What This Video Assumes

- The same assumptions from Part 1 and Part 2 of the Video Series are met.
- That you've watched Part 1 and Part 2 of the Video Series.
- That you have Edizon SE and a Noexs sysmodule that is compatible with Noexs and Edizon SE working simultaneously. I'm not going over the process of getting these things.
- That you have PointerSearcherSE

Video Contents

- What Is a Pointer Code
- Format of a Pointer
- Meaning of the Pointer Format
- Pointer Code “Skeleton”
- Code Types (Code Type 5, 7, and 6)
- How to Make Pointer Codes with Noexs and Edizon SE
- Code Type 8
- Code Type 2
- How to Make Pointer Codes with PointerSearcherSE
- And examples and explanations of each

What Is a Pointer Code

- The purpose of a pointer code is to be able to write to a value that changes its location in memory.
- For instance, the value for money might move from HEAP + 0x00F0000000 at one start up of the game to HEAP + 0x00C0000000 at the next start up of the game.
- This change might even occur every level, after a certain interval of time, or at random.
- Values located in BASE + 0x?????????? can be written to, unlike we were unable to do in Part 1 and Part 2 of the series.

What Is a Pointer Code

- Pointer codes work by finding a place in memory that's static (doesn't move) and then using that as a starting point to track down whatever address you're trying to find.
- Areas of memory that are static include all of MAIN and what I call Static HEAP. There isn't a particular region of HEAP that is static as far as I know, but based on the game, some addresses in HEAP don't change their locations.
- Pointer codes follow and locate an address from the following points:
  MAIN -> Dynamic HEAP
  MAIN -> BASE
  Static HEAP -> Dynamic HEAP
  Static HEAP -> BASE
- I will explain why knowing this is important later on in the video.

Format of a Pointer

- The basic format of a pointer is as follows:
  [XXXX + 0x??????????] +/- YYYYYYYY
  or
  [[XXXX + 0x??????????] + YYYYYYYY]
  or even
  [XXXX + 0x??????????] + YYYYYYYY] +/- ZZZZZZZZ
- Where the following values are:
  XXXX = MAIN or HEAP
  0x?????????? = Value Relative to MAIN or HEAP
  +/- Means Either Addition or Subtraction
  YYYYYYYY = Some Value in Hexadecimal
  ZZZZZZZZ = Some Other Value in Hexadecimal
- It is important to know the format of a pointer, as that determines how to format our pointer code.

Format of a Pointer (EX)

- Some Valid Examples of Pointers Are as Follows:
  [MAIN + 0x00759E0000] + 98C
  [HEAP + 0x00759E0000] + 98C
  [[MAIN + 0x00759E0000] + 98C]
  [[MAIN + 0x00759E0000] - 98C
  [[[MAIN + 0x0049800000] + 420] + 68]
  [[MAIN + 0x0049800000] + 420] + 68
  [[HEAP + 0x0042000000] + 690] + 44

- A pointer need not have the above numbers, these are just fabricated examples. You will get pointers in a similar format as above.
- Despite the fact that pointers like [MAIN + 0x00759E0000] + 98C and [[MAIN + 0x00759E0000] + 98C] might look the same, they probably “point” to two different addresses. There IS a difference.

Format of a Pointer (EX)

- Some Invalid Examples of Pointers Are as Follows:
  [BASE + 0x00759E0000] + 98C (Reason 1)
  [MAIN - 0x00759E0000] + 98C (Reason 2)
  [HEAP - 0x00759E0000] + 98C (Reason 2)
  [[HEAP + 0x00759E0000] - 98C] (Reason 3)
- These might look like legitimate pointers, but they're not.
- A pointer cannot start in BASE. The end result the pointer code "points" to can end in BASE, resulting in a write to a BASE address, but BASE is never the beginning point. This is because BASE is dynamic and always changes. (Reason 1).
- A pointer cannot have a subtraction from a memory region. The pointer [MAIN + 0x00759E0000] + 98C would be valid but [MAIN - 0x00759E0000] + 98C cannot be valid. This is because if subtraction occurs, it must necessarily be outside of the region specified by the pointer. If addition occurs, the pointer could still be in the range of the specified memory region. (Reason 2).
- A pointer cannot have a subtraction in brackets (Reason 3).
- For example, if MAIN is equal to 0x19E0000000 and goes from 0x19E0000000 to 0x1A60000000, the address [MAIN + 0x00759E0000] is equal to 0x1A559E0000,which is between 0x19E0000000 and 0x1A60000000, meaning the address is in MAIN. However, [MAIN - 0x00759E0000] is equal to 0x196A620000, which is not between 0x19E0000000 to 0x1A60000000, which means the address is not in MAIN.

Meaning of the Pointer Format

- To get an understanding of what the pointer format means, as well as the subtle differences seen in each pointer, let's compare two pointers:
  [MAIN + 0x00759E0000] + 98C
  [[MAIN + 0x00759E0000] + 98C]

- The brackets, seen as [] or], mean to look at the value of the address in memory.
- Pretend MAIN is equal to 0x19E0000000. Given the calculation MAIN + 0x00759E0000 is equal to 0x1A559E0000. Let's say the value stored at 0x1A559E0000 is 0x3567E00000. What [MAIN + 0x00759E0000] means then is that the value contained by the address MAIN + 0x00759E0000, which is address 0x1A559E0000, is looked at. So, the value 0x3567E00000 is stored to memory as our address.
- [MAIN + 0x00759E0000] + 98C then means the actual address we write to is 0x3567E00000 + 0x98C, which is 0x3567E0098C. When our code is finished, the address 0x3567E0098C is where we write our preferred value to.
- So, what does [[MAIN + 0x00759E0000] + 98C] mean? It means that the value contained by 0x3567E0098C is what is stored to memory as our address. Let's assume the value contained by 0x3567E0098C is 0x4000000000. That means that when our code is finished, address 0x4000000000 is where we write our preferred value to.
- If HEAP is from 0x3500000000 to 0x4100000000, then [MAIN + 0x00759E0000] + 98C actually points to the address [HEAP + 0x0067E00000] and [[MAIN + 0x00759E0000] + 98C] actually points to [HEAP + 0x0C00000000]. In this case, we went from MAIN -> Dynamic HEAP.

Meaning of the Pointer Format (Cont)

- A lot of the math I did might seem complex. Fortunately, I've never had to do that math whenever making a pointer code. That is just showing how pointers work from the point of view of the memory and the codes we'll make to search for these addresses and write to memory.
- The "Meaning of the Pointer Format" section then, in terms of making pointer codes, is fundamentally meaningless. I just wanted to show the logic behind what we're about to make.
- In the next section, I will use the examples given in the "Format of a Pointer (EX)" sections and turn them into workable codes.

Pointer Code “Skeleton”

- Below, I’m going to give the basic “skeleton” of a pointer code. That is, the basic code you’ll use to make all your pointer codes from here-on-out:

- This is the “skeleton” of a pointer code
  58MRP0XX XXXXXXXX
  780RS0YY YYYYYYYY
  6T0R0000 ZZZZZZZZ ZZZZZZZZ
- M = Address Type (0 = MAIN, 1 = HEAP)

•R = Register (Hexadecimal value from 0x0-0xF. Registers, in the context of our cheating software, are special addresses we can store things in to make things easier on ourselves. For pointer codes, using F will suffice.)

- P = Position Type (0 = First 5 Code Type, 1 = After First 5 Code Type)
- XXXXXXXXXX = Address Relative to MAIN, HEAP, or Brackets
- S = Arithmetic Type (0 = Addition, 1 = Subtraction)
- YYYYYYYYYY = Value to Add to Value in Register R

•T = Data-Type (1 for 8-bit, 2 for 16-bit, 4 for 32-bit, 8 for 64-bit. T represents the amount of bytes a data-type has.)

- ZZZZZZZZ ZZZZZZZZ = What to Write to Pointer Address

Code Type 6: Store Static Value to Register Memory Address

- Basically, Code Type 6 is Code Type 0 but for Pointer Addresses. We use Code Type 5 and Code Type 7 to calculate the address to write to and then we use Code Type 6 to write to the calculated pointer address. There are four basic variations of Code Type 6 based on the data-type.
- 8-Bit Write:
  610R0000 00000000 000000ZZ
- 16-Bit Write:
  620R0000 00000000 0000ZZZZ
- 32-Bit Write:
  640R0000 00000000 ZZZZZZZZ
- 64-Bit Write:
  680R0000 ZZZZZZZZ ZZZZZZZZ

Code Types 5, 7, and 6

- If I decide to do a video exclusively over code types, I'll go over Code Type 5 and Code Type 7 in more detail, as they can be used for things other than Pointer Addresses. Specifically, they can be used to store values into the atmosphere code handler and you can do further things with those values.
- Code Type 6 has been thoroughly explained already and Code Type 6 will never be covered again outside of this video over Pointers. I've never seen Code Type 6 used for anything other than writing to Pointer Addresses.
- Some coders use Register F for operations other than pointers. I use Register F for operations with pointers. This can bring about issues in your codes, as a code that moves a value into Register F would do it for all of the code handler, not just within the code. Furthermore, you might want to do this yourself and do other operations with pointers within a code. You can change the register used for a pointer address if necessary.

Code Types 5, 7, and 6 (Cont)

- Below is an example of attempting to write to two pointers within the same code. Unless you need to write to multiple pointers within one code, you usually don't need to change the register unless another codemaker's code conflicts with yours.
- This is the example:
  58MFP0XX XXXXXXXX
  780FS0YY YYYYYYYY
  6T0F0000 ZZZZZZZZ ZZZZZZZZ
  58MEP0XX XXXXXXXX
  780ES0YY YYYYYYYY
  6T0E0000 ZZZZZZZZ ZZZZZZZZ
- The above code stores the results of one pointer address into Register F and stores the results of another pointer address into Register E. This doesn't entirely make sense yet. It will after the next few slides.

Code Types 5, 7, and 6 (EX)

- Let's use the "skeleton" given previously to convert pointers into codes we can use to write to our desired addresses. I'll use all of the previous pointers and convert them into codes. Furthermore, I'll explain what each code does:
- Pointer 1 = [MAIN + 0x00759E0000] + 98C. We'll write the 32-Bit value 0x7FFFFFFF to the pointer address.
  580F0000 759E0000 <- Moves the Value of MAIN + 0x00759E0000 into Register F
  780F0000 0000098C <- Adds 0x98C to the Value in Register F. Gives Pointer Address [MAIN +0x00759E0000] + 98C in Register F.
  640F0000 00000000 7FFFFFFF <- Writes 0x7FFFFFFF (32-Bit) to Pointer Address [MAIN +0x00759E0000] + 98C in Register F.
- For those of you used to x86, the above code can be represented as the following in order:
- MOV r15, [MAIN + 0x00759E0000]
- ADD r15, 98C
- MOV [r15], 7FFFFFFF

Code Types 5, 7, and 6 (EX - Cont)

- Pointer 2 = [HEAP + 0x00759E0000] + 98C. We'll write the 8-Bit value 0x7F to the pointer address.

  581F0000 759E0000 <- Moves the Value of HEAP + 0x00759E0000 into Register F. Notice it's 581F0000 instead of 580F0000 for HEAP.

  780F0000 0000098C <- Adds 0x98C to the Value in Register F. Gives Pointer Address [MAIN + 0x00759E0000] + 98C in Register F. Notice the 7 Code Type is used for codes without brackets.

  610F0000 00000000 0000007F <- Writes 0x7F (8-Bit) to Pointer Address [MAIN + 0x00759E0000] + 98C in Register F.

- Pointer 3 = [[MAIN + 0x00759E0000] + 98C]. We'll write the 64-bit value 0x7FFFFFFFFFFFFFFF to the pointer address.

  580F0000 759E0000 <- Moves the Value of MAIN + 0x00759E0000 into Register F.

  580F1000 0000098C <- Moves the Value of [[MAIN + 0x00759E0000] + 98C] into Register F. This is to cover the last bracket. Notice the 5 code type is used for brackets. Also notice 580F1000 instead of 580F0000. This is because this is a Code Type 5 used after the first Code Type 5. All subsequent code type 5's are done like this.

  680F0000 7FFFFFFF FFFFFFFF <- Writes 0x7FFFFFFFFFFFFFFF (64-Bit) to Pointer Address [MAIN + 0x00759E0000] + 98C] in Register F.

- Pointer 4 = [MAIN + 0x00759E0000] - 98C. We'll write the 16-bit value 0x7FFF to the pointer address.

  580F0000 759E0000 <- Moves the Value of MAIN + 0x00759E0000 into Register F.

  780F1000 0000098C <- Subtracts 0x98C to the Value in Register F. Gives Pointer Address [MAIN + 0x00759E0000] - 98C in Register F. Notice the 7 Code Type is used for codes without brackets

  620F0000 00000000 00007FFF <- Writes 0x7FFF (16-Bit) to Pointer Address [MAIN + 0x00759E0000] + 98C] in Register F.

- For those of you used to x86, the above code can be represented as the following in order:

  MOV r15, [MAIN + 0x00759E0000]

  SUB r15, 98C

  MOV [r15], 7FFF

Code Types 5, 7, and 6 (EX - Cont)

- Pointer 5 = [[[MAIN + 0x0049800000] + 420] + 68]. We'll write the 8-Bit value 0x80 to the pointer address.

  580F0000 49800000 <- Moves the Value of MAIN + 0x0049800000 into Register F. Notice it's 581F0000 instead of 580F0000 for HEAP.

  580F1000 00000420 <- Moves the value of Pointer Address [MAIN + 0x0049800000] + 420] into Register F. Notice 580F1000 instead of 580F0000. This is because this is a Code Type 5 used after the first Code Type 5. All subsequent code type 5's are done like this.

  580F1000 00000068 <- Moves the value of Pointer Address [MAIN + 0x0049800000] + 420] + 68] into Register F. Notice 580F1000 instead of 580F0000. This is because this is a Code Type 5 used after the first Code Type 5. All subsequent code type 5's are done like this.

  610F0000 00000000 00000080 <- Writes 0x80 (8-Bit) to Pointer Address [MAIN + 0x0049800000] + 420] + 68] in Register F.

- For those of you used to x86, the above code can be represented as the following in order:

  MOV r15, [MAIN + 0x0049800000]
  MOV r15, [r15 + 420]
  MOV r15, [r15 + 68]
  MOV [r15], 80

Code Types 5, 7, and 6 (EX - Cont)

- Pointer 6 = [[MAIN + 0x0049800000] + 420] + 68. We'll write the 8-Bit value 0x80 to the pointer address.

  580F0000 49800000 <- Moves the Value of MAIN + 0x0049800000 into Register F. Notice it's 581F0000 instead of 580F0000 for HEAP.

  580F1000 00000420 <- Moves the value of Pointer Address [MAIN + 0x0049800000] + 420] into Register F. Notice 580F1000 instead of 580F0000. This is because this is a Code Type 5 used after the first Code Type 5. All subsequent code type 5's are done like this.

  780F0000 00000068 <- Adds 0x68 to the Value in Register F. Gives Pointer Address [MAIN + 0x0049800000] + 420] + 68 in Register F. Notice the 7 Code Type is used for codes without brackets.

  610F0000 00000000 00000080 <- Writes 0x80 (8-Bit) to Pointer Address [MAIN + 0x0049800000] + 420] + 68 in Register F.

- For those of you used to x86, the above code can be represented as the following in order:

  MOV r15, [MAIN + 0x0049800000]
  MOV r15, [r15 + 420]
  ADD r15, 68
  MOV [r15], 80

Code Types 5, 7, and 6 (EX - Cont)

- Pointer 7 = [[HEAP + 0x0042000000] + 690] + 44. We'll write the 16-Bit value 0x8000 to the pointer address.

  581F0000 42000000 <- Moves the Value of HEAP + 0x0042000000 into Register F. Notice it's 581F0000 instead of 580F0000 for HEAP.

  580F1000 00000690 <- Moves the value of Pointer Address [HEAP + 0x0042000000] + 690] into Register F. Notice 580F1000 instead of 580F0000. This is because this is a Code Type 5 used after the first Code Type 5. All subsequent Code Type 5's are done like this. Also notice that the second Code Type 5 is 580F1000 instead of 581F1000. Designating HEAP or MAIN is only necessary in the first Code Type 5.

  780F0000 00000044 <- Adds 0x44 to the Value in Register F. Gives Pointer Address [HEAP + 0x0042000000] + 690] + 44 in Register F. Notice the 7 Code Type is used for codes without brackets.

  620F0000 00000000 00008000 <- Writes 0x8000 (16-Bit) to Pointer Address [HEAP + 0x0042000000] + 690] + 44 in Register F.

- For those of you used to x86, the above code can be represented as the following in order:

  MOV r15, [HEAP + 0x0042000000]
  MOV r15, [r15 + 690]
  ADD r15, 44
  MOV [r15], 8000

How to Make Pointer Codes with Noexs and Edizon SE

- First, get your game started up and get to a point where your value is in use (for instance, if you're modifying ammo, only start this process whenever your ammo counter shows up in-game).
- Next, attach Noexs. Do a search of any kind. This allows Noexs to take a RAM Dump. Make note of the start of MAIN, the start of MAIN code_mutable, the end of MAIN code_mutable, the start of HEAP, and the end of HEAP. You can detach and exit out of Noexs after the search ends and you get a full RAM Dump.
- Use Edizon SE to find the current address of the value you're looking for. Edizon SE is used in place of Edizon, which was shown in Parts 1 and 2 of the video series. Use the same processes as last time to find the address. Edizon SE is significantly faster at finding addresses than Noexs and the old version of Edizon. After finding the address, write it down. In Edizon SE, write down the start and ending addresses of BASE.
- Open Noexs again. Go to the "Pointer Search" tab and click "Browse" next to "Dump File." Select the recently created RAM Dump.
- Set Main to the beginning of MAIN. Click the "Filter" checkbox and enter the start of MAIN code_mutable into "Min" and the end of MAIN code_mutable into "Max."
- As shown in the previous videos, take the start of the memory region you're working with and add it to the offset found in the address. Place this value into the "Address" box above the "Dump File." If you got [BASE + 0x0040000000], add the start of BASE to 0x0040000000 to get the address.
- Set "Max Depth" to 1 and set "Max Offset" to 0000F000. Click "Search." If that doesn't find anything, increase the "Max Offset" to 00020000 and repeat the same steps. As Max Offset increases, the time it takes to search for pointers increases. You can either continue to increase this to see if anything's there to your liking or set "Max Depth" to 2 and set "Max Offset" to 0000F000. If that still doesn't find anything, increase the "Max Offset" back to 00020000. This process finds a pointer that starts in MAIN.
- If the above procedure doesn't find anything, repeat the above procedure, but replace "Main" and "Min" with the start of HEAP. Replace "Max" with the end of HEAP. This procedure finds a pointer that starts in HEAP.
- After finding a list of pointers, click "Export List" on Noexs to export the found pointers. We're still not done, though.
- You may have only got about 3-10 pointers, in which case, you can stop here and attempt making all of them into codes and see which work in the game. However, considering some pointers only work for the entirety of a level and not the entirety of the game, I recommend always doing this process again on another level. That is, go to a new level, take a new dump, write down your memory regions, find the address you're looking for again, and "Export List" again. This also helps narrow down the amount of pointers you have, as only a couple will consistently work with your game. Furthermore, you can expect to get hundreds of pointers, so repeating this helps to narrow these pointers down. It also might be possible that no pointer exists that handles the value for every level in the game, so you may even have to make a pointer code for EVERY level in the game. I haven't had this happen yet, but it is a possibility.
- To narrow down these pointers, go to the website http://www.listdiff.com/compare-2-lists-difference-tool and put your first list of exported pointers into List A and the second list into List B. Click compare lists. The pointers that show up in A ∩ B are the pointers that are in both lists, and thus, the most likely to work. This often cuts down the list of pointers you're working with quite a bit.
- If there's still a lot of pointers, even after comparison, save the list seen in A ∩ B into a notepad file, take a 3rd dump, repeat the process again, and place the list from A ∩ B into List A and place the 3rd dump's list into List B. Click compare list. This should narrow down the pointers you have even further and the list contained in A ∩ B will be your most likely pointers. You may have to repeat this process SEVERAL times (like 10-20 even). This is a long and hard process.
- You'll know if you have to make a pointer for each level or not based on comparing the RAM Dump from one level to another. If you get zero things in A ∩ B, that means you'll likely have to make a pointer code for each level. In that case, you'd need two or more RAM Dumps for each level and you'll need to manually make a pointer for each using the process above. Hopefully, this doesn't happen to you. It hasn't happened to me, yet.
- After you get to a comfortable number of pointers to deal with, make pointer codes for each pointer you made. Try breaking the pointer codes in-game by selecting different characters, different levels, going between chapters, etc.
- Release the most stable pointer code that crashes the least. Sometimes, you might find multiple pointer codes that are just as stable as the rest. Either release them all just in case or release one of your choice. If you release one of your choice, and it suddenly doesn't work for people, retest your pointer codes and release another one of the pointer codes you didn't release.

How to Make Pointer Codes with Noexs and Edizon SE (EX)

- Item Storage Quantity Address in Pokemon Mystery Dungeon: Rescue Team DX (Pointer found in MAIN, points to Dynamic HEAP)
- Ammo Address in Resident Evil 5 (Pointer found in Static HEAP, points to Dynamic HEAP)
- Keep in mind that only the starting address matters for a pointer code (MAIN/Static HEAP). Basically, the same type of code would be made for a pointer that starts in MAIN and points to Dynamic HEAP or BASE. The same is true for a pointer that starts in Static HEAP: it'd be the same type of code whether or not it points to Dynamic HEAP or BASE. However, the codes made for a pointer that starts in MAIN would be different from a code made for a pointer that starts in HEAP. This was demonstrated in the "Code Types 5, 7, and 6" section.
- I will make working pointer codes for both addresses in the video as an example. In my experience, the lower the Max Offset and the lower the Max Depth, the more likely a pointer is to be valid. Your experience may vary.
- Using Noexs for the first search and Edizon SE for the second search avoids issues with Noexs crashing at the 2nd search as described in the previous video. That, and Edizon SE is faster than Noexs at searching anyways, and without the need for internet.

Code Types 8 and 2

- The purpose of using Code Type 8 and Code Type 2 is to make it so a code only activates whenever a certain button combination is pressed.
- This is useful because you may not always want your codes writing to the game constantly. You might want the code to only activate whenever you want it to. Maybe you want to play the game legitimately until certain moments of the game, when you'll decide to press the buttom combination.
- Also, using button activators can help get around the issue of certain codes/pointer codes crashing, as some pointers only load properly into memory whenever they're actively used in the game. As a result, writing the pointer before this can crash the game. By doing this, you can make the pointer code only write whenever you want it to (that is, whenever the address you're looking to write to is in use), getting around the crashing issue.

Code Type 8: Begin Keypress Conditional Block

- 8KKKKKKK
- Where K can equal the following values:
  0000001: A
  0000002: B
  0000004: X
  0000008: Y
  0000010: Left Stick Pressed
  0000020: Right Stick Pressed
  0000040: L
  0000080: R
  0000100: ZL
  0000200: ZR
  0000400: Plus
  0000800: Minus
  0001000: Left
  0002000: Up
  0004000: Right
  0008000: Down
  0010000: Left Stick Left
  0020000: Left Stick Up
  0040000: Left Stick Right
  0080000: Left Stick Down
  0100000: Right Stick Left
  0200000: Right Stick Up
  0400000: Right Stick Right
  0800000: Right Stick Down
  1000000: SL
  2000000: SR

Code Type 8: Begin Keypress Conditional Block (Cont)

- What does Code Type 8 do, exactly? It looks as if I just posted a bunch of random numbers and expected you to immediately get what it mean.
- Code Type 8 is a button activator code. By using Code Type 8, you can make it so your codes only activate whenever a button is pressed. How is this related to pointer codes? Well, pointers are in the game before they point to the address you want to write to. This means the game can crash if you use a pointer code before that address loads. To prevent this problem, we can make it so the pointer code only activates whenever we want it to (when the pointer points to the value we want to write to), preventing the problem of crashing. This can also be used for non-pointer codes, as well.
- Here's an example of setting Code Type 8 properly:
  80000080

- Recall that when K = 0000080, the value is equal to R. So, if you use the above Code Type 8, you'll be setting up your code to only activate whenever the R button is pressed on the console.
- You can also add buttons together to make it so a code only activates when two buttons are pressed at once. Recall that if you wanted to set Code Type 8 to trigger whenever L is pressed, K = 0000040. But what if we wanted to make it so our code activates whenever we press not R, not L, but L and R at the same time (L+R)?
- In order to combine buttons, you do a mathematical operation known as a "Bitwise OR." This is an operation that occurs with binary. You do NOT need to know how to do this yourself, just use a calculator. This online calculator works perfectly fine for this: https://www.rapidtables.com/calc/math/binary-calculator.html
- So, to do this for L and R, type 80 in the "First number" box and set the dropdown list to the right of the box to hex. Type 40 in the "Second number" box and set the dropdown list to the right of the box to hex. Then, set the "Operation" dropdown box to "or (|)". Click calculate. Look in the "Hex result" box for the OR'd value.
- 40 OR 80 = C0, so set K to 00000C0. The proper Code Type 8 to use for L+R is 800000C0.

Code Type 2: End Conditional Block

- 20000000
- Basically, all this Code Type does is end a code that begins with either Code Type 8 or Code Type 1. Code Type 1 will not be went over in this video, but Code Type 8 has already been covered. Any code that begins with Code Type 8 must be ended with Code Type 2.
- To properly use Code Type 8 and 2 together, here's the basic "skeleton" to use:
  8KKKKKKK
  CODE HERE
  20000000
- Let's say we wanted to make it so pointer address [MAIN + 0x00759E0000] + 98C had it's 32-bit value written to 0x7FFFFFFF, but only whenever L and R were pressed at the same time. Remember K = 00000C0 for L+R. This is how we'd sandwich our pointer code between our button activator:
  800000C0
  580F0000 759E0000
  780F0000 0000098C
  640F0000 00000000 7FFFFFFF
  20000000
- Codewise, this basically says that "If Button is equal to L+R" "Then" write Pointer [MAIN + 0x00759E0000] + 98C to 0x7FFFFFFF. "Else", do nothing.

- In x86, the following code is basically this:
  CMP Button, C0
  JE WritePointer
  END:

  WritePointer:
  MOV r15, [MAIN + 0x00759E0000]
  ADD r15, 98C
  MOV [r15], 7FFFFFFF

Code Types 8 and 2 (Cont)

- Code Types 8 and 2 not only work with writing to Pointer Addresses, but pretty much any code type or function you can think of. For instance, we could just as easily sandwich Code Type 0 between Code Types 8 and 2. Let's say we wanted to make it so the 8-bit value 0x80 was written to [HEAP+0x00D0000000]. Let's also say we want this write to occur whenever L and R are pressed at the same time. The following code would accomplish just that:

  800000C0
  01100000 D0000000 00000080
  20000000

- The above code basically says that "If the Button is equal to L+R" "Then" write [HEAP + 0x00D0000000] to 0x80. "Else", do nothing.

• How to Make Pointer Codes with PointerSearcherSE

- Sometimes, even with all the work you did, you may not find a single pointer. This either means that your address is static or Noexs just couldn’t find anything. In that case, PointerSearcherSE can often find valid pointers that Noexs cannot. Noexs tends to only allow you to find 2-level pointers, with 3-level pointers being the maximum the program will go. PointerSearcherSE, however, can resolve some rather deep pointer chains.
- As an example, I’ll be making a pointer code with PointerSearcherSE for Super Smash Bros. Ultimate, version 11.0.1, to write the Player Stocks Address to 99 consistently, making it so the player cannot lose lives.

How to Make Pointer Codes with PointerSearcherSE

- Get your Switch's IP Address, go in-game and have your targeted address load, and go into Edizon SE.
- When in Edizon SE, open PointerSearcherSE and enter your IP Address into the "IP Address" box under "Wireless feature". Click "Attach to" and then click "Resume Game".
- Find your target address in Edizon SE, then after that, press the "Plus" button to add the address as a bookmark. Give it a name of some sort. Click "L" to toggle Bookmark.
- Now, click "Download Bookmark" in PointerSearcherSE. Click on the address to highlight it, then click "Dump ptr".
- After it finishes, write the bookmark Address into "TargetAddress1" and click "Read 1st Dump Data." Then, click "Reset and Search."
- If no results are found, change the values in :Max Depth", "OffsetNum", or "Offset Range". At first, I increase "OffsetNum" to 2, Read 1st Dump Data, and click Reset and Search again. If that still doesn't work, then increase "OffsetNum" to 3, then to 4, and if that still doesn't work, then I increase "Max Depth" to 5. If that still doesn't work, you can continue the previous pattern or decide to increase the "OffsetRange". It's likely that with each increase in each value section, the search will take longer.
- If you find so many pointers that you're not comfortable testing them all, write down the address in "TargetAddress1" into a Notepad file, along with the used "Max Depth", "OffsetNum", and "OffsetRange". Save that information in a notepad file and then exit out of PointerSearcherSE. Then, rename the extension the file Direct Transfer.dmp0 of .dmp0 to something else, like .dmp1 or .dmp2. Also, rename the Direct Transfer.tmp0 file to a matching number, like .tmp1 or .tmp2.
- Repeat the process, getting another dump, another bookmarked address, and another pointer search finished. Go up top to click an empty box in the "Path" section and select the previous dump you renamed. Place the written address in "TargetAddress1", switch the file from File 1 to File 2, then click "Narrow Down". Rinse and repeat this as necessary, incrementing the file and renaming each new file each time, as the new file is always saved as .dmp0.
- After you get to a comfortable number of pointers to deal with, make pointer codes for each pointer you made. Try breaking the pointer codes in-game by selecting different characters, different levels, going between chapters, etc.
- Release the most stable pointer code that crashes the least. Sometimes, you might find multiple pointer codes that are just as stable as the rest. Either release them all just in case or release one of your choice. If you release one of your choice, and it suddenly doesn't work for people, retest your pointer codes and release another one of the pointer codes you didn't release.

• Sections That Carry Over to Other Consoles

- Pointers and their Logic (Although different cheating softwares have different Code Types, Pointer Addresses are something you'll see and need to work with across different consoles. Memory regions like MAIN and HEAP do not carry over to other consoles, though.)
- Button Activators (Once again, different Code Types for these button activators will exist, but generally, code handlers do have the ability to set button activators)
- x86-64 (The assembler language I briefly went over in the video can be found in most of the video games seen in Windows operating systems. You can use Cheat Engine to edit x86 instructions in order to make cheats for your games. The Nintendo Switch itself uses ARM for its assembler language and consoles might use other programming languages.)