Write', and 'Execute.' For memory addresses, you'll only use 'Read' and 'Write.'
- Finally, set the size to the bit-type of the address. If 8-bit, set size to 0x1. If 16-bit, set size to 0x2. If 32-bit, set size to 0x4. Now, press OK.
- Now, press the 'Play' button on IDA Pro to resume the game.
- If at any point you're hit with an error like SIGTRAP, write down the address of the instruction you set a breakpoint for, detach IDA Pro, then reattach it to the process to make sure IDA Pro functions properly.

Breakpoints (Write)

- A breakpoint set on Write will trigger whenever a value changes. After the breakpoint triggers, IDA Pro will report the memory address where the instruction is at. That is, whenever an instruction writes to the address. The instruction typically reported for the Write breakpoint is generally STR, as that writes from a register to an address. The game will pause.
- We'll mostly be messing with Write Breakpoints to be honest, as this will let us change what value is stored in the address.
- To test to see if this instruction is what is really changing the address's value, you can NOP it. Resume the game and try changing the value again. If the value stops changing after the NOP, that's the instruction you're looking for and you can stop setting breakpoints. If not, you'll need to set a breakpoint again on that address and continue NOP'ing instructions until you get to the right one. You can go over to the breakpoints tab and disable or enable the breakpoints whenever necessary to keep them from triggering unnecessarily while you eliminate each potential instruction.
- Keep in mind that multiple instructions might write to one address, and if that's the case, you may have to NOP them all at once to stop them from writing. Furthermore, if you wanted to do anything other than use a NOP, you'd still might have to alter every instruction that you found via breakpoints. This is typically the case for games that are ported from another console.
- To get the MAIN address for the instruction that controls your address, go down to the box right next to the button that says 'GDB.' Type in "get info" into the box and press enter. Go into "Modules" and find the module with .nss on the end. The beginning address for MAIN is the leftmost address. Now, take the memory value for the instruction you found in IDA Pro 7.6 and subtract it from the beginning of MAIN. There's your MAIN address for the instruction(s). Make it into a code with Atmosphere based on Part 4.
- You may also want to write a specific value into an address instead of just preventing it from changing. To do this, open up your copy of MAIN in a second window of IDA Pro 7.6 and then navigate to the address you found from doing the subtraction. Go above the STR instruction and see if you can find a BL or an LDR with the same register on the left as STR has. If you can find either, you can replace that instruction with a MOV to move the value of your choice into the register after STR executes. This is usually done via Atmosphere's Code Type 0 or by IDA Pro's application patcher. You may have to write multiple lines of Code Type 0 if you had to write to multiple instructions to get the desired effect from your code.

Breakpoints (Write - EXs)

- Resident Evil 5’s Ammo Address
- Borderlands TPS’s Vehicle Boost Address (Requires NOP’ing multiple instructions)

- We’ll use the application patcher to write our cheats in real time.
- We’ll also make Atmosphere Codes for both games based on the Write breakpoint results in order to give an infinite amount of ammo and vehicle boost.

Using IDA Pro 7.6’s Application Patcher

- After you find the instruction, whether it be by finding it via breakpoint or adding MAIN to an already known address, you can use IDA Pro to write to the instruction without using Atmosphere to test to see if your code would work.
- To do this, click on the address, go to the top ribbon and click ‘Edit’, highlight ‘Patch program’, then click ‘Change byte…’
- Write the desired instruction in ARMConverter. Make sure GDB/LLDB is grayed out and not blue. Then copy the bytes from ARMConverter based on if the instruction is AArch64, ARM32, or THUMB. Then, go and delete the first 4 bytes or 2 bytes, depending on what the instruction type is. Put spaces between each byte. Afterwards, press ‘OK.’ The instruction will change and you can then see how it affects the game. If you like the results, turn it into a code.
- You may notice we didn’t use GDB/LLDB here, even though I told you to use it with Atmosphere codes last video. This is because IDA Pro and Atmosphere read bytes differently, so make sure GDB/LLDB is blue whenever writing Atmosphere codes and make sure GDB/LLDB is gray whenever using IDA Pro to patch the program.

Breakpoints (Read)

- A breakpoint set on Read will trigger whenever an instruction from RAM accesses the address. After the breakpoint triggers, IDA Pro will report the memory address where the instruction is at. As the game is constantly reading from the address, it's likely a Read breakpoint will trigger immediately after unpausing the game. The instruction typically reported for the Read breakpoint is generally LDR, as that reads from the address and writes the results into a register. The game will pause when the breakpoint triggers.
- It could be useful to use a Read breakpoint to eventually prevent a read instruction from seeing a value. We could also load something into the register ourselves by replacing LDR with a MOV, allowing us to change another address down the line to something else. This could be useful to write to the visual address of something without writing to the actual address, as well as finding the visual address if you didn't know where it was.
- We'll rarely, if ever, be using these.
- Same process as in the Write breakpoint section if you'd like to locate it in MAIN or write to it via Code Type 0. You could also replace this instruction via IDA Pro's application patcher.

Breakpoints (Read - EX)

- Resident Evil 5’s Current Item Address

- We’ll use the application patcher to write our cheats in real time.
- We’ll also make an atmosphere code to change the appearance of the current item.

Breakpoints (Execute)

- The process for an Execute breakpoint is different than the Read or Write breakpoints. You must know where the instruction is located in memory to set an execute breakpoint. To find a known instruction based on its main address, type “get info” into the box where GDB is, find the .nss module, and then add the leftmost address to the ?s of [MAIN+0x????????????]. Then, click “Jump to address” and type MEMORY:???????????.
- You simply navigate to the instruction you’d like to observe using IDA Pro and then click “Add breakpoint” on it. If the instruction is in AArch64 or ARM32, set the Size to 0x4. If it’s THUMB, set the Size to 0x2. If the instruction is not written in hexadecimal, the box to set the options won’t even pop up and the proper breakpoint will be set automatically. Whenever the instruction executes, the breakpoint will trigger and the game will freeze.
- Execute breakpoints could be useful to see how your code works after you’ve set it via Atmosphere codetypes and then detached Edizon SE from the game. By seeing what each register currently is, you could see if your code is working properly and maybe even what’s causing a problem if there is one. This could also be used to find the address of the value the instruction writes to or reads from if you don’t currently know what it is.

Breakpoints (Execute - EX)

- Resident Evil 5’s Ammo Address
- Resident Evil 5’s Current Item Address

Sections That Transfer Over to Other Consoles

- ARM32 and THUMB are used on the Nintendo DS, as well as some 32-bit Nintendo Switch games. If you’ve learned AArch64, you can probably make ASM cheats for the Nintendo DS already.
- The whole section on breakpoints transfers over to pretty much every console with a debugger, minus using IDA Pro 7.6 for that purpose and the specific method of setting the breakpoints alongside their options.