

# EVM

From Solidity to bytes code, memory and storage

Peter Robinson & David Hyland-Wood

10 June 2020

# Acknowledgements

Thank you to the following people for reviewing these slides:

- Ben Edgington
- Nicolas Liochon
- Meredith Baxter
- Akua Nti
- Joseph Chow

# Agenda

- Introduction
- Ethereum Transactions
- OpCodes in the Ethereum Yellow Paper
- Stack, Memory, Storage, Code, CallData, & Logs
- Contract Deployment, Constructors, & init code fragments
- Function Calls in a Simple Contract
- Payable Fallback Functions
- Storage
- Memory
- CodeCopy and ExtCodeCopy: Using Contract as Static Storage
- Aux Data / Solidity Metadata

# Examples (all Apache 2 Licensed)

All example code is available here:

- <https://github.com/ConsenSys/EthEngGroupSolidityExamples/tree/master/evm>

The bytecode output was created using:

- ByteCodeDump in this repo: <https://github.com/PegaSysEng/codewitness>

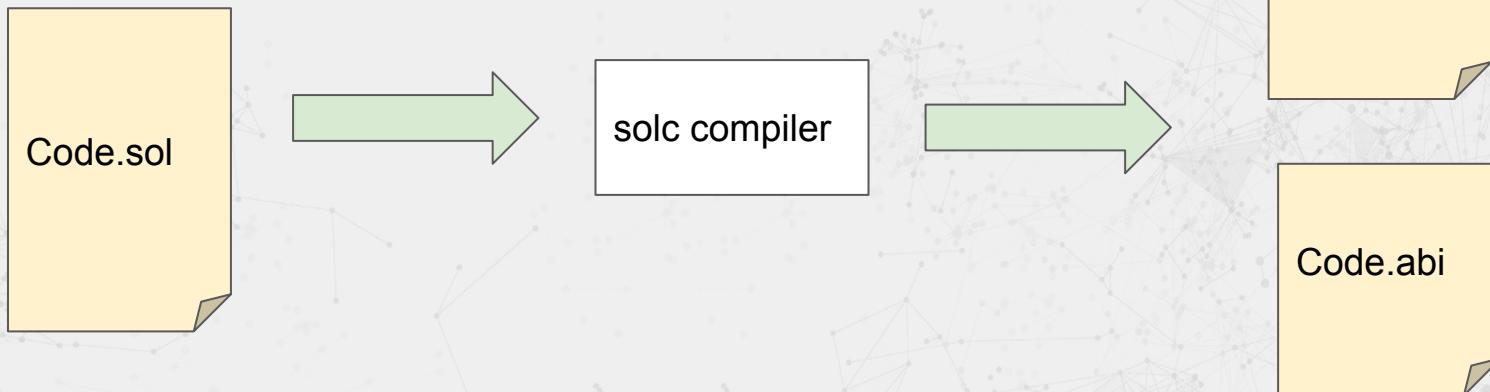
Stack analysis was created with the help of:

- FunctionIdAnalysis in this repo: <https://github.com/PegaSysEng/codewitness>

# Introduction



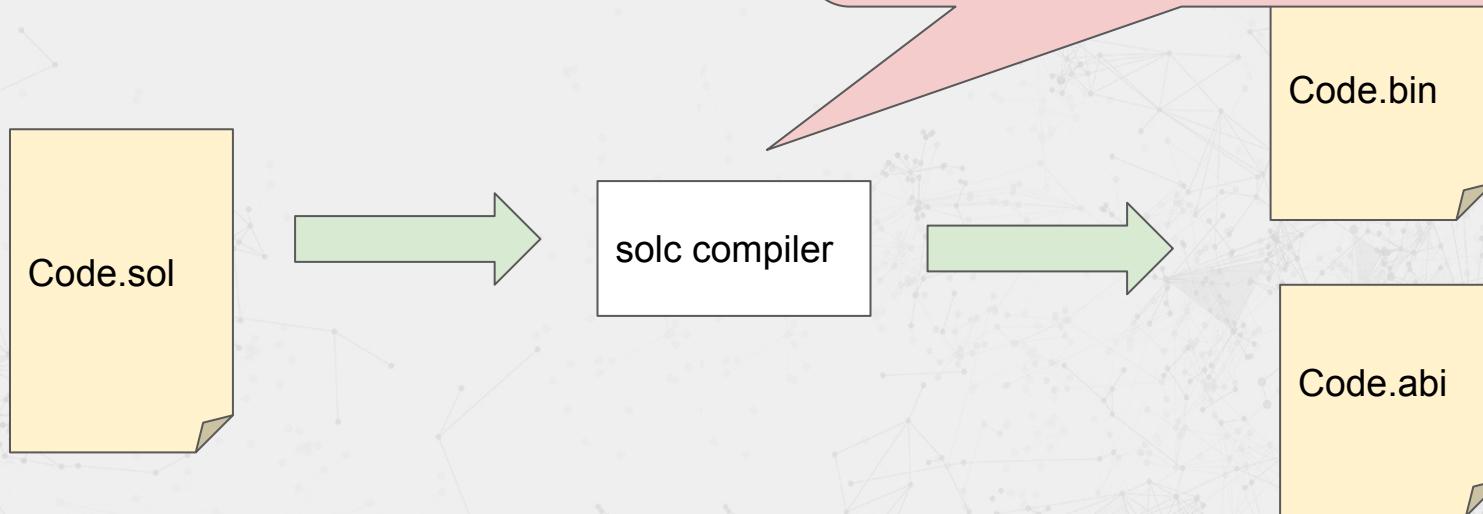
# Compilation Process



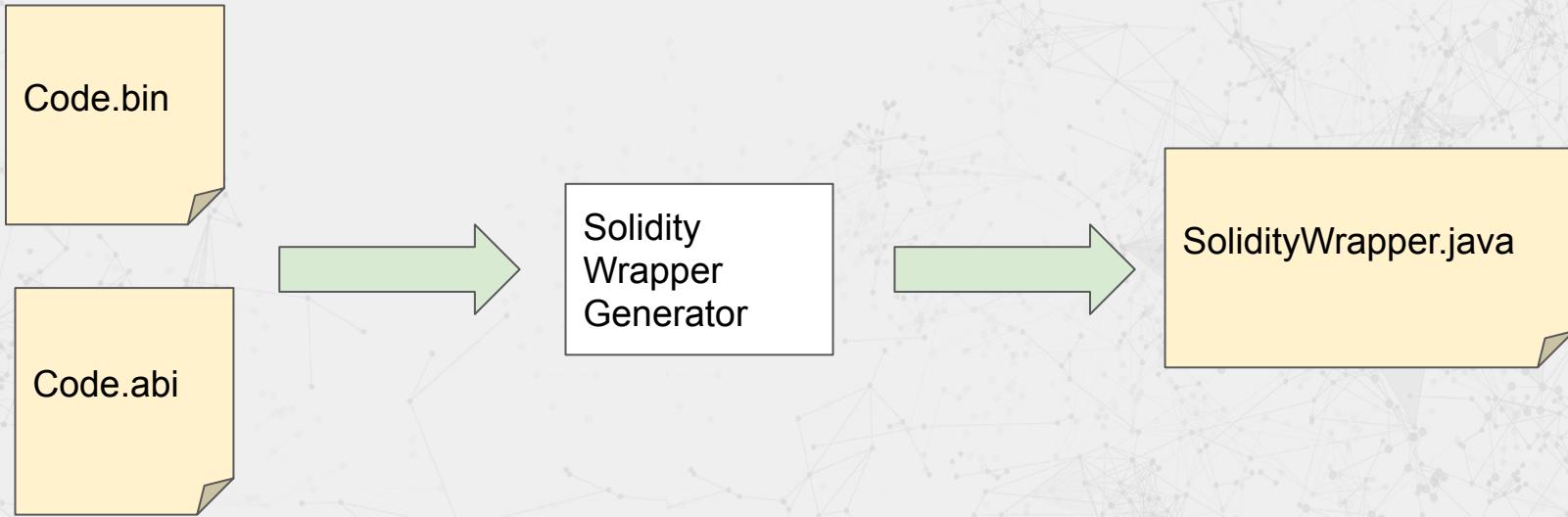
# Compilation Process

Solidity is only one EVM language.

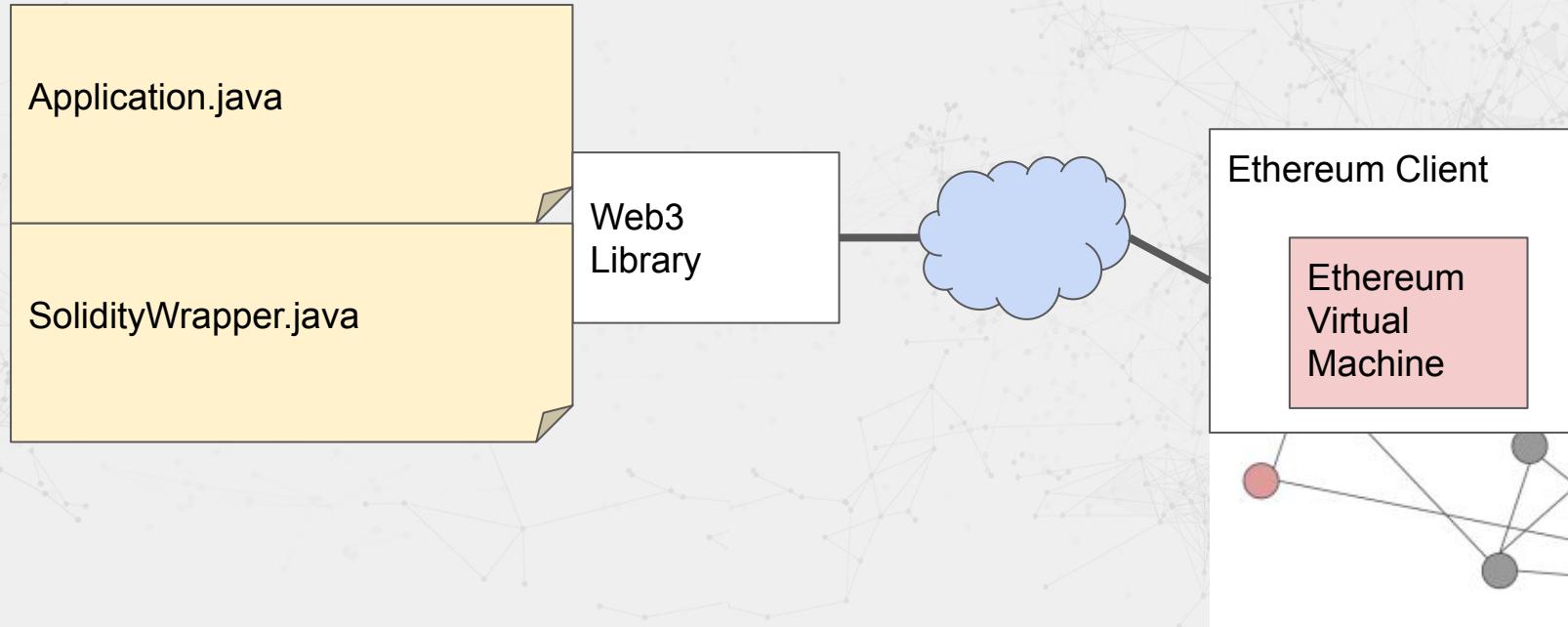
For Vyper:  
Code.vy is compiled with the **vyper** compiler.



# Application Creation Process



# Deployment Architecture



# Compiler

This talk is going to discuss the bytecode and how it interacts with the Ethereum Virtual Machine (EVM)

Code.sol



solc compiler



Code.bin

Code.abi

# Ethereum Transactions

# Transaction Fields

- **Nonce:** the transaction number for this account, starting with 0.
- **Gas Price:** the price in Wei that this transaction will pay per unit of gas.
- **Gas Limit:** the maximum amount of gas that this transaction can use.
- **To:** the address the transaction is sent to.
- **Value:** the amount of Wei to send.
- **Data:** Data to be sent to the To address.
- **v, r, s:** Components of the transaction signature.

# Transaction Fields: Value transfer

- **Nonce:** the transaction number for this account, starting with 0.
- **Gas Price:** the price in Wei that this transaction will pay for gas.
- **Gas Limit:** 21000
- **To:** the address to send the Ether to.
- **Value:** the amount of Wei to send.
- **Data:** Empty
- **v, r, s:** Components of the transaction signature.

# Transaction Fields: Value transfer

- **Nonce:** the transaction number for this block.
- **Gas Price:** the price in Wei that this transaction costs.
- **Gas Limit:** 21000
- **To:** the address to send the ether to.
- **Value:** the amount of ether to send.
- **Data:** Empty
- **v, r, s:** Components of the transaction signature.

Actually, it's not mandatory to be empty. For example, if you just want to record a commitment on chain, you can send yourself a zero valued transfer with data here.

... but....

Transactions aren't part of world state.

# Transaction Fields: Contract Deployment

- **Nonce**: the transaction number for this account, starting with 0.
- **Gas Price**: the price in Wei that this transaction will pay for gas.
- **Gas Limit**: the maximum amount of gas that this transaction can use.
- **To**: Empty
- **Value**: the amount of Wei to send.
- **Data**: The init code to set-up the contract state and deploy the contract.
- **v, r, s**: Components of the transaction signature.

# Transaction Fields: Function Call

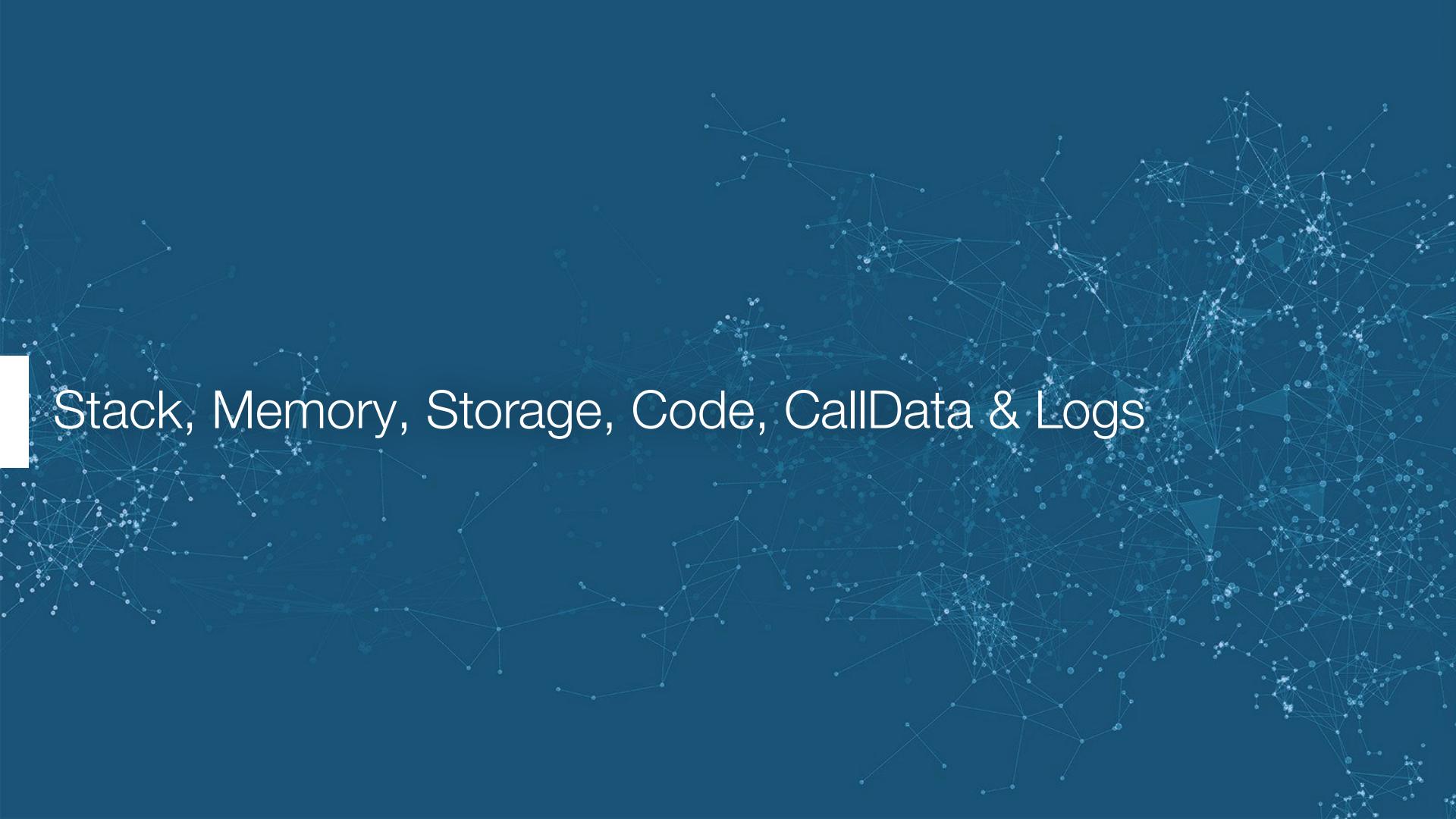
- **Nonce**: the transaction number for this account, starting with 0.
- **Gas Price**: the price in Wei that this transaction will pay for gas.
- **Gas Limit**: the maximum amount of gas that this transaction can use.
- **To**: the address the transaction is sent to.
- **Value**: the amount of Wei to send.
- **Data**: Data that the code in the EVM processes and does stuff with.
- **v, r, s**: Components of the transaction signature.

# Transaction Fields: Function

- **Nonce**: the transaction number for this account.
- **Gas Price**: the price in Wei that this transaction will pay for gas.
- **Gas Limit**: the maximum amount of gas this transaction is allowed to consume.
- **To**: the address the transaction is sent to.
- **Value**: the amount of Wei to send.
- **Data**: Function to call + parameters.
- **v, r, s**: Components of the transaction signature.

For Solidity contracts and other contracts that conform to the Application Binary Interface.

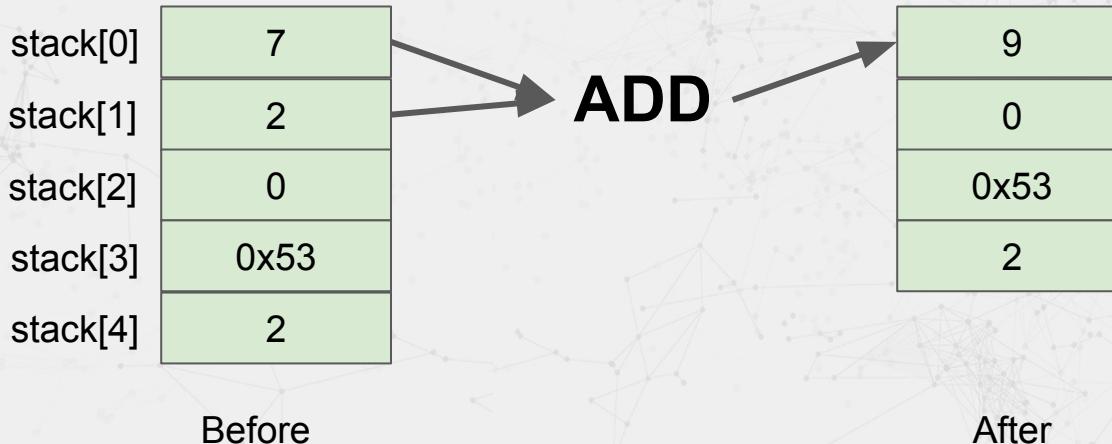
REF: <https://solidity.readthedocs.io/en/v0.6.9/abi-spec.html?highlight=abi>



Stack, Memory, Storage, Code, CallData & Logs

# Stack, Memory, Storage, Code, CallData, Logs

- The EVM is a stack based processor.
- EVM Opcodes pop information from and push information onto the stack.



# Stack, Memory, Storage, Code, CallData, Logs

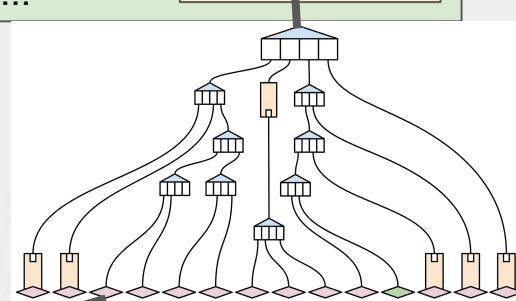
The EVM can access and store information in six places:

- Stack: EVM Opcodes pop information from and push data onto the stack.
- CallData: The data field of a transaction. These are parameters to the call.
- Memory: Information store accessible for the duration of a transaction.
- Storage: Persistent data store.
- Code: Executing code and static data storage.
- Logs: Write-only logger / event output.

## Block Header

Parent hash, ommers, block number, time stamp, difficulty, nonce, transaction root, receipts root...

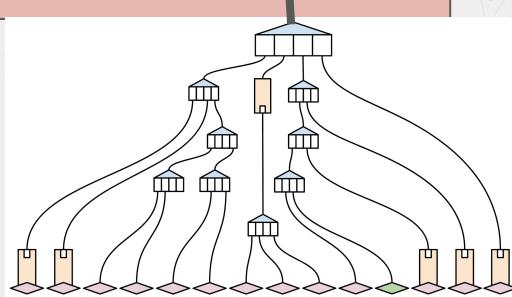
## State Root



## Account State

nonce, balance, code hash, Storage Root

## Storage Root



Merkle Patricia Trie of all storage values within the account, where the key is a 256 bit number.

# OpCodes in the Ethereum Yellow Paper

<https://github.com/ethereum/yellowpaper>

## ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER

BYZANTIUM VERSION 7e819ec - 2019-10-20

DR. GAVIN WOOD  
FOUNDER, ETHEREUM & PARITY  
GAVIN@PARITY.IO

**ABSTRACT.** The blockchain paradigm when coupled with cryptographically-secured transactions has demonstrated its utility through a number of projects, with Bitcoin being one of the most notable ones. Each such project can be seen as a simple application on a decentralised, but singleton, compute resource. We can call this paradigm a transactional singleton machine with shared-state.

Ethereum implements this paradigm in a generalised manner. Furthermore it provides a plurality of such resources, each with a distinct state and operating code but able to interact through a message-passing framework with others. We discuss its design, implementation issues, the opportunities it provides and the future hurdles we envisage.

### 1. INTRODUCTION

With ubiquitous internet connections in most places of the world, global information transmission has become incredibly cheap. Technology-rooted movements like Bitcoin have demonstrated through the power of the default, consensus mechanisms, and voluntary respect of the social contract, that it is possible to use the internet to make a decentralised value-transfer system that can be shared across the world and virtually free to use. This system can be said to be a very specialised version of a cryptographically secure, transaction-based state machine. Follow-up systems such as Namecoin adapted this original “currency application” of the technology into other applications albeit rather simplistic ones.

Ethereum is a project which attempts to build the generalised technology; technology on which all transaction-based state machine concepts may be built. Moreover it aims to provide to the end-developer a tightly integrated end-to-end system for building software on a hitherto unexplored compute paradigm in the mainstream: a trustful object messaging compute framework.

**1.1. Driving Factors.** There are many goals of this project; one key goal is to facilitate transactions between consenting individuals who would otherwise have no means to trust one another. This may be due to geographical separation, interfacing difficulty, or perhaps the incompatibility, incompetence, unwillingness, expense, uncertainty, inconvenience, or corruption of existing legal systems. By specifying a state-change system through a rich and unam-

is often lacking, and plain old prejudices are difficult to shake.

Overall, we wish to provide a system such that users can be guaranteed that no matter with which other individuals, systems or organisations they interact, they can do so with absolute confidence in the possible outcomes and how those outcomes might come about.

**1.2. Previous Work.** Buterin [2013a] first proposed the kernel of this work in late November, 2013. Though now evolved in many ways, the key functionality of a blockchain with a Turing-complete language and an effectively unlimited inter-transaction storage capability remains unchanged.

Dwork and Naor [1992] provided the first work into the usage of a cryptographic proof of computational expenditure (“proof-of-work”) as a means of transmitting a value signal over the Internet. The value-signal was utilised here as a spam deterrence mechanism rather than any kind of currency, but critically demonstrated the potential for a basic data channel to carry a *strong economic signal*, allowing a receiver to make a physical assertion without having to rely upon *trust*. Back [2002] later produced a system in a similar vein.

The first example of utilising the proof-of-work as a strong economic signal to secure a currency was by Vishnumurthy et al. [2003]. In this instance, the token was used to keep peer-to-peer file trading in check, providing “consumers” with the ability to make micro-payments to “suppliers” for their services. The security model afforded by the proof-of-work was augmented with digital signatures and a ledger in order to ensure that the historical record

# EVM OpCodes start on pg 28:

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER BYZANTIUM VERSION 7e819ec - 2019-10-2028

## 0s: Stop and Arithmetic Operations

All arithmetic is modulo  $2^{256}$  unless otherwise noted. The zero-th power of zero  $0^0$  is defined to be one.

Value	Mnemonic	$\delta$	$\alpha$	Description
0x00	STOP	0	0	Halts execution.
0x01	ADD	2	1	Addition operation. $\mu'_s[0] \equiv \mu_s[0] + \mu_s[1]$
0x02	MUL	2	1	Multiplication operation. $\mu'_s[0] \equiv \mu_s[0] \times \mu_s[1]$
0x03	SUB	2	1	Subtraction operation. $\mu'_s[0] \equiv \mu_s[0] - \mu_s[1]$
0x04	DIV	2	1	Integer division operation. $\mu'_s[0] \equiv \begin{cases} 0 & \text{if } \mu_s[1] = 0 \\ \lfloor \mu_s[0] \div \mu_s[1] \rfloor & \text{otherwise} \end{cases}$
0x05	SDIV	2	1	Signed integer division operation (truncated). $\mu'_s[0] \equiv \begin{cases} 0 & \text{if } \mu_s[1] = 0 \\ -2^{255} & \text{if } \mu_s[0] = -2^{255} \wedge \mu_s[1] = -1 \\ \text{sgn}(\mu_s[0] \div \mu_s[1]) \lfloor  \mu_s[0] \div \mu_s[1]  \rfloor & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers. Note the overflow semantic when $-2^{255}$ is negated.
0x06	MOD	2	1	Modulo remainder operation. $\mu'_s[0] \equiv \begin{cases} 0 & \text{if } \mu_s[1] = 0 \\ \mu_s[0] \bmod \mu_s[1] & \text{otherwise} \end{cases}$
0x07	SMOD	2	1	Signed modulo remainder operation. $\mu'_s[0] \equiv \begin{cases} 0 & \text{if } \mu_s[1] = 0 \\ \text{sgn}(\mu_s[0])(\lfloor \mu_s[0] \bmod  \mu_s[1]  \rfloor) & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers.
0x08	ADDMOD	3	1	Modulo addition operation. $\mu'_s[0] \equiv \begin{cases} 0 & \text{if } \mu_s[2] = 0 \\ (\mu_s[0] + \mu_s[1]) \bmod \mu_s[2] & \text{otherwise} \end{cases}$ All intermediate calculations of this operation are not subject to the $2^{256}$ modulo.
0x09	MULMOD	3	1	Modulo multiplication operation. $\mu'_s[0] \equiv \begin{cases} 0 & \text{if } \mu_s[2] = 0 \\ (\mu_s[0] \times \mu_s[1]) \bmod \mu_s[2] & \text{otherwise} \end{cases}$ All intermediate calculations of this operation are not subject to the $2^{256}$ modulo.
0x0a	EXP	2	1	Exponential operation. $\mu'_s[0] \equiv \mu_s[0]^{\mu_s[1]}$
0x0b	SIGNEXTEND	2	1	Extend length of two's complement signed integer. $\forall i \in [0..255] : \mu'_s[0]_i \equiv \begin{cases} \mu_s[1]_t & \text{if } i \leq t \text{ where } t = 256 - 8(\mu_s[0] + 1) \\ \mu_s[1]_i & \text{otherwise} \end{cases}$ $\mu_s[x]_i$ gives the $i$ th bit (counting from zero) of $\mu_s[x]$

## 0s: Stop and Arithmetic Operations

All arithmetic is modulo  $2^{256}$  unless otherwise noted. The zero-th power of zero  $0^0$  is defined to be one.

Value	Mnemonic	$\delta$	$\alpha$	Description
0x00	STOP	0	0	Halts execution.
0x01	ADD	2	1	Addition operation. $\mu'_s[0] \equiv \mu_s[0] + \mu_s[1]$
0x02	MUL	2	1	Multiplication operation. $\mu'_s[0] \equiv \mu_s[0] \times \mu_s[1]$
0x03	SUB	2	1	Subtraction operation. $\mu'_s[0] \equiv \mu_s[0] - \mu_s[1]$
0x04	DIV	2	1	Integer division operation. $\mu'_s[0] \equiv \begin{cases} 0 & \text{if } \mu_s[1] = 0 \\ \lfloor \mu_s[0] \div \mu_s[1] \rfloor & \text{otherwise} \end{cases}$
0x05	SDIV	2	1	Signed integer division operation (truncated).

# Contract Deployment, Constructors & Init Code Fragments

# Example

```
pragma solidity >=0.4.23;

contract Simple {
    uint256 public val1;
    uint256 public val2;

    constructor() public {
        val2 = 3;
    }

    function set(uint256 _param) external {
        val1 = _param;
    }
}
```

# Example

```
pragma solidity >=0.4.23;
```

```
contract Simple {
    uint256 public val1;
    uint256 public val2;

    constructor() public {
        val2 = 3;
    }

    function set(uint256 _param) external {
        val1 = _param;
    }
}
```

The init function includes code to deploy the contract plus the constructor to set-up the contract state.

# Compiling

```
solc Simple.sol --bin --abi --optimize -o . --overwrite
```

# Simple.bin

```
608060405234801561001057600080fd5b50600360015560c18061002460003960
00f3fe6080604052348015600f57600080fd5b5060043610603c5760003560e01c8
06360fe47b114604157806395cacbe014605d578063c82fdf36146075575b600080
fd5b605b60048036036020811015605557600080fd5b5035607b565b005b606360
80565b60408051918252519081900360200190f35b60636086565b600055565b60
015481565b6000548156fea265627a7a723058204e00129b67b55015b0de73c316
7fb19ae30a4bb9b293318b7fb6c40bc080b864736f6c634300050a0032
```

# Simple.bin

Init code fragment

```
608060405234801561001057600080fd5b50600360015560c18061002460003960
00f3fe6080604052348015600f57600080fd5b5060043610603c5760003560e01c8
06360fe47b114604157806395cacbe014605d578063c82fdf36146075575b600080
fd5b605b60048036036020811015605557600080fd5b5035607b565b005b606360
80565b60408051918252519081900360200190f35b60636086565b600055565b60
015481565b6000548156fea265627a7a723058204e00129b67b55015b0de73c316
7fb19ae30a4bb9b293318b7fb6c40bc080b864736f6c634300050a0032
```

Code to be deployed to the blockchain

# Constructors

Contracts are deployed using transactions where:

- **To** address is not specified.
- **Data** is the init code fragment. This includes the contract binary. This is the compiler output in the \*.bin file.

Note:

- The init function is NOT stored on the blockchain.
- The **Data** of the transaction is treated as code for contract deployment.

# Byte Code and OpCodes

6080

60405234801561001057600080fd5b50600360015560c18061002460003960  
00f3fe6080604052348015600f57600080fd5b5060043610603c5760003560e01c8  
06360fe47a114604157806395cacbe014605d578063c82fdf36146075575b600080  
fd5b605b60048186036020811015605557600080fd5b5035607b565b005b606360  
80565b604080519152519081900360200190f35b60636086565b600055565b60  
015481565b6000548153865627a7a723058204e00129b67b55015b0de73c316  
7fb19ae30a4bb9b293318b7a9bc080b864736f6c634300050a0032

PUSH 0x80

# ByteCode

PC: 0x0, opcode: PUSH1 0x80  
PC: 0x2, opcode: PUSH1 0x40  
PC: 0x4, opcode: MSTORE  
PC: 0x5, opcode: CALLVALUE  
PC: 0x6, opcode: DUP1  
PC: 0x7, opcode: ISZERO  
PC: 0x8, opcode: PUSH2 0x0010  
PC: 0xb, opcode: JUMPI  
PC: 0xc, opcode: PUSH1 0x00  
PC: 0xe, opcode: DUP1  
PC: 0xf, opcode: REVERT  
PC: 0x10, opcode: JUMPDEST  
....

# ByteCode

```
PC: 0x0, opcode: PUSH1 0x80
PC: 0x2, opcode: PUSH1 0x40
PC: 0x4, opcode: MSTORE
PC: 0x5, opcode: CALLVALUE
PC: 0x6, opcode: DUP1
PC: 0x7, opcode: ISZERO
PC: 0x8, opcode: PUSH2 0x0010
PC: 0xb, opcode: JUMPI
PC: 0xc, opcode: PUSH1 0x00
PC: 0xe, opcode: DUP1
PC: 0xf, opcode: REVERT
PC: 0x10, opcode: JUMPDEST
....
```

Push a word onto the stack.

The stack has a maximum depth of 1024 words.

stack[0] is the value at the top of the stack.  
stack[1] is the value one below the top of the stack.

PUSH1: push the byte following this opcode onto the stack. The top 31 bytes of the word are zero filled.

# ByteCode

```
PC: 0x0, opcode: PUSH1 0x80  
PC: 0x2, opcode: PUSH1 0x40  
PC: 0x4, opcode: MSTORE  
PC: 0x5, opcode: CALLVALUE  
PC: 0x6, opcode: DUP1  
PC: 0x7, opcode: ISZERO  
PC: 0x8, opcode: PUSH2 0x0010  
PC: 0xb, opcode: JUMPI  
PC: 0xc, opcode: PUSH1 0x00  
PC: 0xe, opcode: DUP1  
PC: 0xf, opcode: REVERT  
PC: 0x10, opcode: JUMPDEST  
....
```

PUSH1 0x80

Push a new item onto the stack.

stack[0] = 0x80

# ByteCode

```
PC: 0x0, opcode: PUSH1 0x80
PC: 0x2, opcode: PUSH1 0x40
PC: 0x4, opcode: MSTORE
PC: 0x5, opcode: CALLVALUE
PC: 0x6, opcode: DUP1
PC: 0x7, opcode: ISZERO
PC: 0x8, opcode: PUSH2 0x0010
PC: 0xb, opcode: JUMPI
PC: 0xc, opcode: PUSH1 0x00
PC: 0xe, opcode: DUP1
PC: 0xf, opcode: REVERT
PC: 0x10, opcode: JUMPDEST
....
```

PUSH1 0x40

stack[1] = stack[0]  
stack[0] = 0x40

Stack now looks like:  
stack[0]: 0x40  
stack[1]: 0x80

# ByteCode

PC: 0x0, opcode: PUSH1 0x80  
PC: 0x2, opcode: PUSH1 0x40  
**PC: 0x4, opcode: MSTORE**  
PC: 0x5, opcode: CALLVALUE  
PC: 0x6, opcode: DUP1  
PC: 0x7, opcode: ISZERO  
PC: 0x8, opcode: PUSH2 0x0010  
PC: 0xb, opcode: JUMPI  
PC: 0xc, opcode: PUSH1 0x00  
PC: 0xe, opcode: DUP1  
PC: 0xf, opcode: REVERT  
PC: 0x10, opcode: JUMPDEST  
....

MSTORE: Store a word in memory.

Pop two values off the stack.  
stack[0] is the location to write to.  
stack[1] is the value to write

Stack is now empty

# ByteCode

```
PC: 0x0, opcode: PUSH1 0x80  
PC: 0x2, opcode: PUSH1 0x40  
PC: 0x4, opcode: MSTORE  
  
PC: 0x5, opcode: CALLVALUE  
PC: 0x6, opcode: DUP1  
PC: 0x7, opcode: ISZERO  
PC: 0x8, opcode: PUSH2 0x0010  
PC: 0xb, opcode: JUMPI  
PC: 0xc, opcode: PUSH1 0x00  
PC: 0xe, opcode: DUP1  
PC: 0xf, opcode: REVERT  
PC: 0x10, opcode: JUMPDEST  
  
....
```

## Set-up the Free Memory Pointer

Memory [0x40] = 0x80

Meaning that the Solidity code can safely allocate memory starting at 0x80. If code allocates memory, the pointer should be updated.

NOTE: the addressing is in bytes, but MSTORE stores 32 byte words.

# ByteCode

PC: 0x0, opcode: PUSH1 0x80  
PC: 0x2, opcode: PUSH1 0x40  
PC: 0x4, opcode: MSTORE  
**PC: 0x5, opcode: CALLVALUE**  
PC: 0x6, opcode: DUP1  
PC: 0x7, opcode: ISZERO  
PC: 0x8, opcode: PUSH2 0x0010  
PC: 0xb, opcode: JUMPI  
PC: 0xc, opcode: PUSH1 0x00  
PC: 0xe, opcode: DUP1  
PC: 0xf, opcode: REVERT  
PC: 0x10, opcode: JUMPDEST  
....

## CALLVALUE

Push onto the stack how much Wei was sent with the transaction. That is the VALUE transaction field.

Stack now looks like:  
stack[0]: value sent with transaction

# ByteCode

```
PC: 0x0, opcode: PUSH1 0x80  
PC: 0x2, opcode: PUSH1 0x40  
PC: 0x4, opcode: MSTORE  
PC: 0x5, opcode: CALLVALUE  
PC: 0x6, opcode: DUP1  
PC: 0x7, opcode: ISZERO  
PC: 0x8, opcode: PUSH2 0x0010  
PC: 0xb, opcode: JUMPI  
PC: 0xc, opcode: PUSH1 0x00  
PC: 0xe, opcode: DUP1  
PC: 0xf, opcode: REVERT  
PC: 0x10, opcode: JUMPDEST  
....
```

DUP1

Duplicate the top of the stack.

Stack now looks like:

stack[0]: value sent with transaction  
stack[1]: value sent with transaction

Note: DUP2 pushes a copy of stack[1] onto the stack and DUP3 pushes a copy of stack[2] onto the stack.

# ByteCode

PC: 0x0, opcode: PUSH1 0x80

PC: 0x2, opcode: PUSH1 0x40

PC: 0x4, opcode: MSTORE

PC: 0x5, opcode: CALLVALUE

PC: 0x6, opcode: DUP1

PC: 0x7, opcode: ISZERO

PC: 0x8, opcode: PUSH2 0x0010

PC: 0xb, opcode: JUMPI

PC: 0xc, opcode: PUSH1 0x00

PC: 0xe, opcode: DUP1

PC: 0xf, opcode: REVERT

PC: 0x10, opcode: JUMPDEST

....

## ISZERO

Pop a word off the stack. If the word is zero, push 1 onto the stack, otherwise push 0.

Stack now looks like:

stack[0]: 1 if value==0 and 0 if value!=0

stack[1]: value sent with transaction

# ByteCode

PC: 0x0, opcode: PUSH1 0x80  
PC: 0x2, opcode: PUSH1 0x40  
PC: 0x4, opcode: MSTORE  
PC: 0x5, opcode: CALLVALUE  
PC: 0x6, opcode: DUP1  
PC: 0x7, opcode: ISZERO  
**PC: 0x8, opcode: PUSH2 0x0010**  
PC: 0xb, opcode: JUMPI  
PC: 0xc, opcode: PUSH1 0x00  
PC: 0xe, opcode: DUP1  
PC: 0xf, opcode: REVERT  
PC: 0x10, opcode: JUMPDEST  
....

PUSH2 0x0010

Push the two bytes following the opcode onto the stack.

Stack now looks like:

stack[0]: 0x10

stack[1]: 1 if value==0 and 0 if value!=0

stack[2]: value sent with transaction

# ByteCode

PC: 0x0, opcode: PUSH1 0x80  
PC: 0x2, opcode: PUSH1 0x40  
PC: 0x4, opcode: MSTORE  
PC: 0x5, opcode: CALLVALUE  
PC: 0x6, opcode: DUP1  
PC: 0x7, opcode: ISZERO  
PC: 0x8, opcode: PUSH2 0x0010  
PC: 0xb, opcode: JUMPI  
PC: 0xc, opcode: PUSH1 0x00  
PC: 0xe, opcode: DUP1  
PC: 0xf, opcode: REVERT  
PC: 0x10, opcode: JUMPDEST  
....

## JUMPI

Set the Program Counter (PC) to stack[0] if stack[1] is not zero. Pop two values off the stack.

Stack now looks like:

stack[0]: value sent with transaction

# ByteCode

PC: 0x0, opcode: PUSH1 0x80  
PC: 0x2, opcode: PUSH1 0x40  
PC: 0x4, opcode: MSTORE  
PC: 0x5, opcode: CALLVALUE  
PC: 0x6, opcode: DUP1  
PC: 0x7, opcode: ISZERO  
PC: 0x8, opcode: PUSH2 0x0010  
PC: 0xb, opcode: JUMPI  
PC: 0xc, opcode: PUSH1 0x00  
PC: 0xe, opcode: DUP1  
PC: 0xf, opcode: REVERT  
PC: 0x10, opcode: JUMPDEST  
....

PUSH1 0x0, DUP1

Push 0x0 onto the stack twice

Stack now looks like:

stack[0]: 0

stack[1]: 0

stack[2]: value sent with transaction

# ByteCode

PC: 0x0, opcode: PUSH1 0x80  
PC: 0x2, opcode: PUSH1 0x40  
PC: 0x4, opcode: MSTORE  
PC: 0x5, opcode: CALLVALUE  
PC: 0x6, opcode: DUP1  
PC: 0x7, opcode: ISZERO  
PC: 0x8, opcode: PUSH2 0x0010  
PC: 0xb, opcode: JUMPI  
PC: 0xc, opcode: PUSH1 0x00  
PC: 0xe, opcode: DUP1  
PC: 0xf, opcode: REVERT  
PC: 0x10, opcode: JUMPDEST  
....

## REVERT

Halt execution and indicate a REVERT has occurred. Use stack[0] as a memory location and stack[1] as a length of Revert Reason.

Stack *looked* like this prior to execution:

stack[0]: 0

stack[1]: 0

stack[2]: value sent with transaction

# ByteCode

PC: 0x0, opcode: PUSH1 0x80  
PC: 0x2, opcode: PUSH1 0x40  
PC: 0x4, opcode: MSTORE  
PC: 0x5, opcode: CALLVALUE  
PC: 0x6, opcode: DUP1  
PC: 0x7, opcode: ISZERO  
PC: 0x8, opcode: PUSH2 0x0010  
PC: 0xb, opcode: JUMPI  
PC: 0xc, opcode: PUSH1 0x00  
PC: 0xe, opcode: DUP1  
PC: 0xf, opcode: REVERT  
PC: 0x10, opcode: JUMPDEST  
....

REVERT if the constructor which is NOT payable has Wei in the contract deploy transaction.

# ByteCode

```
PC: 0x0, opcode: PUSH1 0x80
PC: 0x2, opcode: PUSH1 0x40
PC: 0x4, opcode: MSTORE
PC: 0x5, opcode: CALLVALUE
PC: 0x6, opcode: DUP1
PC: 0x7, opcode: ISZERO
PC: 0x8, opcode: PUSH2 0x0010
PC: 0xb, opcode: JUMPI
PC: 0xc, opcode: PUSH1 0x00
PC: 0xe, opcode: DUP1
PC: 0xf, opcode: REVERT
PC: 0x10, opcode: JUMPDEST
...

```

## JUMPDEST

If the jump to program counter 0x10 was taken it would arrive here.

Valid jump destinations are indicated by the JUMPDEST opcode.

Stack now looks like:  
stack[0]: value sent with transaction

# ByteCode

```
PC: 0x10, opcode: JUMPDEST  
PC: 0x11, opcode: POP  
PC: 0x12, opcode: PUSH1 0x03  
PC: 0x14, opcode: PUSH1 0x01  
PC: 0x16, opcode: SSTORE  
PC: 0x17, opcode: PUSH1 0xc1  
PC: 0x19, opcode: DUP1  
PC: 0x1a, opcode: PUSH2 0x0024  
PC: 0x1d, opcode: PUSH1 0x00  
PC: 0x1f, opcode: CODECOPY  
PC: 0x20, opcode: PUSH1 0x00  
PC: 0x22, opcode: RETURN  
PC: 0x23, opcode: INVALID  
PC: 0x24, opcode: PUSH1 0x80  
....
```

POP

Pop to top value off the stack.

Stack is now empty

# Recall Example

```
pragma solidity >=0.4.23;
```

```
contract Simple {  
    uint256 public val1;  
    uint256 public val2;
```

```
constructor() public {  
    val2 = 3;  
}
```

```
function set(uint256 _param) external {  
    val1 = _param;  
}
```

NOTE: Constructor stores the constant 3 into the second storage variable, val2.

# ByteCode

PC: 0x10, opcode: JUMPDEST

PC: 0x11, opcode: POP

PC: 0x12, opcode: PUSH1 0x03

PC: 0x14, opcode: PUSH1 0x01

PC: 0x16, opcode: SSTORE

PC: 0x17, opcode: PUSH1 0xc1

PC: 0x19, opcode: DUP1

PC: 0x1a, opcode: PUSH2 0x0024

PC: 0x1d, opcode: PUSH1 0x00

PC: 0x1f, opcode: CODECOPY

PC: 0x20, opcode: PUSH1 0x00

PC: 0x22, opcode: RETURN

PC: 0x23, opcode: INVALID

PC: 0x24, opcode: PUSH1 0x80

....

SSTORE: Store a word to storage.

Pop two values off the stack.

stack[0] is the location to write to.

stack[1] is the value to write

Storage[1] = 3

Stack is now empty

# ByteCode

PC: 0x10, opcode: JUMPDEST

PC: 0x11, opcode: POP

PC: 0x12, opcode: PUSH1 0x03

PC: 0x14, opcode: PUSH1 0x01

PC: 0x16, opcode: SSTORE

PC: 0x17, opcode: PUSH1 0xc1

PC: 0x19, opcode: DUP1

PC: 0x1a, opcode: PUSH2 0x0024

PC: 0x1d, opcode: PUSH1 0x00

PC: 0x1f, opcode: CODECOPY

PC: 0x20, opcode: PUSH1 0x00

PC: 0x22, opcode: RETURN

PC: 0x23, opcode: INVALID

PC: 0x24, opcode: PUSH1 0x80

....

Set-up the stack to prepare for the CODECOPY opcode and RETURN opcode.

Stack now looks like:

stack[0]: 0

stack[1]: 0x24

stack[2]: 0xC1

stack[3]: 0xC1

# ByteCode

PC: 0x10, opcode: JUMPDEST  
PC: 0x11, opcode: POP  
PC: 0x12, opcode: PUSH1 0x03  
PC: 0x14, opcode: PUSH1 0x01  
PC: 0x16, opcode: SSTORE  
PC: 0x17, opcode: PUSH1 0xc1  
PC: 0x19, opcode: DUP1  
PC: 0x1a, opcode: PUSH2 0x0024  
PC: 0x1d, opcode: PUSH1 0x00  
PC: 0x1f, opcode: CODECOPY  
PC: 0x20, opcode: PUSH1 0x00  
PC: 0x22, opcode: RETURN  
PC: 0x23, opcode: INVALID  
PC: 0x24, opcode: PUSH1 0x80  
....

## CODECOPY

Copy from code that is executing to memory.  
stack[0] is memory offset to write to.  
stack[1] is code offset to read from.  
stack[2] is the length in bytes to copy.

Stack *looked* like:

stack[0]: 0  
stack[1]: 0x24  
stack[2]: 0xC1  
stack[3]: 0xC1

# ByteCode

PC: 0x10, opcode: JUMPDEST  
PC: 0x11, opcode: POP  
PC: 0x12, opcode: PUSH1 0x03  
PC: 0x14, opcode: PUSH1 0x01  
PC: 0x16, opcode: SSTORE  
PC: 0x17, opcode: PUSH1 0xc1  
PC: 0x19, opcode: DUP1  
PC: 0x1a, opcode: PUSH2 0x0024  
PC: 0x1d, opcode: PUSH1 0x00  
PC: 0x1f, opcode: CODECOPY  
PC: 0x20, opcode: PUSH1 0x00  
PC: 0x22, opcode: RETURN  
PC: 0x23, opcode: INVALID  
PC: 0x24, opcode: PUSH1 0x80  
....

## CODECOPY

Copy from code that is executing to memory.  
stack[0] is memory offset to write to.  
stack[1] is code offset to read from.  
stack[2] is the length in bytes to copy.

Stack now looks like:  
stack[3]: 0xC1

# ByteCode

PC: 0x10, opcode: JUMPDEST

PC: 0x11, opcode: POP

PC: 0x12, opcode: PUSH1 0x03

PC: 0x14, opcode: PUSH1 0x01

PC: 0x16, opcode: SSTORE

PC: 0x17, opcode: PUSH1 0xc1

PC: 0x19, opcode: DUP1

PC: 0x1a, opcode: PUSH2 0x0024

PC: 0x1d, opcode: PUSH1 0x00

PC: 0x1f, opcode: CODECOPY

PC: 0x20, opcode: PUSH1 0x00

PC: 0x22, opcode: RETURN

PC: 0x23, opcode: INVALID

PC: 0x24, opcode: PUSH1 0x80

....

**Copy the contract's code that will be on the blockchain to memory.**

The start offset of the code is 0x24.

The length is 0xC1.

Copy to memory offset 0x00.

# ByteCode

PC: 0x10, opcode: JUMPDEST  
PC: 0x11, opcode: POP  
PC: 0x12, opcode: PUSH1 0x03  
PC: 0x14, opcode: PUSH1 0x01  
PC: 0x16, opcode: SSTORE  
PC: 0x17, opcode: PUSH1 0xc1  
PC: 0x19, opcode: DUP1  
PC: 0x1a, opcode: PUSH2 0x0024  
PC: 0x1d, opcode: PUSH1 0x00  
PC: 0x1f, opcode: CODECOPY  
PC: 0x20, opcode: PUSH1 0x00  
PC: 0x22, opcode: RETURN  
PC: 0x23, opcode: INVALID  
PC: 0x24, opcode: PUSH1 0x80  
....

## RETURN

End execution, return a result and indicate successful execution.

stack[0] is the starting offset (in memory) of the result.

stack[1] is the ending offset (in memory) of the result.

Stack *looked* like:

stack[0]: 0

stack[1]: 0xC1

# ByteCode

```
PC: 0x10, opcode: JUMPDEST  
PC: 0x11, opcode: POP  
PC: 0x12, opcode: PUSH1 0x03  
PC: 0x14, opcode: PUSH1 0x01  
PC: 0x16, opcode: SSTORE  
PC: 0x17, opcode: PUSH1 0xc1  
PC: 0x19, opcode: DUP1  
PC: 0x1a, opcode: PUSH2 0x0024  
PC: 0x1d, opcode: PUSH1 0x00  
PC: 0x1f, opcode: CODECOPY  
PC: 0x20, opcode: PUSH1 0x00  
PC: 0x22, opcode: RETURN  
PC: 0x23, opcode: INVALID  
PC: 0x24, opcode: PUSH1 0x80  
....
```

For contract deployment:

Store the return result into deployed contract's address.

In the example, the 193bytes (0xc1) of code at program counter offset 0x24 was copied to memory [0x0...0xc1] and is the code you'll find at that contract's address.

# ByteCode

```
PC: 0x10, opcode: JUMPDEST  
PC: 0x11, opcode: POP  
PC: 0x12, opcode: PUSH1 0x03  
PC: 0x14, opcode: PUSH1 0x01  
PC: 0x16, opcode: SSTORE  
PC: 0x17, opcode: PUSH1 0xc1  
PC: 0x19, opcode: DUP1  
PC: 0x1a, opcode: PUSH2 0x0024  
PC: 0x1d, opcode: PUSH1 0x00  
PC: 0x1f, opcode: CODECOPY  
PC: 0x20, opcode: PUSH1 0x00  
PC: 0x22, opcode: RETURN  
PC: 0x23, opcode: INVALID  
PC: 0x24, opcode: PUSH1 0x80  
....
```

INVALID

Invalid operation marks the end of the init code.

# Init Code Summary

- Set-up the Free Memory Pointer (which wasn't used)
- Cause a REVERT if Wei was sent with the transaction.
- Set-up storage locations with initial non-zero values.
- Return a pointer to the contract code and length of code to be stored on the blockchain.

# Function Calls



# Example

```
pragma solidity >=0.4.23;

contract Simple {
    uint256 public val1;
    uint256 public val2;

    constructor() public {
        val2 = 3;
    }

    function set(uint256 _param) external {
        val1 = _param;
    }
}
```

# Example

```
pragma solidity >=0.4.23;

contract Simple {
    uint256 public val1;
    uint256 public val2;

    constructor() public {
        val2 = 3;
    }

    function set(uint256 _param) external {
        val1 = _param;
    }
}
```

**public** storage variables automatically have “getter” view calls created.

# Example

```
pragma solidity >=0.4.23;

contract Simple {
    uint256 public val1;
    uint256 public val2;

    constructor() public {
        val2 = 3;
    }

    function set(uint256 _param) external {
        val1 = _param;
    }
}
```

This means that there are three functions in this contract, and not just the one function for **set(uint256)**.

# ByteCode

```
PC: 0x0, opcode: PUSH1 0x80  
PC: 0x2, opcode: PUSH1 0x40  
PC: 0x4, opcode: MSTORE
```

```
PC: 0x5, opcode: CALLVALUE  
PC: 0x6, opcode: DUP1  
PC: 0x7, opcode: ISZERO  
PC: 0x8, opcode: PUSH1 0x0f  
PC: 0xa, opcode: JUMPI  
PC: 0xb, opcode: PUSH1 0x00  
PC: 0xd, opcode: DUP1  
PC: 0xe, opcode: REVERT  
PC: 0xf, opcode: JUMPDEST
```

....

Set-up the Free Memory Pointer

# ByteCode

PC: 0x0, opcode: PUSH1 0x80

PC: 0x2, opcode: PUSH1 0x40

PC: 0x4, opcode: MSTORE

PC: 0x5, opcode: CALLVALUE

PC: 0x6, opcode: DUP1

PC: 0x7, opcode: ISZERO

PC: 0x8, opcode: PUSH1 0x0f

PC: 0xa, opcode: JUMPI

PC: 0xb, opcode: PUSH1 0x00

PC: 0xd, opcode: DUP1

PC: 0xe, opcode: REVERT

PC: 0xf, opcode: JUMPDEST

....

Revert if the transaction VALUE isn't zero.

# ByteCode

```
PC: 0xf, opcode: JUMPDEST  
PC: 0x10, opcode: POP  
PC: 0x11, opcode: PUSH1 0x04  
PC: 0x13, opcode: CALLDATASIZE  
PC: 0x14, opcode: LT  
PC: 0x15, opcode: PUSH1 0x3c  
PC: 0x17, opcode: JUMPI  
PC: 0x18, opcode: PUSH1 0x00
```

....

Remove the VALUE from the stack

# ByteCode

PC: 0xf, opcode: JUMPDEST  
PC: 0x10, opcode: POP  
PC: 0x11, opcode: PUSH1 0x04  
**PC: 0x13, opcode: CALLDATASIZE**  
PC: 0x14, opcode: LT  
PC: 0x15, opcode: PUSH1 0x3c  
PC: 0x17, opcode: JUMPI  
PC: 0x18, opcode: PUSH1 0x00

## CALLDATASIZE

Push the size of the transaction data field onto the stack.

# ByteCode

PC: 0xf, opcode: JUMPDEST  
PC: 0x10, opcode: POP  
PC: 0x11, opcode: PUSH1 0x04  
PC: 0x13, opcode: CALLDATASIZE  
**PC: 0x14, opcode: LT**  
PC: 0x15, opcode: PUSH1 0x3c  
PC: 0x17, opcode: JUMPI  
PC: 0x18, opcode: PUSH1 0x00

LT: Less Than

If  $\text{stack}[0] < \text{stack}[1]$ :  
pop  $\text{stack}[0]$  and  $\text{stack}[1]$  off the stack  
push 1 onto the stack  
otherwise  
push 0 onto the stack.

# ByteCode

PC: 0xf, opcode: JUMPDEST

PC: 0x10, opcode: POP

PC: 0x11, opcode: PUSH1 0x04

PC: 0x13, opcode: CALLDATASIZE

PC: 0x14, opcode: LT

PC: 0x15, opcode: PUSH1 0x3c

PC: 0x17, opcode: JUMPI

PC: 0x18, opcode: PUSH1 0x00

....

PC: 0x3c, opcode: JUMPDEST

PC: 0x3d, opcode: PUSH1 0x00

PC: 0x3f, opcode: DUP1

PC: 0x40, opcode: REVERT

Revert if the transaction data field is less than 4 bytes long

# Function Selector

“The first four bytes of the call data for a function call specifies the function to be called. It is the first (left, high-order in big-endian) four bytes of the Keccak-256 hash of the signature of the function. The signature is defined as the canonical expression of the basic prototype without data location specifier, i.e. the function name with the parenthesised list of parameter types. Parameter types are split by a single comma - no spaces are used.”

# ByteCode

PC: 0xf, opcode: JUMPDEST  
PC: 0x10, opcode: POP  
PC: 0x11, opcode: PUSH1 0x04  
PC: 0x13, opcode: CALLDATASIZE  
PC: 0x14, opcode: LT  
PC: 0x15, opcode: PUSH1 0x3c  
PC: 0x17, opcode: JUMPI  
PC: 0x18, opcode: PUSH1 0x00  
....

PC: 0x3c, opcode: JUMPDEST  
PC: 0x3d, opcode: PUSH1 0x00  
PC: 0x3f, opcode: DUP1  
PC: 0x40, opcode: REVERT

Revert if the transaction data field is less than ~~4 bytes long~~ the size of a Function Selector

# "Jump table" for functions

# ByteCode

PC: 0x18, opcode: PUSH1 0x00

PC: 0x1a, opcode: CALLDATALOAD

PC: 0x1b, opcode: PUSH1 0xe0

PC: 0x1d, opcode: SHR

PC: 0x1e, opcode: DUP1

PC: 0x1f, opcode: PUSH4 0x60fe47b1

PC: 0x24, opcode: EQ

PC: 0x25, opcode: PUSH1 0x41

PC: 0x27, opcode: JUMPI

....

CALLDATALOAD

Push onto the stack the 32 bytes (of CALLDATA) starting at offset stack[0].

# ByteCode

PC: 0x18, opcode: PUSH1 0x00

PC: 0x1a, opcode: CALLDATALOAD

PC: 0x1b, opcode: PUSH1 0xe0

PC: 0x1d, opcode: SHR

PC: 0x1e, opcode: DUP1

PC: 0x1f, opcode: PUSH4 0x60fe47b1

PC: 0x24, opcode: EQ

PC: 0x25, opcode: PUSH1 0x41

PC: 0x27, opcode: JUMPI

....

CALLDATALOAD

Push onto the stack the 32 bytes starting at offset 0x00.

# ByteCode

PC: 0x18, opcode: PUSH1 0x00  
PC: 0x1a, opcode: CALLDATALOAD  
PC: 0x1b, opcode: PUSH1 0xe0  
**PC: 0x1d, opcode: SHR**  
PC: 0x1e, opcode: DUP1  
PC: 0x1f, opcode: PUSH4 0x60fe47b1  
PC: 0x24, opcode: EQ  
PC: 0x25, opcode: PUSH1 0x41  
PC: 0x27, opcode: JUMPI  
....

## SHR

Shift stack[1] to the right stack[0] times, pop stack[0] off the stack.

# ByteCode

PC: 0x18, opcode: PUSH1 0x00

PC: 0x1a, opcode: CALLDATALOAD

PC: 0x1b, opcode: PUSH1 0xe0

PC: 0x1d, opcode: SHR

PC: 0x1e, opcode: DUP1

PC: 0x1f, opcode: PUSH4 0x60fe47b1

PC: 0x24, opcode: EQ

PC: 0x25, opcode: PUSH1 0x41

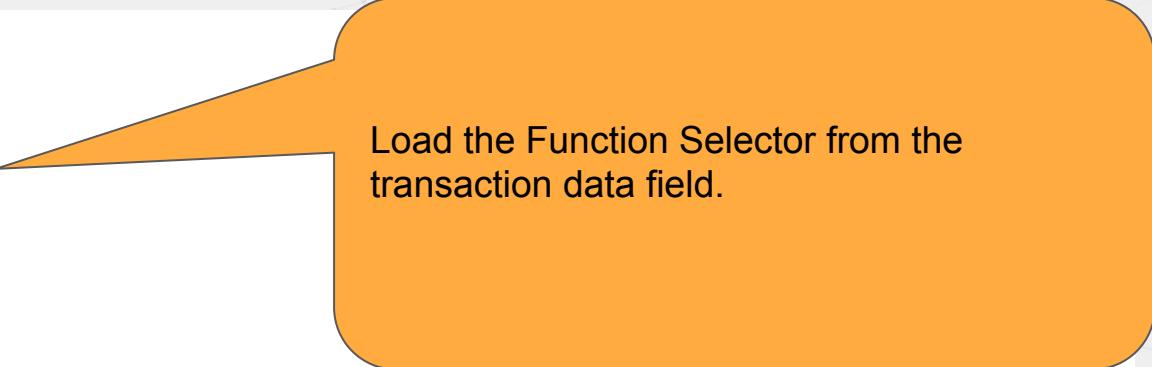
PC: 0x27, opcode: JUMPI

....

Shift the top 32 bits to the bottom 32 bits of the word.

# ByteCode

```
PC: 0x18, opcode: PUSH1 0x00  
PC: 0x1a, opcode: CALLDATALOAD  
PC: 0x1b, opcode: PUSH1 0xe0  
PC: 0x1d, opcode: SHR  
PC: 0x1e, opcode: DUP1  
PC: 0x1f, opcode: PUSH4 0x60fe47b1  
PC: 0x24, opcode: EQ  
PC: 0x25, opcode: PUSH1 0x41  
PC: 0x27, opcode: JUMPI  
....
```



Load the Function Selector from the transaction data field.

# ByteCode

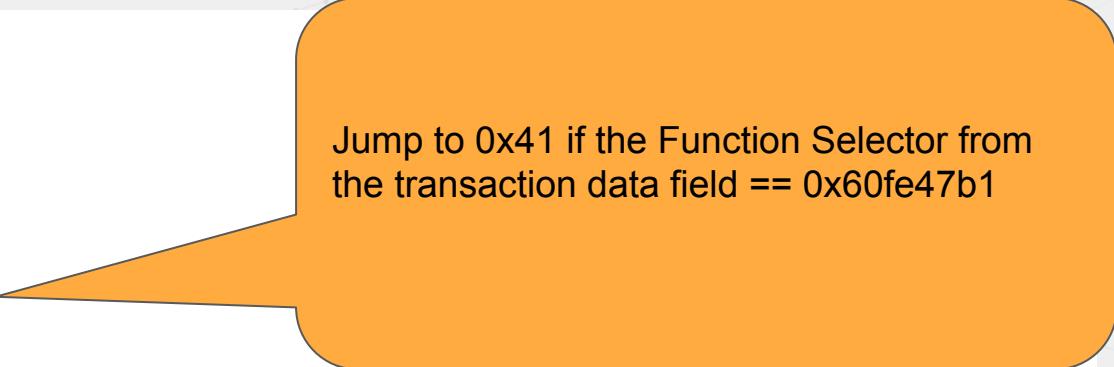
PC: 0x18, opcode: PUSH1 0x00  
PC: 0x1a, opcode: CALLDATALOAD  
PC: 0x1b, opcode: PUSH1 0xe0  
PC: 0x1d, opcode: SHR  
**PC: 0x1e, opcode: DUP1**  
PC: 0x1f, opcode: PUSH4 0x60fe47b1  
PC: 0x24, opcode: EQ  
PC: 0x25, opcode: PUSH1 0x41  
PC: 0x27, opcode: JUMPI  
....

Make a copy of stack[0], the Function Selector from the transaction data field.

# ByteCode

PC: 0x18, opcode: PUSH1 0x00  
PC: 0x1a, opcode: CALLDATALOAD  
PC: 0x1b, opcode: PUSH1 0xe0  
PC: 0x1d, opcode: SHR  
PC: 0x1e, opcode: DUP1  
PC: 0x1f, opcode: PUSH4 0x60fe47b1  
PC: 0x24, opcode: EQ  
PC: 0x25, opcode: PUSH1 0x41  
PC: 0x27, opcode: JUMPI

....



Jump to 0x41 if the Function Selector from the transaction data field == 0x60fe47b1

# ByteCode

PC: 0x18, opcode: PUSH1 0x00

[https://www.4byte.directory/signatures/?bytes4\\_signature=0x60fe47b1](https://www.4byte.directory/signatures/?bytes4_signature=0x60fe47b1)

PC: 0x10, opcode: SHR

Function Selector from

the transaction data field == 0x60fe47b1

PC: 0x1e

Ethereum Function Signature Database

Browse Signatures

Submit Signatures

Submit ABI

Submit Solidity Sou

PC: 0x1f,

API Docs

PC: 0x24

PC: 0x25

PC: 0x27

....

Search Signatures

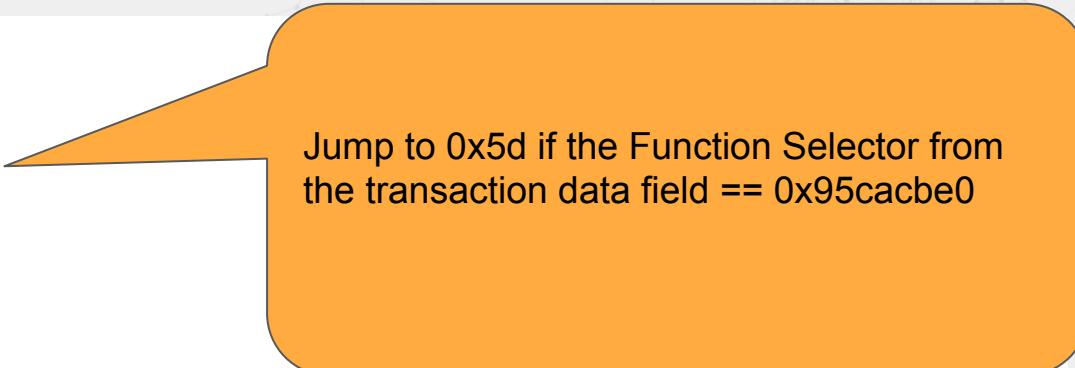
0x60fe47b1

Search

ID	Text Signature	Bytes Signature
824	set(uint256)	0x60fe47b1

# ByteCode

```
PC: 0x28, opcode: DUP1  
PC: 0x29, opcode: PUSH4 0x95cacbe0  
PC: 0x2e, opcode: EQ  
PC: 0x2f, opcode: PUSH1 0x5d  
PC: 0x31, opcode: JUMPI  
  
PC: 0x32, opcode: DUP1  
PC: 0x33, opcode: PUSH4 0xc82fdf36  
PC: 0x38, opcode: EQ  
PC: 0x39, opcode: PUSH1 0x75  
PC: 0x3b, opcode: JUMPI  
PC: 0x3c, opcode: JUMPDEST  
PC: 0x3d, opcode: PUSH1 0x00  
PC: 0x3f, opcode: DUP1  
PC: 0x40, opcode: REVERT  
....
```



Jump to 0x5d if the Function Selector from the transaction data field == 0x95cacbe0

# ByteCode

PC: 0x28, opcode: DUP1

PC: 0x29, opcode: PUSH4 0x95cacbe0

PC: 0x2e, opcode: EQ

PC: 0x2f, opcode: PUSH1 0x5d

PC: 0x31, opcode: JUMPI

PC: 0x32, opcode: DUP1

PC: 0x33, opcode: PUSH4 0xc82fdf36

PC: 0x38, opcode: EQ

PC: 0x39, opcode: PUSH1 0x75

PC: 0x3b, opcode: JUMPI

PC: 0x3c, opcode: JUMPDEST

PC: 0x3d, opcode: PUSH1 0x00

PC: 0x3f, opcode: DUP1

PC: 0x40, opcode: REVERT

....

Jump to 0x75 if the Function Selector from  
the transaction data field == 0xc82fdf36

# ByteCode

PC: 0x28, opcode: DUP1  
PC: 0x29, opcode: PUSH4 0x95cacbe0  
PC: 0x2e, opcode: EQ  
PC: 0x2f, opcode: PUSH1 0x5d  
PC: 0x31, opcode: JUMPI  
PC: 0x32, opcode: DUP1  
PC: 0x33, opcode: PUSH4 0xc82fdf36  
PC: 0x38, opcode: EQ  
PC: 0x39, opcode: PUSH1 0x75  
PC: 0x3b, opcode: JUMPI

PC: 0x3c, opcode: JUMPDEST  
PC: 0x3d, opcode: PUSH1 0x00  
PC: 0x3f, opcode: DUP1  
PC: 0x40, opcode: REVERT

....

If the Function Selector from the transaction data field doesn't equal any of the functions then Revert.

# ByteCode

PC: 0x28, opcode: DUP1

PC: 0x29, opcode: PUSH4 0x95cacbe0

PC: 0x2e, opcode: EQ

PC: 0x2f, opcode: PUSH1 0x5d

PC: 0x31, opcode: JUMPI

PC: 0x32, opcode: DUP1

PC: 0x33, opcode: PUSH4 0xc82fdf36

PC: 0x38, opcode: EQ

PC: 0x39, opcode: PUSH1 0x75

PC: 0x3b, opcode: JUMPI

PC: 0x3c, opcode: JUMPDEST

PC: 0x3d, opcode: PUSH1 0x00

PC: 0x3f, opcode: DUP1

PC: 0x40, opcode: REVERT

....

From Ben Edgington:

Fun fact - Solidity orders its function selectors by numerical value. If you choose names for your frequently used functions that hash to a low values they are checked first and you can save some (tiny amount of) gas :-)

However... once you have more than about four functions, the code will do a "binary search" of the Function Selectors.

The upshot: look at the generated bytecode if you want to do this type of thing to ensure the result is what you expect.



# set(uint256)

# Example

```
pragma solidity >=0.4.23;

contract Simple {
    uint256 public val1;
    uint256 public val2;

    constructor() public {
        val2 = 3;
    }

    function set(uint256 _param) external {
        val1 = _param;
    }
}
```

Focusing on `set(uint256)`

# ByteCode

PC: 0x41, opcode: JUMPDEST  
PC: 0x42, opcode: PUSH1 0x5b  
PC: 0x44, opcode: PUSH1 0x04  
PC: 0x46, opcode: DUP1  
PC: 0x47, opcode: CALLDATASIZE  
PC: 0x48, opcode: SUB  
PC: 0x49, opcode: PUSH1 0x20  
PC: 0x4b, opcode: DUP2  
PC: 0x4c, opcode: LT  
PC: 0x4d, opcode: ISZERO  
PC: 0x4e, opcode: PUSH1 0x55  
PC: 0x50, opcode: JUMPI  
PC: 0x51, opcode: PUSH1 0x00  
PC: 0x53, opcode: DUP1  
PC: 0x54, opcode: REVERT

...

Put a “return address” onto the stack.

This will be used later.

# ByteCode

PC: 0x41, opcode: JUMPDEST

PC: 0x42, opcode: PUSH1 0x5b

PC: 0x44, opcode: PUSH1 0x04

PC: 0x46, opcode: DUP1

PC: 0x47, opcode: CALLDATASIZE

PC: 0x48, opcode: SUB

PC: 0x49, opcode: PUSH1 0x20

PC: 0x4b, opcode: DUP2

PC: 0x4c, opcode: LT

PC: 0x4d, opcode: ISZERO

PC: 0x4e, opcode: PUSH1 0x55

PC: 0x50, opcode: JUMPI

PC: 0x51, opcode: PUSH1 0x00

PC: 0x53, opcode: DUP1

PC: 0x54, opcode: REVERT

...

stack[0] = CALLDATASIZE - 4

That is, the call data size less the length of the Function Selector.

# ByteCode

PC: 0x41, opcode: JUMPDEST  
PC: 0x42, opcode: PUSH1 0x5b  
PC: 0x44, opcode: PUSH1 0x04  
PC: 0x46, opcode: DUP1  
PC: 0x47, opcode: CALLDATASIZE  
PC: 0x48, opcode: SUB  
PC: 0x49, opcode: PUSH1 0x20  
PC: 0x4b, opcode: DUP2  
PC: 0x4c, opcode: LT  
PC: 0x4d, opcode: ISZERO  
PC: 0x4e, opcode: PUSH1 0x55  
PC: 0x50, opcode: JUMPI  
PC: 0x51, opcode: PUSH1 0x00  
PC: 0x53, opcode: DUP1  
PC: 0x54, opcode: REVERT

NOTE: LT followed by ISZERO means  
NOT(LT) which is the same as  
Greater Than Or Equal To.

# ByteCode

PC: 0x41, opcode: JUMPDEST  
PC: 0x42, opcode: PUSH1 0x5b  
PC: 0x44, opcode: PUSH1 0x04  
PC: 0x46, opcode: DUP1  
PC: 0x47, opcode: CALLDATASIZE  
PC: 0x48, opcode: SUB  
PC: 0x49, opcode: PUSH1 0x20  
PC: 0x4b, opcode: DUP2  
PC: 0x4c, opcode: LT  
PC: 0x4d, opcode: ISZERO  
PC: 0x4e, opcode: PUSH1 0x55  
PC: 0x50, opcode: JUMPI  
PC: 0x51, opcode: PUSH1 0x00  
PC: 0x53, opcode: DUP1  
PC: 0x54, opcode: REVERT

Revert if the  $(CALLDATASIZE - 4) <$  the size of a uint256 (32 bytes = 0x20 bytes)

# ByteCode

PC: 0x55, opcode: JUMPDEST

PC: 0x56, opcode: POP

PC: 0x57, opcode: CALLDATALOAD

PC: 0x58, opcode: PUSH1 0x7b

PC: 0x5a, opcode: JUMP

PC: 0x5b, opcode: JUMPDEST

PC: 0x5c, opcode: STOP

...

PC: 0x7b, opcode: JUMPDEST

PC: 0x7c, opcode: PUSH1 0x00

PC: 0x7e, opcode: SSTORE

PC: 0x7f, opcode: JUMP

Stack after the POP is:  
stack[0]: 0x04  
stack[1]: 0x5b

# ByteCode

PC: 0x55, opcode: JUMPDEST

PC: 0x56, opcode: POP

PC: 0x57, opcode: CALLDATALOAD

PC: 0x58, opcode: PUSH1 0x7b

PC: 0x5a, opcode: JUMP

PC: 0x5b, opcode: JUMPDEST

PC: 0x5c, opcode: STOP

...

PC: 0x7b, opcode: JUMPDEST

PC: 0x7c, opcode: PUSH1 0x00

PC: 0x7e, opcode: SSTORE

PC: 0x7f, opcode: JUMP

Push the word at offset 0x04 in the transaction data field onto the stack and jump to 0x7b.

# ByteCode

PC: 0x55, opcode: JUMPDEST

PC: 0x56, opcode: POP

PC: 0x57, opcode: CALLDATALOAD

PC: 0x58, opcode: PUSH1 0x7b

PC: 0x5a, opcode: JUMP

PC: 0x5b, opcode: JUMPDEST

PC: 0x5c, opcode: STOP

...

PC: 0x7b, opcode: JUMPDEST

PC: 0x7c, opcode: PUSH1 0x00

PC: 0x7e, opcode: SSTORE

PC: 0x7f, opcode: JUMP

Push **\_param** onto the stack and jump to 0x7b.

# ByteCode

PC: 0x55, opcode: JUMPDEST  
PC: 0x56, opcode: POP  
PC: 0x57, opcode: CALLDATALOAD  
PC: 0x58, opcode: PUSH1 0x7b  
PC: 0x5a, opcode: JUMP  
PC: 0x5b, opcode: JUMPDEST  
PC: 0x5c, opcode: STOP  
  
...

PC: 0x7b, opcode: JUMPDEST  
PC: 0x7c, opcode: PUSH1 0x00  
PC: 0x7e, opcode: SSTORE  
PC: 0x7f, opcode: JUMP

Store **\_param** to storage location 0x00.

Stack now contains:  
stack[0]: 0x5b

# ByteCode

PC: 0x55, opcode: JUMPDEST  
PC: 0x56, opcode: POP  
PC: 0x57, opcode: CALLDATALOAD  
PC: 0x58, opcode: PUSH1 0x7b  
PC: 0x5a, opcode: JUMP  
PC: 0x5b, opcode: JUMPDEST  
PC: 0x5c, opcode: STOP  
...  
PC: 0x7b, opcode: JUMPDEST  
PC: 0x7c, opcode: PUSH1 0x00  
PC: 0x7e, opcode: SSTORE  
PC: 0x7f, opcode: JUMP

Jump to the location set-up earlier: 0x5b.

# ByteCode

PC: 0x55, opcode: JUMPDEST

PC: 0x56, opcode: POP

PC: 0x57, opcode: CALLDATALOAD

PC: 0x58, opcode: PUSH1 0x7b

PC: 0x5a, opcode: JUMP

PC: 0x5b, opcode: JUMPDEST

PC: 0x5c, opcode: STOP

...

PC: 0x7b, opcode: JUMPDEST

PC: 0x7c, opcode: PUSH1 0x00

PC: 0x7e, opcode: SSTORE

PC: 0x7f, opcode: JUMP

STOP

End execution, indicate execution success,  
and don't return any data.

# val2() view returns (uint256)

# Example

```
pragma solidity >=0.4.23;
```

```
contract Simple {  
    uint256 public val1;  
    uint256 public val2;
```

```
constructor() public {  
    val2 = 3;  
}
```

```
function set(uint256 _param) external {  
    val1 = _param;  
}
```

Focusing on val2() view returns (uint256)

# ByteCode

PC: 0x5d, opcode: JUMPDEST

PC: 0x5e, opcode: PUSH1 0x63

PC: 0x60, opcode: PUSH1 0x80

PC: 0x62, opcode: JUMP

...

PC: 0x80, opcode: JUMPDEST

PC: 0x81, opcode: PUSH1 0x01

PC: 0x83, opcode: SLOAD

PC: 0x84, opcode: DUP2

PC: 0x85, opcode: JUMP

Set-up “return address”

stack[0] = 0x63

# ByteCode

PC: 0x5d, opcode: JUMPDEST  
PC: 0x5e, opcode: PUSH1 0x63  
PC: 0x60, opcode: PUSH1 0x80  
PC: 0x62, opcode: JUMP  
...  
PC: 0x80, opcode: JUMPDEST  
PC: 0x81, opcode: PUSH1 0x01  
PC: 0x83, opcode: SLOAD  
PC: 0x84, opcode: DUP2  
PC: 0x85, opcode: JUMP

**val2** is at storage location 0x01.

DUP2 is to duplicate 2nd stack item i.e stack[1]  
stack[0] = 0x63  
stack[1] = val2  
stack[2] = 0x63

So at PC 0x85 we JUMP to 0x63

# ByteCode

PC: 0x63, opcode: JUMPDEST

PC: 0x64, opcode: PUSH1 0x40

PC: 0x66, opcode: DUP1

PC: 0x67, opcode: MLOAD

PC: 0x68, opcode: SWAP2

PC: 0x69, opcode: DUP3

PC: 0x6a, opcode: MSTORE

PC: 0x6b, opcode: MLOAD

PC: 0x6c, opcode: SWAP1

PC: 0x6d, opcode: DUP2

PC: 0x6e, opcode: SWAP1

PC: 0x6f, opcode: SUB

PC: 0x70, opcode: PUSH1 0x20

PC: 0x72, opcode: ADD

PC: 0x73, opcode: SWAP1

PC: 0x74, opcode: RETURN

Stack now contains:  
stack[0]: val2  
stack[1]: 0x63

# ByteCode

PC: 0x63, opcode: JUMPDEST

PC: 0x64, opcode: PUSH1 0x40

PC: 0x66, opcode: DUP1

PC: 0x67, opcode: MLOAD

PC: 0x68, opcode: SWAP2

PC: 0x69, opcode: DUP3

PC: 0x6a, opcode: MSTORE

PC: 0x6b, opcode: MLOAD

PC: 0x6c, opcode: SWAP1

PC: 0x6d, opcode: DUP2

PC: 0x6e, opcode: SWAP1

PC: 0x6f, opcode: SUB

PC: 0x70, opcode: PUSH1 0x20

PC: 0x72, opcode: ADD

PC: 0x73, opcode: SWAP1

PC: 0x74, opcode: RETURN

Load the Free Memory Pointer from location 0x40.

Stack now contains:

stack[0]: 0x80

stack[1]: 0x40

stack[2]: val2

stack[3]: 0x63

# ByteCode

PC: 0x63, opcode: JUMPDEST  
PC: 0x64, opcode: PUSH1 0x40  
PC: 0x66, opcode: DUP1  
PC: 0x67, opcode: MLOAD  
PC: 0x68, opcode: SWAP2  
PC: 0x69, opcode: DUP3  
PC: 0x6a, opcode: MSTORE  
PC: 0x6b, opcode: MLOAD  
PC: 0x6c, opcode: SWAP1  
PC: 0x6d, opcode: DUP2  
PC: 0x6e, opcode: SWAP1  
PC: 0x6f, opcode: SUB  
PC: 0x70, opcode: PUSH1 0x20  
PC: 0x72, opcode: ADD  
PC: 0x73, opcode: SWAP1  
PC: 0x74, opcode: RETURN

SWAP2: Swap stack[0] and stack[2]

Stack now contains:

stack[0]: val2  
stack[1]: 0x40  
stack[2]: 0x80  
stack[3]: 0x63

# ByteCode

PC: 0x63, opcode: JUMPDEST  
PC: 0x64, opcode: PUSH1 0x40  
PC: 0x66, opcode: DUP1  
PC: 0x67, opcode: MLOAD  
PC: 0x68, opcode: SWAP2  
**PC: 0x69, opcode: DUP3**  
PC: 0x6a, opcode: MSTORE  
PC: 0x6b, opcode: MLOAD  
PC: 0x6c, opcode: SWAP1  
PC: 0x6d, opcode: DUP2  
PC: 0x6e, opcode: SWAP1  
PC: 0x6f, opcode: SUB  
PC: 0x70, opcode: PUSH1 0x20  
PC: 0x72, opcode: ADD  
PC: 0x73, opcode: SWAP1  
PC: 0x74, opcode: RETURN

DUP3: Push a copy of stack[2] onto the stack

Stack now contains:

stack[0]: 0x80  
stack[1]: val2  
stack[2]: 0x40  
stack[3]: 0x80  
stack[4]: 0x63

# ByteCode

PC: 0x63, opcode: JUMPDEST  
PC: 0x64, opcode: PUSH1 0x40  
PC: 0x66, opcode: DUP1  
PC: 0x67, opcode: MLOAD  
PC: 0x68, opcode: SWAP2  
PC: 0x69, opcode: DUP3  
PC: 0x6a, opcode: MSTORE  
PC: 0x6b, opcode: MLOAD  
PC: 0x6c, opcode: SWAP1  
PC: 0x6d, opcode: DUP2  
PC: 0x6e, opcode: SWAP1  
PC: 0x6f, opcode: SUB  
PC: 0x70, opcode: PUSH1 0x20  
PC: 0x72, opcode: ADD  
PC: 0x73, opcode: SWAP1  
PC: 0x74, opcode: RETURN

MSTORE: Store stack[1] in memory location specified by stack[0].

$\text{memory}[0x80] = \text{val2}$

Stack now contains:  
stack[0]: 0x40  
stack[1]: 0x80  
stack[2]: 0x63

# ByteCode

PC: 0x63, opcode: JUMPDEST  
PC: 0x64, opcode: PUSH1 0x40  
PC: 0x66, opcode: DUP1  
PC: 0x67, opcode: MLOAD  
PC: 0x68, opcode: SWAP2  
PC: 0x69, opcode: DUP3  
PC: 0x6a, opcode: MSTORE  
PC: 0x6b, opcode: MLOAD  
PC: 0x6c, opcode: SWAP1  
PC: 0x6d, opcode: DUP2  
PC: 0x6e, opcode: SWAP1  
PC: 0x6f, opcode: SUB  
PC: 0x70, opcode: PUSH1 0x20  
PC: 0x72, opcode: ADD  
PC: 0x73, opcode: SWAP1  
PC: 0x74, opcode: RETURN

MLOAD: Push onto the stack the value at memory location specified by stack[0].

stack[0] = memory[0x40]

Stack now contains:

stack[0]: 0x80

stack[1]: 0x80

stack[2]: 0x63

# ByteCode

PC: 0x63, opcode: JUMPDEST  
PC: 0x64, opcode: PUSH1 0x40  
PC: 0x66, opcode: DUP1  
PC: 0x67, opcode: MLOAD  
PC: 0x68, opcode: SWAP2  
PC: 0x69, opcode: DUP3  
PC: 0x6a, opcode: MSTORE  
PC: 0x6b, opcode: MLOAD  
PC: 0x6c, opcode: SWAP1  
PC: 0x6d, opcode: DUP2  
PC: 0x6e, opcode: SWAP1  
PC: 0x6f, opcode: SUB  
PC: 0x70, opcode: PUSH1 0x20  
PC: 0x72, opcode: ADD  
PC: 0x73, opcode: SWAP1  
PC: 0x74, opcode: RETURN

SWAP1: Swap stack[0] and stack[1].

Stack now contains:

stack[0]: 0x80

stack[1]: 0x80

stack[2]: 0x63

# ByteCode

PC: 0x63, opcode: JUMPDEST  
PC: 0x64, opcode: PUSH1 0x40  
PC: 0x66, opcode: DUP1  
PC: 0x67, opcode: MLOAD  
PC: 0x68, opcode: SWAP2  
PC: 0x69, opcode: DUP3  
PC: 0x6a, opcode: MSTORE  
PC: 0x6b, opcode: MLOAD  
PC: 0x6c, opcode: SWAP1  
**PC: 0x6d, opcode: DUP2**  
PC: 0x6e, opcode: SWAP1  
PC: 0x6f, opcode: SUB  
PC: 0x70, opcode: PUSH1 0x20  
PC: 0x72, opcode: ADD  
PC: 0x73, opcode: SWAP1  
PC: 0x74, opcode: RETURN

DUP2: Push a copy of stack[1] onto the stack.

Stack now contains:  
stack[0]: 0x80  
stack[1]: 0x80  
stack[2]: 0x80  
stack[3]: 0x63

# ByteCode

PC: 0x63, opcode: JUMPDEST  
PC: 0x64, opcode: PUSH1 0x40  
PC: 0x66, opcode: DUP1  
PC: 0x67, opcode: MLOAD  
PC: 0x68, opcode: SWAP2  
PC: 0x69, opcode: DUP3  
PC: 0x6a, opcode: MSTORE  
PC: 0x6b, opcode: MLOAD  
PC: 0x6c, opcode: SWAP1  
PC: 0x6d, opcode: DUP2  
PC: 0x6e, opcode: SWAP1  
PC: 0x6f, opcode: SUB  
PC: 0x70, opcode: PUSH1 0x20  
PC: 0x72, opcode: ADD  
PC: 0x73, opcode: SWAP1  
PC: 0x74, opcode: RETURN

SWAP1: Swap stack[0] and stack[1].

Stack now contains:

stack[0]: 0x80

stack[1]: 0x80

stack[2]: 0x80

stack[3]: 0x63

# ByteCode

PC: 0x63, opcode: JUMPDEST  
PC: 0x64, opcode: PUSH1 0x40  
PC: 0x66, opcode: DUP1  
PC: 0x67, opcode: MLOAD  
PC: 0x68, opcode: SWAP2  
PC: 0x69, opcode: DUP3  
PC: 0x6a, opcode: MSTORE  
PC: 0x6b, opcode: MLOAD  
PC: 0x6c, opcode: SWAP1  
PC: 0x6d, opcode: DUP2  
PC: 0x6e, opcode: SWAP1  
PC: 0x6f, opcode: SUB

PC: 0x70, opcode: PUSH1 0x20  
PC: 0x72, opcode: ADD  
PC: 0x73, opcode: SWAP1  
PC: 0x74, opcode: RETURN

SUB:  $\text{stack}[0] = \text{stack}[0] - \text{stack}[1]$

Stack now contains:  
 $\text{stack}[0]: 0x00$   
 $\text{stack}[1]: 0x80$   
 $\text{stack}[2]: 0x63$

# ByteCode

PC: 0x63, opcode: JUMPDEST  
PC: 0x64, opcode: PUSH1 0x40  
PC: 0x66, opcode: DUP1  
PC: 0x67, opcode: MLOAD  
PC: 0x68, opcode: SWAP2  
PC: 0x69, opcode: DUP3  
PC: 0x6a, opcode: MSTORE  
PC: 0x6b, opcode: MLOAD  
PC: 0x6c, opcode: SWAP1  
PC: 0x6d, opcode: DUP2  
PC: 0x6e, opcode: SWAP1  
PC: 0x6f, opcode: SUB  
**PC: 0x70, opcode: PUSH1 0x20**  
PC: 0x72, opcode: ADD  
PC: 0x73, opcode: SWAP1  
PC: 0x74, opcode: RETURN

Push length of return value: uint256

Stack now contains:

stack[0]: 0x20

stack[1]: 0x00

stack[2]: 0x80

stack[3]: 0x63

# ByteCode

PC: 0x63, opcode: JUMPDEST  
PC: 0x64, opcode: PUSH1 0x40  
PC: 0x66, opcode: DUP1  
PC: 0x67, opcode: MLOAD  
PC: 0x68, opcode: SWAP2  
PC: 0x69, opcode: DUP3  
PC: 0x6a, opcode: MSTORE  
PC: 0x6b, opcode: MLOAD  
PC: 0x6c, opcode: SWAP1  
PC: 0x6d, opcode: DUP2  
PC: 0x6e, opcode: SWAP1  
PC: 0x6f, opcode: SUB  
PC: 0x70, opcode: PUSH1 0x20

PC: 0x72, opcode: ADD

PC: 0x73, opcode: SWAP1  
PC: 0x74, opcode: RETURN

ADD:  $\text{stack}[0] = \text{stack}[0] + \text{stack}[1]$

Stack now contains:

stack[0]: 0x20

stack[1]: 0x80

stack[2]: 0x63

# ByteCode

PC: 0x63, opcode: JUMPDEST  
PC: 0x64, opcode: PUSH1 0x40  
PC: 0x66, opcode: DUP1  
PC: 0x67, opcode: MLOAD  
PC: 0x68, opcode: SWAP2  
PC: 0x69, opcode: DUP3  
PC: 0x6a, opcode: MSTORE  
PC: 0x6b, opcode: MLOAD  
PC: 0x6c, opcode: SWAP1  
PC: 0x6d, opcode: DUP2  
PC: 0x6e, opcode: SWAP1  
PC: 0x6f, opcode: SUB  
PC: 0x70, opcode: PUSH1 0x20  
PC: 0x72, opcode: ADD  
PC: 0x73, opcode: SWAP1  
PC: 0x74, opcode: RETURN

SWAP1: Swap stack[0] and stack[1]

Stack now contains:

stack[0]: 0x80

stack[1]: 0x20

stack[2]: 0x63

# ByteCode

PC: 0x63, opcode: JUMPDEST  
PC: 0x64, opcode: PUSH1 0x40  
PC: 0x66, opcode: DUP1  
PC: 0x67, opcode: MLOAD  
PC: 0x68, opcode: SWAP2  
PC: 0x69, opcode: DUP3  
PC: 0x6a, opcode: MSTORE  
PC: 0x6b, opcode: MLOAD  
PC: 0x6c, opcode: SWAP1  
PC: 0x6d, opcode: DUP2  
PC: 0x6e, opcode: SWAP1  
PC: 0x6f, opcode: SUB  
PC: 0x70, opcode: PUSH1 0x20  
PC: 0x72, opcode: ADD  
PC: 0x73, opcode: SWAP1  
PC: 0x74, opcode: RETURN

## RETURN

End execution, return a result and indicate successful execution.

stack[0] is the offset of the result.  
stack[1] is the length of the result.

Stack **contained**:

stack[0]: 0x80  
stack[1]: 0x20  
stack[2]: 0x63

# ByteCode

PC: 0x63, opcode: JUMPDEST  
PC: 0x64, opcode: PUSH1 0x40  
PC: 0x66, opcode: DUP1  
PC: 0x67, opcode: MLOAD  
PC: 0x68, opcode: SWAP2  
PC: 0x69, opcode: DUP3  
PC: 0x6a, opcode: MSTORE  
PC: 0x6b, opcode: MLOAD  
PC: 0x6c, opcode: SWAP1  
PC: 0x6d, opcode: DUP2  
PC: 0x6e, opcode: SWAP1  
PC: 0x6f, opcode: SUB  
PC: 0x70, opcode: PUSH1 0x20  
PC: 0x72, opcode: ADD  
PC: 0x73, opcode: SWAP1  
PC: 0x74, opcode: RETURN

## RETURN

val2 was in memory 0x80 and has length 0x20. (see Slide 104)

stack[0] is the offset of the result.  
stack[1] is the length of the result.

Stack **contained**:

stack[0]: 0x80  
stack[1]: 0x20  
stack[2]: 0x63

# ByteCode

```
PC: 0x63, opcode: JUMPDEST  
PC: 0x64, opcode: PUSH1 0x40  
PC: 0x66, opcode: DUP1  
PC: 0x67, opcode: MLOAD  
PC: 0x68, opcode: SWAP2  
PC: 0x69, opcode: DUP3  
PC: 0x6a, opcode: MSTORE  
PC: 0x6b, opcode: MLOAD  
PC: 0x6c, opcode: SWAP1  
PC: 0x6d, opcode: DUP2  
PC: 0x6e, opcode: SWAP1  
PC: 0x6f, opcode: SUB  
PC: 0x70, opcode: PUSH1 0x20  
PC: 0x72, opcode: ADD  
PC: 0x73, opcode: SWAP1  
PC: 0x74, opcode: RETURN
```

Put together the return result, a uint256 which was put in memory, and return it.

val2 was loaded from storage (SLOAD), written to memory, and then returned.

# Example

```
pragma solidity >=0.4.23;
```

```
contract Simple {  
    uint256 public val1;  
    uint256 public val2;
```

```
constructor() public {  
    val2 = 3;  
}
```

```
function set(uint256 _param) external {  
    val1 = _param;  
}
```

Focusing on val2() view returns (uint256)

# val1() view returns (uint256)

# Example

```
pragma solidity >=0.4.23;
```

```
contract Simple {  
    uint256 public val1;  
    uint256 public val2;
```

```
constructor() public {  
    val2 = 3;  
}
```

```
function set(uint256 _param) external {  
    val1 = _param;  
}
```

Focusing on val1() view returns (uint256)

# ByteCode

PC: 0x75, opcode: JUMPDEST

PC: 0x76, opcode: PUSH1 0x63

PC: 0x78, opcode: PUSH1 0x86

PC: 0x7a, opcode: JUMP

...

PC: 0x86, opcode: JUMPDEST

PC: 0x87, opcode: PUSH1 0x00

PC: 0x89, opcode: SLOAD

PC: 0x8a, opcode: DUP2

PC: 0x8b, opcode: JUMP

PC: 0x8c, opcode: INVALID

Set-up “return address”

NOTE: This is the same program counter offset as was used for val2() view returns(uint256)

That is: val1() and val2() use the same code fragment to return a uint256 value.

**val1** is at storage location 0x00.

# Payable Fallback Functions

# Example: SimpleFallBack

```
pragma solidity >=0.4.23;

contract SimpleFallBack {
    uint256 public val1;
    uint256 public val2;
    constructor() public {
        val2 = 3;
    }
    function set(uint256 _param) external {
        val1 = _param;
    }
    function () external payable {
    }
}
```

## Fallback Functions:

- Allows Ether to be transferred to a contract as if it were an EOA.
- If someone is sending Ether to this contract and they muck up the transaction data field, it will still accept the transferred Wei, and ignore the transaction data field.

# ByteCode

```
PC: 0x5, opcode: PUSH1 0x04  
PC: 0x7, opcode: CALLDATASIZE  
PC: 0x8, opcode: LT  
PC: 0x9, opcode: PUSH1 0x30  
PC: 0xb, opcode: JUMPI
```

```
....  
PC: 0x13, opcode: PUSH4 0x60fe47b1  
PC: 0x18, opcode: EQ  
PC: 0x19, opcode: PUSH1 0x32  
PC: 0x1b, opcode: JUMPI
```

```
....  
PC: 0x2f, opcode: JUMPI
```

```
PC: 0x30, opcode: JUMPDEST  
PC: 0x31, opcode: STOP
```

If the length of the transaction data field is less than 4 bytes jump to 0x30.

Jump to various locations given Function Selectors.

Falls through to here if the first 4 bytes of the transaction data field didn't match any of the Function Selectors. This is the fallback function: this one is empty so next is STOP.

STOP: indicates successful transaction execution.

# ByteCode

PC: 0x32, opcode: JUMPDEST  
PC: 0x33, opcode: CALLVALUE  
PC: 0x34, opcode: DUP1  
PC: 0x35, opcode: ISZERO  
PC: 0x36, opcode: PUSH1 0x3d  
PC: 0x38, opcode: JUMPI  
PC: 0x39, opcode: PUSH1 0x00  
PC: 0x3b, opcode: DUP1  
PC: 0x3c, opcode: REVERT

Now each function needs to check the transaction value field if the function is not payable.

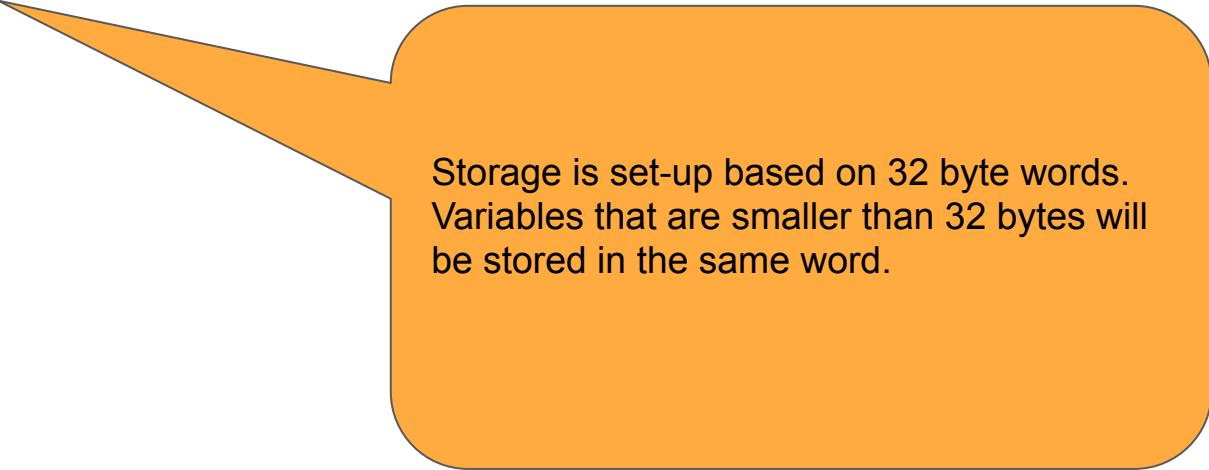
# Storage



# Storage Layout

# Example: Storage Layout

```
contract StorageLayout {  
    byte private valByte;  
    uint256 private valUint256a;  
    uint32 private valUint32;  
    uint64 private valUint64;  
    address private valAddress;  
    uint256 private valUint256b;  
  
    function set() external {  
        valByte = 0x10;  
        valUint256a = 0x11;  
        valUint32 = 0x12;  
        valUint64 = 0x13;  
        valAddress = address(0x14);  
        valUint256b = 0x15;  
    }  
}
```

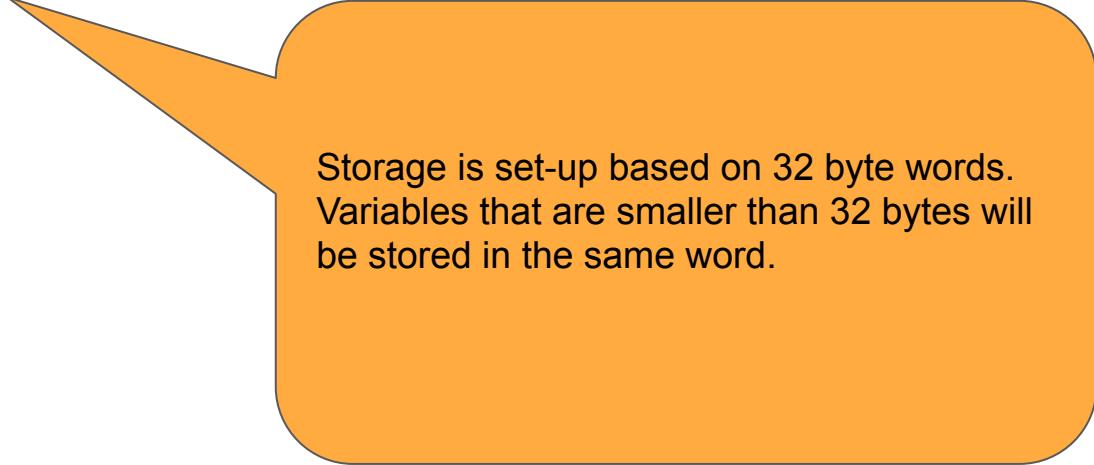


Storage is set-up based on 32 byte words.  
Variables that are smaller than 32 bytes will  
be stored in the same word.

# Example: Storage Layout

```
contract StorageLayout {  
    byte private valByte;  
    uint256 private valUint256a;  
    uint32 private valUint32;  
    uint64 private valUint64;  
    address private valAddress;  
    uint256 private valUint256b;
```

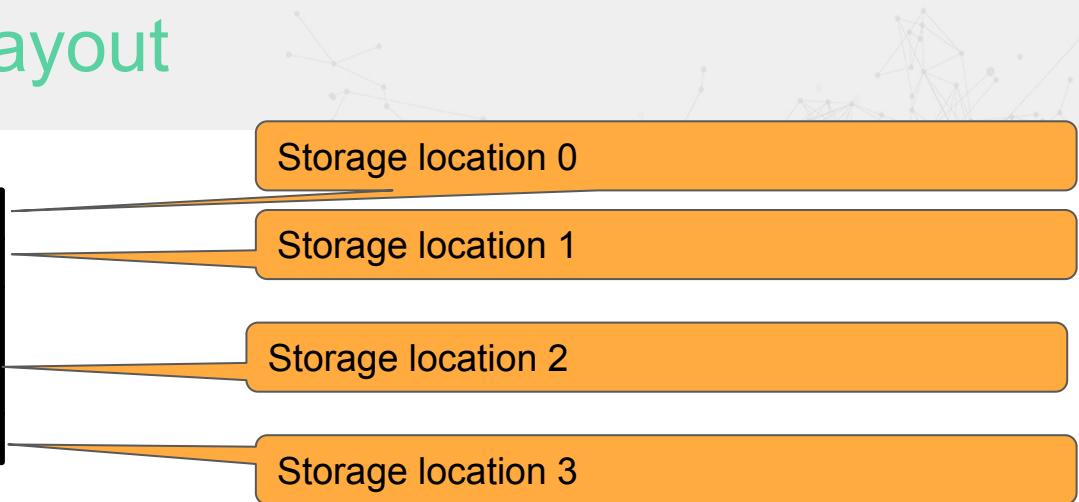
```
function set() external {  
    valByte = 0x10;  
    valUint256a = 0x11;  
    valUint32 = 0x12;  
    valUint64 = 0x13;  
    valAddress = address(0x14);  
    valUint256b = 0x15;  
}
```



Storage is set-up based on 32 byte words.  
Variables that are smaller than 32 bytes will  
be stored in the same word.

# Example: Storage Layout

```
contract StorageLayout {  
    byte private valByte;  
    uint256 private valUint256a;  
    uint32 private valUint32;  
    uint64 private valUint64;  
    address private valAddress;  
    uint256 private valUint256b;  
  
    function set() external {  
        valByte = 0x10;  
        valUint256a = 0x11;  
        valUint32 = 0x12;  
        valUint64 = 0x13;  
        valAddress = address(0x14);  
        valUint256b = 0x15;  
    }  
}
```



# ByteCode

```
PC: 0x35, opcode: JUMPDEST  
PC: 0x36, opcode: PUSH1 0x00  
PC: 0x38, opcode: DUP1  
PC: 0x39, opcode: SLOAD  
PC: 0x3a, opcode: PUSH1 0x10  
PC: 0x3c, opcode: PUSH1 0xff  
PC: 0x3e, opcode: NOT  
PC: 0x3f, opcode: SWAP1  
PC: 0x40, opcode: SWAP2  
PC: 0x41, opcode: AND  
PC: 0x42, opcode: OR  
PC: 0x43, opcode: SWAP1  
PC: 0x44, opcode: SSTORE  
  
...
```

valByte = 0x10;

# ByteCode

PC: 0x35, opcode: JUMPDEST

PC: 0x36, opcode: PUSH1 0x00

PC: 0x38, opcode: DUP1

PC: 0x39, opcode: SLOAD

PC: 0x3a, opcode: PUSH1 0x10

PC: 0x3c, opcode: PUSH1 0xff

PC: 0x3e, opcode: NOT

PC: 0x3f, opcode: SWAP1

PC: 0x40, opcode: SWAP2

PC: 0x41, opcode: AND

PC: 0x42, opcode: OR

PC: 0x43, opcode: SWAP1

PC: 0x44, opcode: SSTORE

Load storage location 0x00

Push the constant we want to set the storage value to.

Create a bit mask 0xFFFFFFFF....FF00

Mask off the bottom byte in the word loaded from the storage location

OR in the value being stored.

Store the result to storage location 0x00

# ByteCode

```
PC: 0x45, opcode: PUSH1 0x11  
PC: 0x47, opcode: PUSH1 0x01  
PC: 0x49, opcode: SSTORE  
....
```

```
valUint256a = 0x11;
```

# ByteCode

```
PC: 0x4a, opcode: PUSH1 0x02  
PC: 0x4c, opcode: DUP1  
PC: 0x4d, opcode: SLOAD  
PC: 0x4e, opcode: PUSH1 0x12  
PC: 0x50, opcode: PUSH4 0xffffffff  
PC: 0x55, opcode: NOT  
PC: 0x56, opcode: SWAP1  
PC: 0x57, opcode: SWAP2  
PC: 0x58, opcode: AND  
PC: 0x59, opcode: OR  
PC: 0x5a, opcode: PUSH12 0xffffffffffffffffffff00000000  
PC: 0x67, opcode: NOT  
PC: 0x68, opcode: AND  
PC: 0x69, opcode: PUSH5 0x1300000000  
PC: 0x6f, opcode: OR  
PC: 0x70, opcode: PUSH12 0xffffffffffffffffffffffffffff  
PC: 0x7d, opcode: AND  
PC: 0x7e, opcode: PUSH1 0x05  
PC: 0x80, opcode: PUSH1 0x62  
PC: 0x82, opcode: SHL  
PC: 0x83, opcode: OR  
PC: 0x84, opcode: SWAP1  
PC: 0x85, opcode: SSTORE
```

For storage location 2:

```
valUint32 = 0x12;  
valUint64 = 0x13;  
valAddress = address(0x14);
```

# ByteCode

PC: 0x4a, opcode: PUSH1 0x02

PC: 0x4c, opcode: DUP1

PC: 0x4d, opcode: SLOAD

PC: 0x4e, opcode: PUSH1 0x12

PC: 0x50, opcode: PUSH4 0xffffffff

PC: 0x55, opcode: NOT

PC: 0x56, opcode: SWAP1

PC: 0x57, opcode: SWAP2

PC: 0x58, opcode: AND

PC: 0x59, opcode: OR

PC: 0x5a, opcode: PUSH12 0xffffffffffffffffffff00000000

PC: 0x67, opcode: NOT

PC: 0x68, opcode: AND

PC: 0x69, opcode: PUSH5 0x1300000000

PC: 0x6f, opcode: OR

PC: 0x70, opcode: PUSH12 0xffffffffffffffffffffffff

PC: 0x7d, opcode: AND

PC: 0x7e, opcode: PUSH1 0x05

PC: 0x80, opcode: PUSH1 0x62

PC: 0x82, opcode: SHL

PC: 0x83, opcode: OR

PC: 0x84, opcode: SWAP1

PC: 0x85, opcode: SSTORE

Load storage location 0x02

Mask in: valUint32 = **0x12**;

Mask in: valUint64 = **0x13**;

Mask in: valAddress = **address(0x14)**;

Store to storage location 0x02

# ByteCode

```
PC: 0x86, opcode: PUSH1 0x15  
PC: 0x88, opcode: PUSH1 0x03  
PC: 0x8a, opcode: SSTORE  
PC: 0x8b, opcode: JUMP
```

```
valUint256b = 0x15;
```

# Summary

- Well laid out Solidity storage variables will result in fewer storage locations.
- Fewer storage locations will save you \$\$\$
- In this example, there was no way to optimize storing valByte and it would have been cheaper to just use a uint256. (The code in Slide 129 is an example of <https://ethereum.stackexchange.com/questions/3067/why-does-uint8-cost-more-gas-than-uint256>)

# Bytes and String

# Bytes and String

- Bytes and String are variable length arrays of bytes that behave the same way.

# Example: Storage Bytes

```
contract StorageBytes {  
    bytes private valBytes;  
  
    function setBytes(byte _val) external {  
        valBytes.push(_val);  
    }  
}
```



# Bytes and String

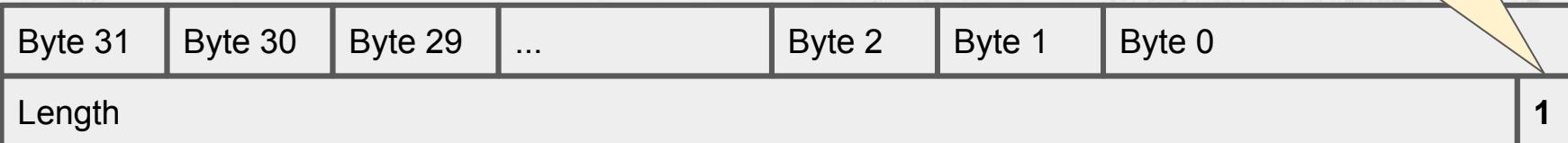
- The bytes is allocated a storage slot.
- For **bytes private valBytes;** this is storage[0]
- For 0 to 31 bytes:



Marker bit indicating Bytes with 31 or fewer bytes

# Bytes and String

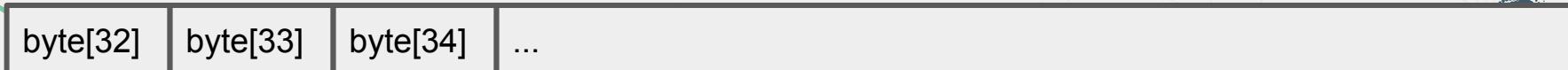
- The bytes is allocated a storage slot.
- For **bytes private** valBytes; this is storage[0]
- For 32 bytes or more:



- At storage location: storage[keccak256(storage slot)]



- At storage location: storage[keccak256(storage slot) + 1]



# ByteCode

```
PC: 0x2d, opcode: JUMPDEST  
PC: 0x2e, opcode: PUSH1 0x51  
PC: 0x30, opcode: PUSH1 0x04  
PC: 0x32, opcode: DUP1  
PC: 0x33, opcode: CALLDATASIZE  
PC: 0x34, opcode: SUB  
PC: 0x35, opcode: PUSH1 0x20  
PC: 0x37, opcode: DUP2  
PC: 0x38, opcode: LT  
PC: 0x39, opcode: ISZERO  
PC: 0x3a, opcode: PUSH1 0x41  
PC: 0x3c, opcode: JUMPI  
PC: 0x3d, opcode: PUSH1 0x00  
PC: 0x3f, opcode: DUP1  
PC: 0x40, opcode: REVERT
```

Program Counter of end code.

Length of transaction data field less size of Function Selector

Revert if the size of **byte\_val** isn't at least 0x20 (32) bytes

All parameters are passed as 32 byte words

# ByteCode

PC: 0x41, opcode: JUMPDEST

PC: 0x42, opcode: POP

PC: 0x43, opcode: CALLDATALOAD

PC: 0x44, opcode: PUSH1 0x01

PC: 0x46, opcode: PUSH1 0x01

PC: 0x48, opcode: PUSH1 0xf8

PC: 0x4a, opcode: SHL

PC: 0x4b, opcode: SUB

PC: 0x4c, opcode: NOT

PC: 0x4d, opcode: AND

PC: 0x4e, opcode: PUSH1 0x53

PC: 0x50, opcode: JUMP

Stack is:

stack[0]: CALLDATASIZE - 0x04

stack[1]: 0x04

stack[2]: 0x51

Load one word of call data from offset 0x04

Create a bit mask 0x00FFFFFF..FF

Mask off **byte\_val**

# ByteCode

PC: 0x53, opcode: JUMPDEST

PC: 0x54, opcode: PUSH1 0x00

PC: 0x56, opcode: DUP2

PC: 0x57, opcode: SWAP1

PC: 0x58, opcode: DUP1

PC: 0x59, opcode: DUP1

PC: 0x5a, opcode: SLOAD

PC: 0x5b, opcode: PUSH1 0x3f

PC: 0x5d, opcode: DUP2

PC: 0x5e, opcode: AND

PC: 0x5f, opcode: DUP1

PC: 0x60, opcode: PUSH1 0x3e

PC: 0x62, opcode: DUP2

PC: 0x63, opcode: EQ

PC: 0x64, opcode: PUSH1 0x84

PC: 0x66, opcode: JUMPI

Stack is:

stack[0]: 0x??00000000..00

stack[1]: 0x51

Load storage location 0x00

Mask off what would be the length if the bytes contains fewer than 31 bytes.

Does **bytes private** valBytes currently contain exactly 31 bytes?

Recall the length is shifted left by 1.

# ByteCode

PC: 0x84, opcode: JUMPDEST

PC: 0x85, opcode: PUSH1 0x00

PC: 0x87, opcode: DUP5

PC: 0x88, opcode: DUP2

PC: 0x89, opcode: MSTORE

PC: 0x8a, opcode: PUSH1 0x20

PC: 0x8c, opcode: SWAP1

PC: 0x8d, opcode: DUP2

PC: 0x8e, opcode: SWAP1

PC: 0x8f, opcode: SHA3

PC: 0x90, opcode: PUSH1 0xff

PC: 0x92, opcode: NOT

PC: 0x93, opcode: DUP6

PC: 0x94, opcode: AND

PC: 0x95, opcode: SWAP1

PC: 0x96, opcode: SSTORE

Stack is:

stack[0]: 0x??00000000..00

stack[1]: 0x??00000000..00

stack[2]: storage location 0x00

stack[3]: 0x00

stack[4]: 0x00

stack[5]: 0x??00000000..00

stack[6]: 0x??00000000..00

stack[7]: 0x51

Memory[0x00] = 0

stack[0] = KECCAK256 from  
memory(stack[0]) stack[1] number of bytes  
stack[0] is 0, stack[1] is 0x20

Mask off the top of the existing storage value,  
removing the length byte

Store the existing bytes into the storage location  
KECCAK256(the storage offset, 0x00).

# ByteCode

```
PC: 0x97, opcode: PUSH1 0x41  
PC: 0x99, opcode: SWAP1  
PC: 0xa, opcode: SWAP5  
PC: 0xb, opcode: SSTORE  
PC: 0xc, opcode: JUMPDEST
```

Store the new length & indicator to storage location 0x00.

Length (0x20) shift left by 1 + 1 to indicate longer than one word.

# ByteCode

```
PC:0x0c, opcode: JUMPDEST  
PC:0x0d, opcode: POP  
PC:0x0e, opcode: POP  
PC:0x0f, opcode: POP  
PC:0x10, opcode: SWAP1  
PC:0x11, opcode: PUSH1 0x01  
PC:0x12, opcode: DUP3  
PC:0x13, opcode: DUP2  
PC:0x14, opcode: SLOAD  
PC:0x15, opcode: PUSH1 0x01  
PC:0x16, opcode: NOT  
PC:0x17, opcode: ISZERO  
PC:0x18, opcode: PUSH1 0x02  
PC:0x19, opcode: DUP1  
PC:0x1a, opcode: SWAP1  
PC:0x1b, opcode: PUSH1 0x00  
PC:0x1c, opcode: MSTORE  
PC:0x1d, opcode: PUSH1 0x20  
PC:0x1e, opcode: PUSH1 0x00  
PC:0x1f, opcode: SHA3  
PC:0x20, opcode: SWAP1  
PC:0x21, opcode: SWAP1  
PC:0x22, opcode: SWAP2  
PC:0x23, opcode: DUP3  
PC:0x24, opcode: DUP3  
PC:0x25, opcode: DIV  
PC:0x26, opcode: ADD  
PC:0x27, opcode: SWAP2  
PC:0x28, opcode: SWAP1  
PC:0x29, opcode: MOD  
PC:0x2a, opcode: JUMPDEST  
PC:0x2b, opcode: SWAP1  
PC:0x2c, opcode: SWAP2  
PC:0x2d, opcode: SWAP3  
PC:0x2e, opcode: SWAP1  
PC:0x2f, opcode: SWAP2  
PC:0x30, opcode: SWAP1  
PC:0x31, opcode: SWAP2  
PC:0x32, opcode: SWAP1  
PC:0x33, opcode: DUP2  
PC:0x34, opcode: PUSH1 0xff  
PC:0x35, opcode: PUSH1 0x01  
PC:0x36, opcode: NOT  
PC:0x37, opcode: AND  
PC:0x38, opcode: SWAP1  
PC:0x39, opcode: SWAP1  
PC:0x3a, opcode: PUSH1 0x01  
PC:0x3b, opcode: PUSH1 0xb  
PC:0x3c, opcode: SHL  
PC:0x3d, opcode: DUP5  
PC:0x3e, opcode: DIV  
PC:0x3f, opcode: MUL  
PC:0x40, opcode: OR  
PC:0x41, opcode: SWAP1  
PC:0x42, opcode: SWAP1  
PC:0x43, opcode: SSTORE  
PC:0x44, opcode: POP  
PC:0x45, opcode: POP  
PC:0x46, opcode: POP  
PC:0x47, opcode: POP  
PC:0x48, opcode: JUMP
```

Insert the **byte\_val** into the bytes array at the right location.

# Arrays

# Example: Storage Arrays

```
contract StorageArrays {  
    uint256[] private arrayUint256;  
    byte[] private arrayByte;  
  
    function setUint256ArrayVal(uint256 _ofs, uint256 _val) external {  
        arrayUint256[_ofs] = _val;  
    }  
  
    function setByteArrayVal(uint256 _ofs, byte _val) external {  
        arrayByte[_ofs] = _val;  
    }  
}
```

# Dynamic Arrays

Values for (**uint256[] private** arrayUint256;) are stored at locations:

storage[keccak256(storage slot number)+key] = value

Note:

- The number of elements in the dynamic array is stored at storage[storage slot number]

# Mappings

# Example: Storage Mappings

```
contract StorageMappings {  
    mapping(uint256 => uint256) private map;  
  
    function setMapVal(uint256 _key, uint256 _val) external {  
        map[_key] = _val;  
    }  
}
```

# Mappings

Values for (**mapping(uint256 => uint256) private map;**) are stored at locations:

`storage[keccak256(key . storage slot number)] = value`

Note:

- “.” is concatenation.
- Nothing is stored at `storage[storage slot number]`

# Mapping of Mapping, Array of Arrays, Arrays of Mappings, Mappings of Arrays, etc

- The rules for working out storage locations are applied recursively for more complex structures such as Mapping of Mapping.
- See [https://solidity.readthedocs.io/en/v0.6.9/internals/layout\\_in\\_storage.html](https://solidity.readthedocs.io/en/v0.6.9/internals/layout_in_storage.html) for more details.

# Memory



# Memory Expansion

Could you write some code that:

Push32 0xFFFFFFFF....FFFFF A BIG NUMBER

Push8 0x00

MSTORE

& hope to crash the EVM by doing out of memory.

# Memory Expansion

But:

- Each word that memory expands costs Gas.
- Requesting a LOT of memory will cause an out of gas error.

# Memory

0x00 - 0x3f	(64 bytes): scratch space for hashing methods
0x40 - 0x5f	(32 bytes): currently allocated memory size (aka. free memory pointer)
0x60 - 0x7f	(32 bytes): zero slot
0x80	Allocated memory starts here. For example: <code>uint[] memory a = new uint[](7);</code>

# CODECOPY and EXTCODECOPY

# CODE COPY

- Code Copy: Copy bytes from this contract to memory.
- Ext Code Copy: Copy bytes from another contract to memory.

## Usage:

- Revert Reason error messages.
- Deploy a contract from this contract.
- Any other static data.

# AuxData / MetaData



# Aux Data / Solidity Metadata

".auxdata" : "a265627a7a723058209832b60d1aa283d2a1ca724b09775bd9232213e63c1e8eaf01c08741fd1b016864736f6c634300050a0032",

- Bytes included at the end of modern Solidity contracts.
- CBOR encoded: <https://tools.ietf.org/html/rfc7049#page-7>
- Indicates:
  - Swarm / IPFS message digest, which is location of the source code.
    - NOTE: You need to upload the code yourself.
  - Compiler name
  - Compiler version
- Allows tools like EtherScan to automatically verify that the deployed contract was compiled from specific source code.

# Aux Data / Solidity Metadata

".auxdata" : "a265627a7a723058209832b60d1aa283d2a1ca724b09775bd9232213e63c1e8eaf01c08741fd1b016864736f6c634300050a0032",

- A2 Map
- 65 6X mean array of characters. 5 is the length
- 627a7a7230 "bzzr0" which means Swarm. This could be IPFS.
- 58
- 20 length of message digest
- 9832b60d1aa283d2a1ca724b09775bd9232213e63c1e8eaf01c08741fd1b0168 message digest
- 64 6X mean array of characters. 4 is the length
- 736f6c63 "solc"
- 43 4X means array. 3 is length.
- 00050a 00,05,0A = version 0.5.10
- 0032 The length of the aux data: 0x32 = 50 bytes

# Aux Data / Solidity Metadata

For more information see:

<https://solidity.readthedocs.io/en/v0.6.9/metadata.html>

# Summary



# Summary

The EVM is a stack based processor that has access to:

- CallData: Transaction parameters.
- Memory: Temporary data store within a transaction.
- Storage: Persistent storage that is part of world state.
- Code: Stores code and static data such as strings.
- Logs: Write-only event log output.

# So much more to cover

- Cross-Contract calls
- Error handling: Assert, Require, Revert and Exceptions
- Events / Logs.
- More on storage: difference between bytes32 and uint256
- More on memory variables.
- Strange contracts from early Ethereum.
- Deep dive on ERC20 contract.
- Deep dive on some of the important contracts on Ethereum MainNet

What else would you like covered in the next talk on this topic?

# Future Talks

- Jun 24: Code Merklization
  - Peter Robinson
- Jul 08: Polynomial Commitments
  - Raghavendra Ramesh
- July 13: Supply Chain on Blockchain Conference
  - Details: <https://scobc.net/>
  - Registrations: <https://www.eventbrite.com.au/e/supply-chain-on-blockchain-conference-2020-tickets-103758121334>
  - Early Bird Registrations (AU\$20): Ends Friday June 12
  - Standard Registrations (AU\$40)
- August 5: EIP-1559 and Escalators: Fixing the Ethereum Fee Market
  - Tim Beiko
  - **NOTE: 9:30 am Brisbane time zone**