This is the second installment in a series of articles that will deep dive into the EVM and build the foundational knowledge needed to become a “shadowy super coder”. This article will build on the knowledge gained from

Part 1so if you haven’t read it yet I encourage you to do so.

In

Part 1we explored how the EVM knows which bytecode to run depending on which contract function is called. This helped us build an understanding of the call stack, calldata, function signatures & the EVM opcodes instructions.

In Part 2 we’ll take a trip down “memory” lane and provide a comprehensive review of what contract memory is and how it works under the EVM hood.

A Trip Down Memory Lane

As you will recall in

Part 1we took a look at the default 1_Storage.sol contract from

remix.

We then generated the byte code and zoomed in on the part relating to function selection. In this article, we going to focus on the first 5 bytes of the contract runtime bytecode.

These 5 bytes represent the initialisation of the “free memory pointer”. To fully understand what that means and what these bytes do we must first build your understanding of the data structures that govern contract memory.

Memory Data Structure

Contract memory is a simple byte array, where data can be stored in 32 bytes (256 bit) or 1 byte (8 bit) chunks and read in 32 bytes (256 bit) chunks. The image below illustrates this structure along with the read/write functionality of contract memory.

This functionality is determined by the 3 opcodes that operate on memory.

- MSTORE (x, y) - Store a 32 byte (256-bit) value “y” starting at memory location “x”
- MLOAD (x) - Load 32 bytes (256-bit) starting at memory location “x” onto the call stack
- MSTORE8 (x, y) - Store a 1 byte (8-bit) value “y” at memory location “x” (the least significant byte of the 32-byte stack value).

MSTORE (x, y) - Store a 32 byte (256-bit) value “y” starting at memory location “x”

MLOAD (x) - Load 32 bytes (256-bit) starting at memory location “x” onto the call stack

MSTORE8 (x, y) - Store a 1 byte (8-bit) value “y” at memory location “x” (the least significant byte of the 32-byte stack value).

You can think of the memory location as simply the array index of where to start writing/reading the data. If you want to write/read more than 1 byte of data you simply continue writing or reading from the next array index.

EVM Playground

This

EVM playground)

While walking through the EVM playground above you may have noticed a few strange occurrences. First, when we wrote a single byte 0x22 using MSTORE8 to memory location 32 (0x20) the memory changed from

to

You may ask the question, what’s with all the additional zeros we only added 1 byte?

Memory Expansion

When your contract writes to memory, you have to pay for the number of bytes written. If you are writing to an area of memory that hasn't been written to before, there is an additional memory expansion cost for using it for the first time.

Memory is expanded in 32 bytes (256-bit) increments when writing to previously untouched memory space.

Memory expansion costs scale linearly for the first 724 bytes and quadratically after that.

Above our memory was 32 bytes before we wrote 1 byte at location 32. At this point we began writing into untouched memory, as a result, the memory was expanded by another 32-byte increment to 64 bytes.

Note that all locations in memory are well-defined initially as zero which is why we see 2200000000000000000000000000000000000000000000000000000000000000 added to our memory.

Remember Memory is a Byte Array

The second thing you may have noticed occurred when we ran an MLOAD from memory location 33 (0x21). We returned the following value to the call stack.

We were able to start our read from a non 32 factor.

Remember memory is a byte array meaning we can start our reads (and our writes) from any memory location. We are not constrained to multiples of 32. Memory is linear and can be addressed at the byte level.

Memory can only be newly created in a function. It can either be newly instantiated complex types like array/struct (e.g. via

new int[...]

) or copied from a

storage

referenced variable.

Now we have an understanding of the data structures let’s return to the free memory pointer.

Free Memory Pointer

The free memory pointer is simply a pointer to the location where free memory starts. It ensures smart contracts keep track of which memory locations have been written to and which haven’t.

This protects against a contract overwriting some memory that has been allocated to another variable.

When a variable is written to memory the contract will first reference the free memory pointer to determine where the data should be stored.

It then updates the free memory pointer by noting how much data is to be written to the new location. A simple addition of these 2 values will yield where the new free memory will start.

Bytecode

As mentioned before the free memory pointer is defined at the start of the runtime bytecode through these 5 opcodes.

These effectively state that the free memory pointer is located in memory at byte 0x40 (64 in decimal) and has a value of 0x80 (128 in decimal).

The immediate questions you may have are why the values 0x40 & 0x80 are used above. The answer to this can be found in the following statement.

Solidity’s memory layout reserves four 32-byte slots:

- 0x00

-

0x3f

(64 bytes): scratch space

- 0x40

-

0x5f

(32 bytes): free memory pointer

- 0x60

-

0x7f

(32 bytes): zero slot

0x00

-

0x3f

(64 bytes): scratch space

0x40

-

0x5f

(32 bytes): free memory pointer

0x60

-

0x7f

(32 bytes): zero slot

We can see that 0x40 is the predefined location by solidity for the free memory pointer. The value 0x80 is merely the first memory byte that is available to write to after the 4 reserved 32-byte slots.

We'll quickly run through what each reserved section does.

- Scratch space, can be used between statements i.e. within inline assembly and for hashing methods.
- Free memory pointer, currently allocated memory size, start location of free memory, 0x80 initially.
- The zero slot, is used as an initial value for dynamic memory arrays and should never be written to.

Scratch space, can be used between statements i.e. within inline assembly and for hashing methods.

Free memory pointer, currently allocated memory size, start location of free memory, 0x80 initially.

The zero slot, is used as an initial value for dynamic memory arrays and should never be written to.

Memory in a Real Contract

To consolidate what we've learned so far we're going to look at how memory and the free memory pointer update within real solidity code.

I've created a MemoryLane Contract and intentionally kept it extremely simple. It has a single function that merely defines two arrays of lengths 5 & 2 and then assigns b[0] a value of 1. Despite the simplicity, there is a lot that goes on when these 3 lines of code are executed.

To view the details of how this solidity code executes within the EVM it can be copied into a

remixIDE. After it's copied you can compile the code, deploy it, run the memoryLane() function and then enter debugging mode to step through the opcodes (See

herefor instructions on how to do this). I have extracted a simplified version into an

EVM
Playground128%3F%2480Qlocatio9for%20NG40)64%3F%C2%8EQjump!9%7Brequired%20to%20prevent%20stack%20underflow%7D%20_PUSH2Gffff***BBB*~~~*%C2%83a%7F%2F*BBB~~~***Qload%2(**
tems%20%C2%86andG80_SWAP1QpushG20)32%C2%88e%7D%2420QDuplicatXthX3rd%3B%C2%86to%20thX%C2%84_DUP3Q0x05%20*G20)5%20*%2032-
9decmial)160%20o9%C2%84(sizXof%20array-9bytes%7D_MULQDuplicatX0xa0)160%3F_DUP1QReturns%20sizXof%20calldata-9bytes%20currently%20just%C2%90%3DG04%20or%204-
9decmial%C2%89stack(0x80%7D_DUP4Q0x80(%7CXcopie%40Ga%C2%8A_Qthis%20offsets%20thX4%20bytes-
9our%20call%20data%20with%20a%20sizXofGa0%20which%20yeild%20a%20160%20bit%20set%20of%200%22s%20to%20bXstored%20at%20thXN!n_Qthis%20effectively-
nitialises%20our%20array-
9m%3A%C2%87%60%C2%80%25eded_%230xa0_DUP1%230x80_DUP3Qnew%20N%20as%20before_ADDQswap%201st(0x120%7D%3Bo9%3E%20and%203rd(0x80%7D_SWAP2K80%7D_P(
9%2440_MLOAD%23N(0x120%7D_DUP1Q0x40)64%3F%2C%2032%20*%202)64%20second%C2%8C2%2440QN(0x12%C2%8240%7D)new%20N_ADDQsavXnew%20N%20valuXat%20freXmem
9m%3Ato-
nitialise%7D_MUL%230x40(N!n%7D_DUP1QsamXas%20beforX4%20bytes%20for%C2%900x04%C2%89%3E%20%3DG120_DUP4Q0x120(%7CXcopie%40G4%C2%8A%C2%87%60%C2%80%
nput-
s%200%20_SHL%C2%91120%7D_DUP2QpushG00)%5B0%5D%20wherXi9thXarray%20should%20this%3Bgo%2400QpushG20)64%20bytes%20thXlength%20of%20thXarray%20%2402%C2%
s%20not%20trying%20to%20storXa%20valuXat%20a!9that%20doesn%22t%20exist-9thXarray%7D_LTQjump!n_PUSH2G00d7Q2%20POPs%20sincXthis-s%20a%20JUMPI(checking-
f%20LT%20returned%20truXor%20false%7D%C2%8F%C2%8F_QpushG20(32%20bytes%20aray%3Bsize%7D%2420Q0x20%20*G00%20%3DG00)0%C2%88X*-
ndex%20to%20determinXbytXoffset%7D_MULQ0x00%20%2BG120_ADD%232nd%20o9stackG01(valuXfor%20b%5B0%5D%7D_DUP2%232nd%20o9stackG120(memory!9for%20b%5B%5D%
%20i)%20%3D%20(%20%7B!%20locatio%23QduplicatX%24_PUSH1G%25%5C'a%5C'%20and%20removes%20any-tems%20that%20arXno%20longer%20ne%3Aemory%20%3B-
tem%20%3EthXstack%3F-9decimal%40d.%7D%2CG04(bytXoffset-9thXcalldata%20to%20copy.%7D%2C%5E%20removXthXtop%3Boff%20stack%20with%20POP_POP_%60ng%20lines-
9this%20sectio9manipulatX%3E%20to%20%7CbytXoffset-
9thXm%3AwherXthXresult%20will%20b%7F%E2%80%9D%20%26%20FreXM%3APointer%20UpdatX~%C2%80ensurXwXhavXthXmemory!9of%20variablX%C2%81wap%20top%202-
temsG%C2%820%7D%20%2B%20spacXfor%20array(0x%C2%83QM%3AAllocatio9VaribalX%E2%80%9C%C2%84top%20of%20%3E%C2%85QSimulated%20jump!n_PUSH2Gffff%C2%86o9%
9this%20casX0x05%20%C2%87_CALLDATACOPY_QThXremaini%C2%88%3F(array%3Bsiz%C2%89_CALLDATASIZE%234th%3Bo9%C2%8A0(bytXsizXto%20copy.%7D%C2%8BInitialisatio
s%20length%20%C2%8DQsimulating%20a%20JUMP%C2%8E%2440_MSTORE%C2%8FQsimulatXJUMPI%20_POP%C2%90%20functio9signaturX%C2%91%232nd%3Bo9stack(0x%01%C2%91%
()-9BGKNQX_~_)and will run through it below.

The simplified version organises the opcodes sequentially removing any JUMP's and any code that isn't relevant to memory manipulation. Comments have been added to the code to provide context to what is being done. The code is split into 6 distinct sections which we will delve into.

I cannot stress enough how important it is to use the

playground128%3F%2480Qlocatio9for%20NG40)64%3F%C2%8EQjump!9%7Brequired%20to%20prevent%20stack%20underflow%7D%20_PUSH2Gffff***BBB*~~~*%C2%83a%7F%2F*BBB~~~***Qload%2(**
tems%20%C2%86andG80_SWAP1QpushG20)32%C2%88e%7D%2420QDuplicatXthX3rd%3B%C2%86to%20thX%C2%84_DUP3Q0x05%20*G20)5%20*%2032-
9decmial)160%20o9%C2%84(sizXof%20array-9bytes%7D_MULQDuplicatX0xa0)160%3F_DUP1QReturns%20sizXof%20calldata-9bytes%20currently%20just%C2%90%3DG04%20or%204-
9decmial%C2%89stack(0x80%7D_DUP4Q0x80(%7CXcopie%40Ga%C2%8A_Qthis%20offsets%20thX4%20bytes-
9our%20call%20data%20with%20a%20sizXofGa0%20which%20yeild%20a%20160%20bit%20set%20of%200%22s%20to%20bXstored%20at%20thXN!n_Qthis%20effectively-
nitialises%20our%20array-

9m%3A%C2%87%60%C2%80%25eded_%230xa0_DUP1%230x80_DUP3Qnew%20N%20as%20before_ADDQswap%201st(0x120%7D%3Bo9%3E%20and%203rd(0x80%7D_SWAP2K80%7D_P(
9%2440_MLOAD%23N(0x120%7D_DUP1Q0x40)64%3F%2C%2032%20%202)64%20second%C2%8C2%2440QN(0x12%C2%8240%7D)new%20N_ADDQsavXnew%20N%20valuXat%20freXmem
9m%3Ato-
nitialise%7D_MUL%230x40(N!n%7D_DUP1QsamXas%20beforX4%20bytes%20for%C2%900x04%C2%89%3E%20%3DG120_DUP4Q0x120(%7CXcopie%40G4%C2%8A%C2%87%60%C2%80%
nput-
s%200%20_SHL%C2%91120%7D_DUP2QpushG00)%5B0%5D%20wherXi9thXarray%20should%20this%3Bgo%2400QpushG20)64%20bytes%20thXlength%20of%20thXarray%20%2402%C2%
s%20not%20trying%20to%20storXa%20valuXat%20a!9that%20doesn%22t%20exist-9thXarray%7D_LTQjump!n_PUSH2G00d7Q2%20POPs%20sincXthis-s%20a%20JUMPI(checking-
*f%20LT%20returned%20truXor%20false%7D%C2%8F%C2%8F_QpushG20(32%20bytes%20aray%3Bsize%7D%2420Q0x20%20*G00%20%3DG00)0%C2%88X*-
ndex%20to%20determinXbytXoffset%7D_MULQ0x00%20%2BG120_ADD%232nd%20o9stackG01(valuXfor%20b%5B0%5D%7D_DUP2%232nd%20o9stackG120(memory!9for%20b%5B%5D%
%20i)%20%3D%20(%20%7B!%20locatio%23QduplicatX%24_PUSH1G%25%5C'a%5C'%20and%20removes%20any-tems%20that%20arXno%20longer%20ne%3Aemory%20%3B-
tem%20%3EthXstack%3F-9decimal%40d.%7D%2CG04(bytXoffset-9thXcalldata%20to%20copy.%7D%2C%5E%20removXthXtop%3Boff%20stack%20with%20POP_POP_%60ng%20lines-
9this%20sectio9manipulatX%3E%20to%20%7CbytXoffset-
9thXm%3AwherXthXresult%20will%20b%7F%E2%80%9D%20%26%20FreXM%3APointer%20UpdatX~%C2%80ensurXwXhavXthXmemory!9of%20variablX%C2%81wap%20top%202-
temsG%C2%820%7D%20%2B%20spacXfor%20array(0x%C2%83QM%3AAllocatio9VaribalX%E2%80%9C%C2%84top%20of%20%3E%C2%85QSimulated%20jump!n_PUSH2Gffff%C2%86o9%
9this%20casX0x05%20%C2%87_CALLDATACOPY_QThXremaini%C2%88%3F(array%3Bsiz%C2%89_CALLDATASIZE%234th%3Bo9%C2%8A0(bytXsizXto%20copy.%7D%C2%8BInitialisatio
s%20length%20%C2%8DQsimulating%20a%20JUMP%C2%8E%2440_MSTORE%C2%8FQsimulatXJUMPI%20_POP%C2%90%20functio9signaturX%C2%91%232nd%3Bo9stack(0x%01%C2%91%
()-9BGKNQX_~_)and step through the opcodes yourself. This will greatly enhance your learning. Now let's dig into the 6 sections.

Free Memory Pointer Initialisation (EVM Playground Lines 1-15)

First, we have "free memory pointer initialisation" which we have discussed above. A value of 0x80 (128 in decimal) is pushed onto the stack. This is the value of the free memory pointer and is determined by Solidity's memory layout. At this stage, we have nothing in memory.

Next, we push the free memory pointer location 0x40 (64 in decimal) again determined by Solidity's memory layout.

Finally, we call MSTORE which pops the first item off the stack 0x40 to determine where to write to in memory and the second value 0x80 as what to write.

This leaves us with an empty stack but we have now populated some memory. This memory representation is in hexadecimal where each character represents 4 bits.

We have 192 hexadecimal characters in memory which means we have 96 bytes (1 byte = 8 bits = 2 hexadecimal characters).

If we refer back to Solidity's memory layout we were told the first 64 bytes would be allocated as scratch space and the next 32 would be for the free memory pointer.

Thats exactly what we have below.

Memory Allocation Variable "a" & Free Memory Pointer Update (EVM Playground Lines 16-34)

For the remaining sections, we're going to skip to the end state of each section and give a high-level overview of what happened for brevity. The individual opcode steps can be seen via the

EVM
playground128%3F%2480locatio9for%20NG40)64%3F%C2%8EQjump!9%7Brequired%20to%20prevent%20stack%20underflow%7D%20_PUSH2Gffff*BBB*~~~*%C2%83a%7F%2F*BBB~~~*Qload%2(*
*tems%20%C2%86andG80_SWAP1QpushG20)32%C2%88e%7D%2420QduplicatXthX3rd%3B%C2%86to%20thX%C2%84_DUP3Q0x05%20*G20)5%20*%2032-*
9decmial)160%20o9%C2%84(sizXof%20array-9bytes%7D_MULQDuplicatX0xa0)160%3F_DUP1QReturns%20sizXof%20calldata-9bytes%20currently%20just%C2%90%3DG04%20or%204-
9decmial%C2%89stack(0x80%7D_DUP4Q0x80(%7CXcopie%40Ga%C2%8A_Qthis%20offsets%20thX4%20bytes-
9our%20call%20data%20with%20a%20sizXofGa0%20which%20yeild%20a%20160%20bit%20set%20of%200%22s%20to%20bXstored%20at%20thXN!n_Qthis%20effectively-
nitialises%20our%20array-
9m%3A%C2%87%60%C2%80%25eded_%230xa0_DUP1%230x80_DUP3Qnew%20N%20as%20before_ADDQswap%201st(0x120%7D%3Bo9%3E%20and%203rd(0x80%7D_SWAP2K80%7D_P(
9%2440_MLOAD%23N(0x120%7D_DUP1Q0x40)64%3F%2C%2032%20%202)64%20second%C2%8C2%2440QN(0x12%C2%8240%7D)new%20N_ADDQsavXnew%20N%20valuXat%20freXmem
9m%3Ato-
nitialise%7D_MUL%230x40(N!n%7D_DUP1QsamXas%20beforX4%20bytes%20for%C2%900x04%C2%89%3E%20%3DG120_DUP4Q0x120(%7CXcopie%40G4%C2%8A%C2%87%60%C2%80%
nput-
s%200%20_SHL%C2%91120%7D_DUP2QpushG00)%5B0%5D%20wherXi9thXarray%20should%20this%3Bgo%2400QpushG20)64%20bytes%20thXlength%20of%20thXarray%20%2402%C2%
s%20not%20trying%20to%20storXa%20valuXat%20a!9that%20doesn%22t%20exist-9thXarray%7D_LTQjump!n_PUSH2G00d7Q2%20POPs%20sincXthis-s%20a%20JUMPI(checking-
*f%20LT%20returned%20truXor%20false%7D%C2%8F%C2%8F_QpushG20(32%20bytes%20aray%3Bsize%7D%2420Q0x20%20*G00%20%3DG00)0%C2%88X*-
ndex%20to%20determinXbytXoffset%7D_MULQ0x00%20%2BG120_ADD%232nd%20o9stackG01(valuXfor%20b%5B0%5D%7D_DUP2%232nd%20o9stackG120(memory!9for%20b%5B%5D%
%20i)%20%3D%20(%20%7B!%20locatio%23QduplicatX%24_PUSH1G%25%5C'a%5C'%20and%20removes%20any-tems%20that%20arXno%20longer%20ne%3Aemory%20%3B-
tem%20%3EthXstack%3F-9decimal%40d.%7D%2CG04(bytXoffset-9thXcalldata%20to%20copy.%7D%2C%5E%20removXthXtop%3Boff%20stack%20with%20POP_POP_%60ng%20lines-
9this%20sectio9manipulatX%3E%20to%20%7CbytXoffset-
9thXm%3AwherXthXresult%20will%20b%7F%E2%80%9D%20%26%20FreXM%3APointer%20UpdatX~%C2%80ensurXwXhavXthXmemory!9of%20variablX%C2%81wap%20top%202-
temsG%C2%820%7D%20%2B%20spacXfor%20array(0x%C2%83QM%3AAllocatio9VaribalX%E2%80%9C%C2%84top%20of%20%3E%C2%85QSimulated%20jump!n_PUSH2Gffff%C2%86o9%
9this%20casX0x05%20%C2%87_CALLDATACOPY_QThXremaini%C2%88%3F(array%3Bsiz%C2%89_CALLDATASIZE%234th%3Bo9%C2%8A0(bytXsizXto%20copy.%7D%C2%8BInitialisatio
s%20length%20%C2%8DQsimulating%20a%20JUMP%C2%8E%2440_MSTORE%C2%8FQsimulatXJUMPI%20_POP%C2%90%20functio9signaturX%C2%91%232nd%3Bo9stack(0x%01%C2%91%
()-9BGKNQX_~_).

Next memory is allocated for variable "a"

(bytes32[5])

and the free memory pointer is updated.

The compiler will have determined how much space is required through the array size and the default array element size.

Remember elements in memory arrays in Solidity always occupy multiples of 32 bytes (this is even true for

bytes1[]

, but not for

bytes

and

string

)

The size of the array multiplied by 32 bytes tells us how much memory we need to allocate.

In this case that calculation 5 * 32 yields 160 or 0xa0 in hex. We can see this being pushed onto the stack and added to the current free memory pointer 0x80 (128 in decimal) to get the new free memory pointer value.

This returns 0x120 (288 in decimal) which we can see has been written to the free memory pointer location.

The call stack keeps the memory location of the variable "a" on the stack 0x80 so it can reference it later if needed. 0xffff represents a JUMP location and can be ignored since it isn't relevant to memory manipulation.

Memory Initialisation Variable "a" (EVM Playground Lines 35-95)

Now that the memory has been allocated and the free memory pointer updated we need to initialise the memory space for variable "a". Since the variable is just declared and not assigned it will be initialised with the zero value.

To do this write the EVM uses CALLDATACOPY which takes in 3 variables.

- memoryOffset (which memory location to copy the data to)
- calldataOffset (byte offset in the calldata to copy)
- size (byte size to copy)

memoryOffset (which memory location to copy the data to)

calldataOffset (byte offset in the calldata to copy)

size (byte size to copy)

In our case, the memoryOffset is the memory location for variable “a” (0x80). The calldataOffset is the actual size of our calldata since we don’t want to copy any of the calldata, we want to initialise the memory with the zero value. Finally, the size is 0xa0 or 160 bytes since that is the size of the variable.

We can see our memory has expanded to 288 bytes (this includes the zero slot) and the stack again holds the memory location of the variable and a JUMP location on the call stack.

Memory Allocation Variable “b” & Free Memory Pointer Update (EVM Playground Lines 96-112)

This is the same as the memory allocation and free memory pointer update for variable “a”

except this time it is for

“bytes32[2] memory b”

.

The memory pointer is updated to 0x160 (352 in decimal) which is equal to the previous free memory pointer 288 plus the size of the new variable in bytes 64.

Note that the free memory pointer has updated in memory to 0x160 and we now have the memory location for variable “b” (0x120) on the stack.

Memory Initialisation Variable “b” (EVM Playground Lines 113-162)

Same as memory initialisation of variable “a”.

Note that memory has increased to 352 bytes. The stack still holds memory locations for the 2 variables.

Assign Value to b[0] (EVM Playground Lines 163-207)

Finally, we get to assigning a value to array “b” index 0. The code states that b[0] should have a value of 1.

This value is pushed onto the stack 0x01. A bit shift left occurs next however the input for the bit shift is 0 meaning our value doesn’t change.

Next, the array index position to be written to 0x00 is pushed to the stack and a check is done to verify this value is less than the length of the array 0x02. If it isn’t the execution jumps to a different part of the bytecode which handles this error state.

The MUL (multiply) & ADD opcodes are used to determine where in memory the value needs to be written for it to correspond to the correct array index.

Remember memory arrays are 32-byte elements so this value represents the start location of an array index. Given we are writing to index 0 we have no offset.

ADD is used to add this offset value to the memory location for variable “b”. Given our offset was 0 we will write our data straight to the assigned memory location.

Finally, an MSTORE stores the value 0x01 to this memory location 0x120.

The image below shows the system state at the end of the function execution. All the stack items have been popped off.

Note in actuality in remix there are a few items left on the stack, a JUMP location and the function signature however they are not relevant to memory manipulation and therefore have been omitted in the EVM playground.

Our memory has been updated to include the b[0] = 1 assignment, on the third last line of our memory a 0 value has turned into a 1.

You can verify the value is at the correct memory location, b[0] should occupy locations 0x120 - 0x13f (bytes 289 - 320).

There we have it , that was a lot of information to take in but we now have a solid understanding of how contract memory works. This will serve us well the next time we need to write s code.

When you’re jumping through some contract opcodes and see certain memory locations that keep popping up (0x40) you’ll now know exactly what they mean.

Next, in the series, we “Demystify Storage Slot Packing” in

EVM Deep Dives - Part 3

Until next time.

noxx

Twitter

@noxx3xxon