What This Video Assumes

- The same assumptions from Part 1 of the Video Series are met.
- That you've watched Part 1 of the Video Series.
- You are able to connect Noexs to your Nintendo Switch. This is done by getting Noexs, a specific version of Java, and a sysmodule/kip. I'm not going over how to do this.
- You know how Code Type 0: Write to Memory works. Every code made from this video uses Code Type 0.

Video Contents

- Noexs
- String Data-Type
- Float and Double Data-Types
- How to Find Unknown-Value Addresses
- And examples and explanations of each

Noexs

- Noexs is a debugging software that allows you to make cheats for the Nintendo Switch.
- Noexs shows real addresses instead of addresses relative to HEAP and MAIN.
- Noexs is similar to Edizon but better for cheat creation in the fact it allows you to look at in-game memory, do unknown-value searches (we’ll get to that in this video), and do pointer searches (we’ll get to that in the next video).

How to Find Known-Value Addresses via Noexs

- Connect Noexs using your Nintendo Switch's Network IP Address. You will not be able to connect if you have a "cheats" folder for the game you're currently playing, so rename it for the sake of this video. Also, if you've opened Edizon before trying to connect with Noexs, Noexs will not connect to your game. You will need to restart your console.
- After connecting, close all of your Homebrew and open a game. Click "Refresh Pids". Go down to the bottom of the PID Selection tab and click the bottom-most number. The Title ID of your game should be shown to the right of that. Validate the Title ID if you're unsure. Now, click "Attach to Process." Your game should freeze. When it does, simply click the check box that says "Auto Resume" to fix that.
- You'll see a bunch of things pop up in the left-most tab. Those show memory regions of the game (labeled by "Name") and the real addresses of values in memory (labeled by "Address"). A memory region comprises the first part that says its name up until it says the next memory region's name. We'll be paying attention to the names "main" and "heap", as that represents the same memory regions we've been working with all along.
- Go to the top and click the "Search" tab. You'll see four drop-down boxes. There's Search Type, which lets you search from beginning to end of a certain area of memory. For instance, if we wanted to search "main", we'd set the Search Type to "Range" and then go back into the tools tab. You'd right click on the start of main and click "Search(Start)" and then you'd right-click the end of main and click "Search(End)." You could do this with any memory region or you could even search through only a part of main, or perhaps all of main and a part of heap. You can define that however you'd like. Unless you know ahead of time where your memory value is located, I'd suggest you set Search Type to "All (R/W)".
- Data-Type is simply what data-type you think you're looking for. We have from 8-Bit to 64-Bit. Set this before you set your search conditions.
- Finally, we have search conditions. For Known-values, we set the first tab to "Known" and the second tab to "Equals." Then, you set your current value in hexadecimal in the box to the right. Also, if you see a value in-game in decimal, you can input it into this box and click "Convert Decimal to Hex" and the tool will convert the number. Values must be in hexadecimal to be searched by Noexs.

How to Find Known-Value Addresses via Noexs (EX)

- Resident Evil 5 Ammo Address (HEAP 16-Bit Address)

How to Convert Noexs Addresses to Edizon Addresses

- Look at the address you found from your search. Go to tools and see what region of memory it falls under. For instance, let's say you get 0x21A0000000 as an Address.
- Look at the memory regions "main" and "heap" and see if they fall if your address falls under that memory region. Let's say for a minute that HEAP is from the range 0x20A0000000-0x2200000000. 0x21A0000000 is between 0x20A0000000-0x2200000000, so the address is a HEAP Address.
- Subtract the address you got from the beginning of the memory region.
- So, (0x21A0000000) – (0x20A0000000) = 0x0100000000. The Address would be represented by [HEAP+0x0100000000] in Edizon. The code that could be used to write to that address is as follows:

  0T100001 00000000 YYYYYYYY
- The code above would only work if the address was static. If you do the same math on another startup and find anything other than [HEAP+0x0100000000] from the calculation, your address is dynamic. I will not cover how to work with dynamic addresses until next video.
- Also, "0x" doesn't appear in addresses shown in Noexs. That is implied to be there by the program and was only used to make sure the math works properly here. Type your addresses in Noexs without the "0x" part.

How to Convert Edizon Addresses to Noexs Addresses

- If the address is static, you can track it back down in Noexs if you know that's the address. Let's say on next startup that HEAP is from the range 0x01E0000000-0x03E0000000. To find the address as shown in Noexs, add the value 0x0100000000 from last time to the beginning of HEAP.
- So, (0x01E0000000) + (0x0100000000) = 0x02E0000000. The address in Noexs is 0x02E0000000.

Noexs Incompatibility

- Fair warning: some games are incompatible or at the very least glitchy with Noexs. This is due to a bug in the tool. If that ends up being the case, I recommend either using Edizon for cheat searching. You can tell this if you do any search after the first one and you get an error message that reads:

  me.mdbell.noexs.core.ConnectionException: Result{mod=1, desc=106}

String Data-Type

- String is simply the way memory represents text.
- Each letter you see displayed in-game is in an address. You can find these addresses and write to them, naming things whatever you'd like.
- String values are going to be represented either by ASCII or Unicode (UTF-8).

String Data-Type (Cont)

- ASCII stands for American Standard Code for Information Interchange. The first 128 characters of ASCII are universal among most modern computer systems.
- The first 128 characters in ASCII, the universal part of ASCII, is represented in memory by hexadecimal values 0x00-0x7F.
- 0x00-0x1F and 0x7F are known as control characters. They enable special textual functions in memory. With the exception of 0x00, we won't worry about these.
- 0x20-0x7E are known as printable characters. They represent letters, numbers, and symbols, much like what you're using to read this slide.
- The last 128 characters in ASCII are known as the Extended Characters. They are represented in memory by values 0x80-0xFF. These extended characters can vary from system to system.
- A single character represented by ASCII can be found in an 8-Bit Address.

String Data-Type (Cont)

- Unicode serves to extend ASCII by adding more characters to it past 0xFF. These characters can be anything from letters from languages other than English, to mathematical symbols and wingdings. The first 128 characters, 0x00-0x7F, are the same as ASCII. 0x80-0xFF are the extended ASCII character set.
- Unicode has two character sets: the basic character set and the supplemental character set. Some computer systems designate string values to be able to use the basic character set only, while others allow for both the basic and supplemental character sets.
- The basic character set ranges from hexadecimal values 0x0000-0xFFFF. If the designated string uses only the basic character set, it will be in a 16-Bit Address.
- The supplemental character set ranges from hexadecimal values 0x010000-0x10FFFF. If a computer uses the supplemental character set, it also uses the basic character set. If the designated string uses this character set, it will be in a 32-Bit Address.
- As of right now, I haven’t seen a Nintendo Switch game use the supplemental character set. Your game will probably use either plain ASCII or the basic character set, which means a letter or symbol you see in-game is likely stored in either an 8-Bit Address or a 16-Bit Address.

How to Find String Addresses

- Essentially, the process is the same as the “How to Find Known-Value Addresses” slide in the last video.
- The “Known-Value” we’re searching for is the letter/symbol we see in-game. We get this known-value via a table.
- https://www.utf8-chartable.de/
- Place “code positions per page” on “128”, “display format for UTF-8 encoding” on “no display”, “Unicode character names” on “displayed”, and everything else on “not displayed”.
- The part that says “Unicode code point” in the table is the known-value in hexadecimal. Envision the U+ as an 0x for hexadecimal.
- For the purposes of the search, we’ll assume our address is ASCII (8-Bit String). To make this assumption hold, make sure the letters input in-game are regular letters (capital letters or not) and numbers 0-9.
- If you were to search for this on Edizon, you click “X” in the Search part of the Cheat Engine in order to switch it to Hexadecimal view, then type the hexadecimal characters as displayed by the table.
- Change the letter to another letter, search it’s known-value based on the table, and repeat this until you’ve narrowed yourself down to the appropriate address.
- Your value could potentially be Unicode and not ASCII. If there’s a 0x00 byte before the next character on each, or you convert Code Type 0 to work with 16-Bit and write a Unicode value and a character past 0xFF appears on the table, your value is actually a 16-Bit Unicode Address. If it’s ASCII and not Unicode, you might notice that the value you wrote in Unicode doesn’t appear in-game. You might also notice the letter you wanting to write to and the one next to it was written to something else.

How to Find String Addresses (EX)

- Super Smash Bros. Ultimate First Character of Name Address (Unicode)
- This address will be found in Edizon and then looked at on Noexs. This is because SSBU is incompatible with Noexs’s search function, but not with it’s memory viewer. I’ll use the memory viewer to show you what strings look like in memory.

How to Properly Write to String Addresses

- If I were working with ASCII, each byte/address represents another character. K# represents a character at each place. For instance, in the text “Hi!”, K1 is H, K2 is i, and K3 is !.
- Let’s assume the name address starts at 0x000000F000, relative to some region in memory. I could write 8 characters like this:
  01M00000 0000F000 000000K1
  01M00000 0000F001 000000K2
  01M00000 0000F002 000000K3
  01M00000 0000F003 000000K4
  01M00000 0000F004 000000K5
  01M00000 0000F005 000000K6
  01M00000 0000F006 000000K7
  01M00000 0000F007 000000K8
  or this:
  02M00000 0000F000 0000K2K1
  02M00000 0000F002 0000K4K3
  02M00000 0000F004 0000K6K5
  02M00000 0000F006 0000K8K7
  or this:
  04M00000 0000F000 K4K3K2K1
  04M00000 0000F004 K8K7K6K5
  or this:
  08M00000 0000F000 K8K7K6K5 K4K3K2K1

How to Properly Write to String Addresses (Cont)

- Let’s say I want to write character 1 (K1) of the previous example to be A in ASCII. I could do it the following ways:

  01M00000 0000F000 00000041

  or this:

  02M00000 0000F000 0000K241

  or this:

  04M00000 0000F000 K4K3K241

  or this:

  08M00000 0000F000 K8K7K6K5 K4K3K241

- Considering we’re working with ASCII instead of Unicode, the “00” part of U+0041 gets cut off. U+0041 is treated like 0x41 in this case.

How to Properly Write to String Addresses (Cont)

- These codes all work because characters are read byte-by-byte in-game. The section with the 01 code type writes each byte in a different code, the section with the 02 code type writes two bytes per code, and so on Essentially, what these code types do is write K1 to 0xF000, K2 to 0xF001, K3 to 0xF002, K4 to 0xF003, K5 to 0xF004, K6 to 0xF005, K7 to 0xF006, and K8 to 0xF007. You can choose any of these ways to do so, but choose the way that has the least lines of code. The cheating software only reads so many lines of codes before it stops reading.
- You can determine which one of these is necessary based on how many characters your name can hold. If my name held six characters, using one 04 line and one 02 line would be the most efficient way to do it. If I had 10 characters, writing one 08 and one 02 line would be the most efficient way of doing things. If I had 8 characters, using one 08 line would be the most efficient way of doing things. If I had 3 characters, one 01 line and one 02 line would be the most efficient way of doing things. If I had 12 characters, one 08 line and one 04 line would be the most efficient way of doing things.
- Let’s say I wanted to write the name Shawn to a 10-character ASCII Address. Once again, assume the name address is 0x00000F000 relative to some memory region. This is how I’d do it:

  08M0000 0000F000 0000006E 77616853

  02M0000 0000F008 00000000

  0xF000 is written to “S”, 0xF001 is written to “h”, 0xF002 is written to “a”, 0xF003 is written to “w”, 0xF004 is written to “n”, and 0xF005-0xF009 are written to 0x00 (null/nothing). The last addresses are written to nothing because we didn’t want to write a name that was 10 characters long. 0x00 is used to fill remaining bytes you don’t use in case you write a name that isn’t exactly the maximum amount of characters. You don’t need to do this for names that are the maximal character length, and the more letters you use, the less you have to fill with nothing.

How to Properly Write to String Addresses (Cont)

- Let’s say we were working with a 16-Bit String Address (Unicode). Let’s also say we want to write 4 characters. Let’s assume the name address starts at 0x000000F000, relative to some memory region. Here’s how that could work.

  02M00000 0000F000 0000K1K1

  02M00000 0000F002 0000K2K2

  02M00000 0000F004 0000K3K3

  02M00000 0000F006 0000K4K4

  or this:

  04M00000 0000F000 K2K2K1K1

  04M00000 0000F004 K4K4K3K3

  or this:

  08M00000 0000F000 K4K4K3K3 K2K2K1K1
- In the case of Unicode values, input them exactly how the table shows them. Let’s say you wanted to write character one to be the letter A in Unicode, you can do it this way:

  02M00000 0000F000 00000041

  or this way:

  04M00000 0000F000 K2K20041

  or this way:

  08M00000 0000F000 K4K4K3K3 K2K20041
- Considering we’re working with Unicode instead of ASCII, the “00” part of U+0041 is left on. U+0041 is treated like 0x0041 in this case.

How to Properly Write to String Addresses (Cont)

- I’m going to explain how the last codes we talked about would work. Let’s use this code for example:

  08M00000 0000F000 K4K4K3K3 K2K2K1K1

- The first byte of K4 is written to 0xF007, the next written to 0xF006. 0xF007 and 0xF006 are read together in memory to produce a Unicode character. Byte 1 of K3 is written to 0xF005 and Byte 2 of K3 is written to 0xF004. Byte 1 of K2 is written to 0xF003 and Byte 2 of K2 is written to 0xF002. Finally, Byte 1 of K1 is written to 0xF001 and Byte 2 of K1 is written to 0xF000.
- Let’s say I wanted to write the name Shawn to a 10-character 16-Bit Unicode Address. Once again, assume the name address is 0x00000F000 relative to some memory region. This is how I’d do it:

  08M00000 0000F000 00770061 00680053

  08M00000 0000F008 00000000 0000006E

  04M00000 0000F010 00000000

  0xF000-0xF001 is written to “S” by writing 0x53 to 0xF000 and 0x00 to 0xF001, 0xF002-0xF003 is written to “h” by writing 0x68 to 0xF002 and 0x00 to 0xF003, 0xF004-0xF005 is written to “a” by writing 0x61 to 0xF004 and 0x00 to 0xF005, 0xF006-0xF007 is written to “w” by writing 0x77 to 0xF006 and 0x00 to 0xF007, 0xF008-0xF009 is written to “n” by writing 0x6E to 0xF008 and 0x00 to 0xF009, and 0xF00A-0xF00B, 0xF00C-0xF00D, 0xF00E-0xF00F, 0xF010-0xF011, and 0xF012-0xF013 are all written to null/nothing by writing 0x00 to both bytes of each unicode address, or 0x0000 to each 16-Bit Address. The last addresses are written to nothing because we didn’t want to write a name that was 10 characters long. 0x0000 is used to fill remaining bytes you don’t use in case you write a name that isn’t exactly the maximum amount of characters. You don’t need to do this for names that are the maximal character length, and the more letters you use, the less you have to fill with nothing.

How to Properly Write to String Addresses (EX)

- Let’s make a code to write to the Super Smash Bros. Ultimate In-Game Character Nickname Address. I’m going to write the name “Shawn”. Remember, this Address was Unicode.

Purpose of Writing to String Addresses

- To rename an item that you've downloaded or received from someone else (A save file, a weapon, a Pokemon, etc).
- To bypass in-game word filters.
- To use special characters you otherwise couldn't in your name.
- To extend the length of a name you have in-game. Sometimes you're only allowed to type so many characters in-game, even though the name you're typing might have more memory allocated to it than you're allowed to use, which would give you a longer name. (Some games will read a pre-defined amount of memory into someone's name. Other games will continue to read strings until 0x00 or 0x0000 is written, which terminates reading in ASCII/Unicode. I don't recommend trying to extend a name, as there's a large chance you'll write over other memory values, causing glitches in-game).

Float and Double Data-Types

- Float and Double are data-types used to store decimal and fractional values in memory.
- Float, also known as Single Precision, stores values in a 32-Bit Address.
- Double, also known as Double Precision because it can store double the decimal places float can, stores values in a 64-Bit Address.
- The value stored in these notations aren't always the exact value of a decimal. The values returned are close enough equivalents when rounding for it to make sense to the computer as that same value. For instance, something written as 10.5412 might get represented as 10.541199999 in memory. It still rounds to 10.5412, though.
- Integer values, such as -1 or +10, and certain decimals, like +10.5, are represented exactly as they are in memory, without any rounding needed. Float and Double both used negative and positive numbers.
- The range of values stored by Float or Double addresses is much greater than what can be stored in typical 64-Bit Addresses.. A 32-Bit Float Address can hold maximal values MUCH bigger than 18446744073709551615 and minimal values much lower than -18446744073709551615. They can also approximate irrational numbers like $\sqrt{2}$ and π.

Float and Double Data-Types (EX)

- The value 10.5 is stored in each data-type as:

  Float: 0x41280000 (actual memory value is 10.5)

  Double: 0x4025000000000000 (actual memory value is 10.5)
- The value 10.5412 is stored in each data-type as:

  Float: 0x4128A8C1 (actual memory value is 10.541199684143066406 25)

  Double: 0x402515182A9930BE (actual memory value is 10.541199999999999903366187936626374721527099609375, which is much closer than what float got. It has double the precision that float has.)
- The value -18446744073709551616 is stored in each data-type as:

  Float: 0xDF800000 (actual memory value is -18446744073709551616)

  Double: 0xC3F0000000000000 (actual memory value is -18446744073709551616)
- The value π is stored in each data-type as:

  Float = 0x40490FDB Double = 0x400921FB54442D18

  Represented as 3.1415927410125732421875 in Float and 3.14159265358979311599796346854418516159057617187 5 in Double
- These results were gotten from this calculator: https://baseconvert.com/ieee-754-floating-point
- To get hexadecimal results, type the number you want in Float/Double in the top-most decimal box.
- To get decimal results, type the hexadecimal number in Float or Double, depending on which you're working with in the hex box. You will then be given a decimal number in the decimal (exact) box for whichever system you're working with.

Special Values in Float and Double

- 0 and -0

  0: Float = 0x00000000 Double = 0x0000000000000000

  -0: Float = 0x80000000 Double = 0x8000000000000000

- ∞ and -∞

  ∞: Float = 0x7F800000 Double = 0x7FF0000000000000

  -∞: Float = 0xFF800000 Double = 0xFFF0000000000000

- NaN

  Literally means Not a Number. Results either from memory errors, improperly converted values, invalid math calculations, or in special cases for debugging/setting illegal operations. 0x7FFFFFFF is an example of an error and 0x7FC00000 is an example of a special operation.

  You can write in-game memory values to any of these. With the exception of 0, I wouldn't toy with these too much. I suppose you could set a health value to be ∞ for yourself and -∞ for your enemies, but you may or may not get glitches from that.

- I'm going to include how to find Float and Double Addresses in the Unknown-Value section of this PowerPoint, as that's the only efficient method I've seen of finding and writing to these values. Every other method I could come up with was sub-par.

How to Properly Write to Float and Double Addresses

- Here’s the format to write to a Float Address:

  04M000AA AAAAAAAA VVVVVVVV
- Here’s the format to write to a Double Address:

  08M000AA AAAAAAAA VVVVVVVV VVVVVVVV
- V is the hexadecimal value found from the Float and Double calculator.
- Let's assume our address is 0x000000F000, relative to some memory region. Let's also assume we want to write 10.5 to address 0x000000F000.
- 10.5 is 0x41280000 in Float and is 0x4025000000000000 in Double.
- If the address is a Float Address (32-Bit), this is how we'd write 10.5 to it:

  04M00000 0000F000 41280000
- If the address is a Double Address (64-Bit), this is how we'd write 10.5 to it:

  08M00000 0000F000 40250000 00000000

Unknown-Value Addresses

- This section will help us find values that we otherwise cannot see represented as a number in-game, as well as float and double addresses, along with addresses we didn’t seem to be able to find with the last sections of this series.
- There still might be a few things this cannot find, like in-game code/ARM. I don’t really know how to work with that, so I cannot possible cover that. If someone would like to show me how, get in contact with me through my Discord:

  (Reclaimer Shawn#4926). I check it daily.

How to Find Unknown-Value Addresses

- As of right now, they can only be found via Noexs. This is why I showed you how to use Noexs earlier.
- First off, set the Search Condition to “Unknown.”
- Unknown-Value addresses can be any data-type. To make narrowing down our address easier, we’ll first assume our search will be a 32-Bit search, so set the “Data-Type” drop-down box to 32-Bit. Afterwards, click “Search.”
- After it gets done dumping your memory, you have a few potential strategies:

  If you see an indication of a value going up or down, you could set the first drop-down box in the search condition area to “Previous” and then set the drop-down box under it to “Greater Than” or “Less Than”. You could also set it to “Greater Than or Equal To” or “Less Than or Equal To” , but if you have an indicator the value has changed, I’d go with a “Greater Than” or “Less Than” search. Make the value change repeated and keep doing these searches until you find a single address or just a few and then test each address out individually to see if it’s what you wanted to write to. Some values that appear to go up, like health bars, could actually be going down. If you can’t find your address, search again but reverse what is “Greater Than” and “Less Than.”

  If you see an indication your value changes, but can’t see it go up or down, you could set the first drop-down box in the search condition area to “Previous” and then set the drop-down box under it to “Not Equal” whenever your value changes. If you find that a lot of values change alongside it that aren’t what you’re looking for, you could try not changing your value, letting everything else change, and then setting the second drop-down box to “Equals.”
- You might find your address but notice it’s a different data-type. Test it to see if it is in Noexs, similar to how we did it before. If you don’t find your address change the “Data-Type” to 16-Bit, then 8-Bit if that doesn’t work, and then finally 64-Bit if that doesn’t work.
- Unknown-Value searches take forever, so have a lot of patience with this. Fair warning: you might find your address but it can still be dynamic. If it’s dynamic, you’ll need the next video to make your code. More likely than not, the address is dynamic.
- Also, make sure you have a lot of space on your computer. Each search can take up several Gigabytes at a time, so pay attention to the amount of space left on your hard drive. The dumps Noexs creates are in the temp folder in the same directory as this program. They are labeled as .dmp. Results lists are labeled as .addrs. Delete them after you’ve found your address and close out of Noexs. You might need to keep the first dump for later if you find your address is dynamic (the first one is the least recent dump).

How to Find Unknown-Value Addresses (EX)

- Resident Evil 5 Health Address (16-Bit HEAP Address)
- Resident Evil 5 Mercenaries Timer Address (Float HEAP Address)

Sections That Carry Over to Other Consoles

- String Data-Type (May use ASCII or Unicode depending on the system's data-type.)
- Float and Double Data-Types (Not all systems use these, but most modern day games and computers do. Also, some systems may attempt to use something like Half-Precision, a type of precision that has half the precision of Float. Less decimal places are available and smaller numbers are used. I've never seen Half-Precision in anything before, so this is unlikely. Some systems might also use Float, but not Double. Some might use Float and Double, and/or possibly even Half-Precision at the same time.
- Unknown-Value Searches (Most cheating softwares allow for unknown-value searches, so you can use some of the same techniques used in this video for other systems.)