Lesson 3 - EVM Deep Dive

EVM Deep Dive

Topics

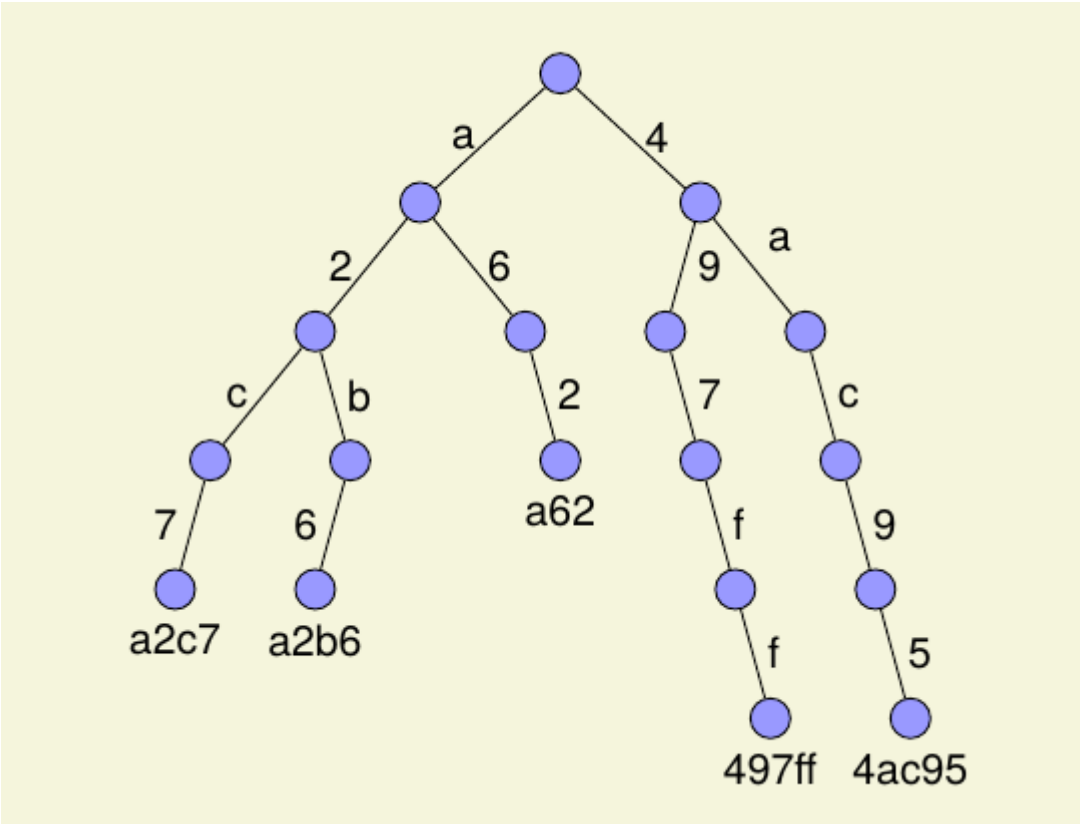
- Ethereum data structures
- Ethereum state
- Transaction and Block details
- The EVM

  Memory

  Storage
- EVM Languages

Ethereum Data Structures

Ethereum uses Merkle Patricia Tries / Radix Tries for their searching performance and low memory footprint.



More recent data structure is the Verkle tree which we will cover in a later lesson.

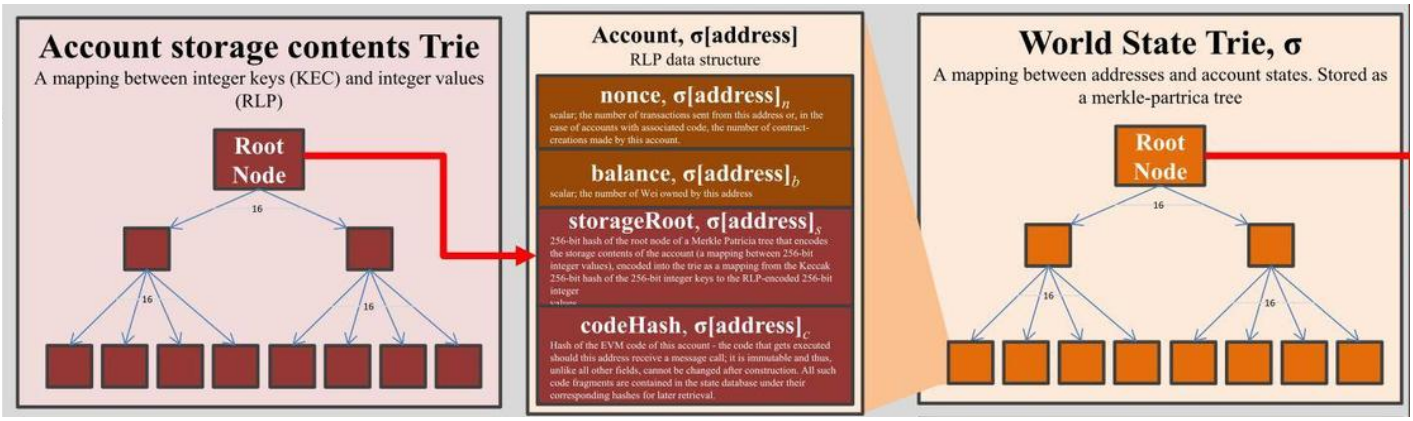
Ethereum State

There are 3 Tries

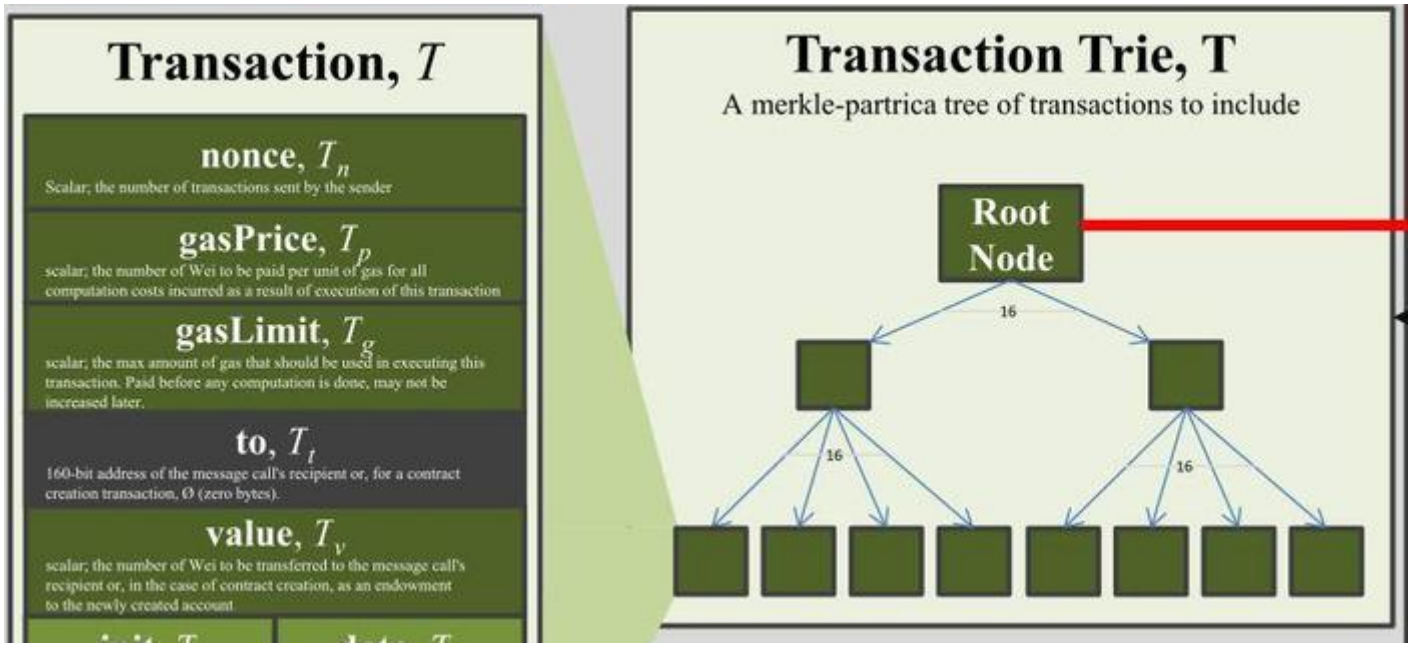
- World State
- Transaction
- Transaction Receipt

See : Ethereum State Trie Architecture Explained**

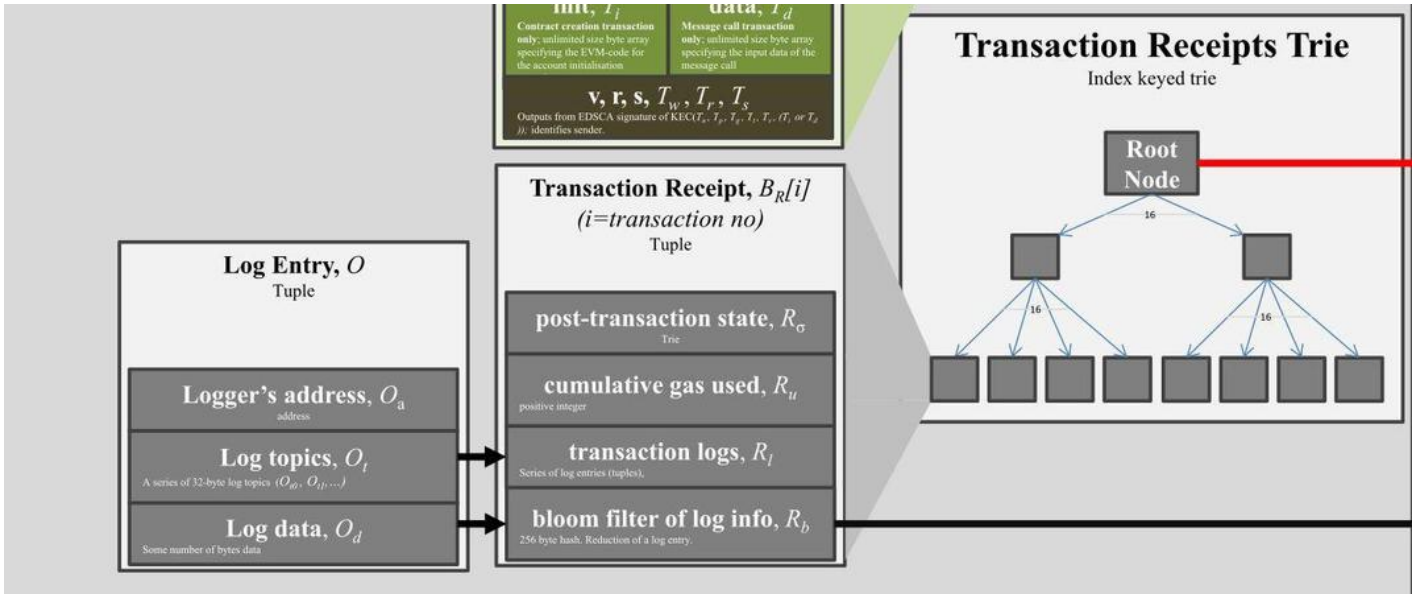
World and Account State



Transactions



Transaction Receipts and Logs



Transaction and Transaction Receipt Tries

Purpose:

- Transaction Tries: records transaction request
- Transaction Receipt Tries: records the transaction outcome

Parameters used in composing a Transaction Trie details in section 4.3 of the [yellow paper]

- nonce,
- gas price,
- gas limit,
- recipient,
- transfer value,
- transaction signature values, and
- account initialization (if transaction is of contract creation type), or transaction data (if transaction is a message call)

Parameters used in composing a Transaction Receipt Trie details in section 4.4.1 of the [yellow paper]:

- post-transaction state,
- the cumulative gas used,
- the set of logs created through execution of the transaction, and
- the Bloom filter composed from information in those logs

Ethereum Block Fields

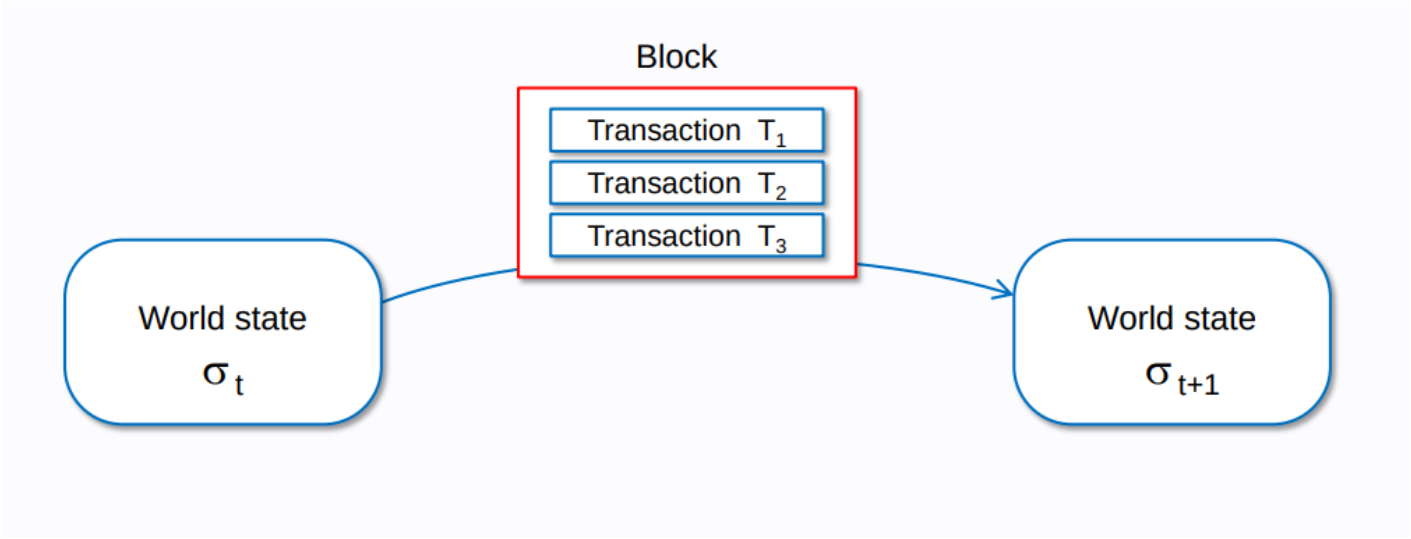
See complete description

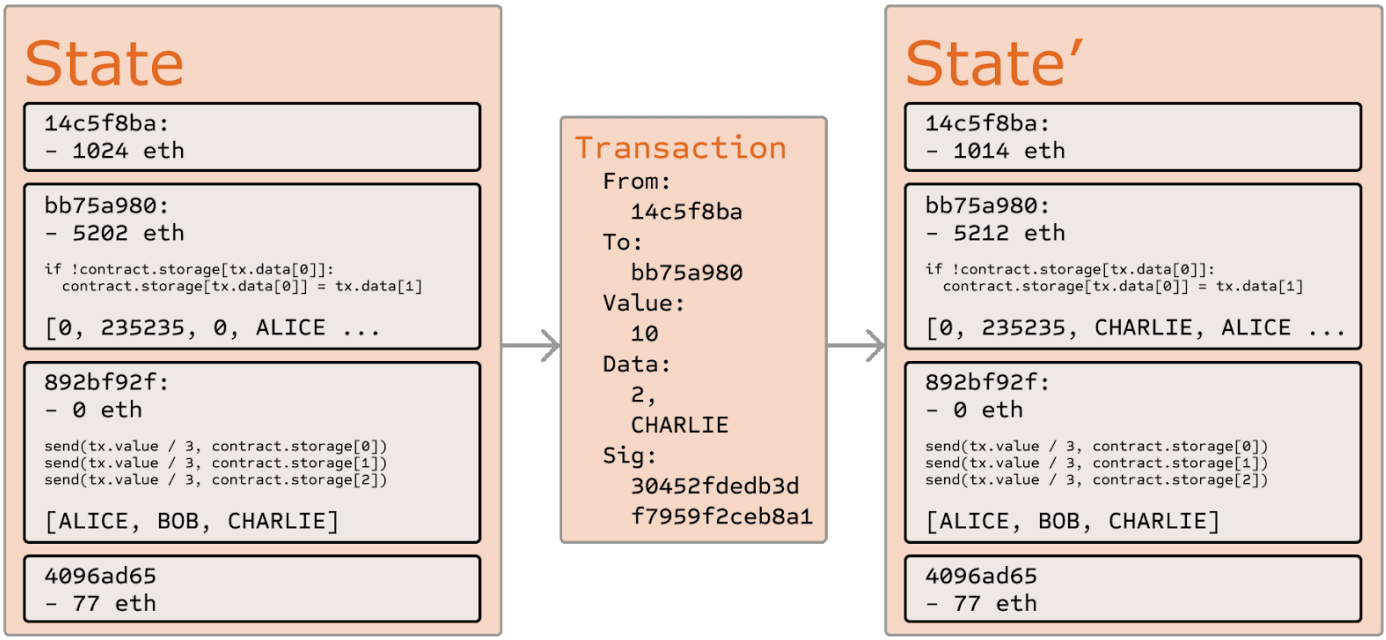
Field	Description
randao_reveal	a value used to select the next block proposer
eth1_data	information about the deposit contract
graffiti	arbitrary data used to tag blocks
proposer_slashings	list of validators to be slashed
attester_slashings	list of validators to be slashed
attestations	list of attestations in favor of the current block
deposits	list of new deposits to the deposit contract
voluntary_exits	list of validators exiting the network
sync_aggregate	subset of validators used to serve light clients
execution_payload	transactions passed from the execution client

Field	Description
parent_hash	hash of the parent block

Field	Description
fee_recipient	account address for paying transaction fees to
state_root	root hash for the global state after applying changes in this block
receipts_root	hash of the transaction receipts trie
logs_bloom	data structure containing event logs
prev_randao	value used in random validator selection
block_number	the number of the current block
gas_limit	maximum gas allowed in this block
gas_used	the actual amount of gas used in this block
timestamp	the block time
extra_data	arbitrary additional data as raw bytes
base_fee_per_gas	the base fee value
block_hash	Hash of execution block
transactions	list of transactions to be executed

Ethereum Transactions





Fields

- `recipient` – the receiving address (if an externally-owned account, the transaction will transfer value. If a contract account, the transaction will execute the contract code)
- `signature` – the identifier of the sender. This is generated when the sender's private key

signs the transaction and confirms the sender has authorised this transaction

- `nonce` - a sequentially incrementing counter which indicates the transaction number from the account
- `value` – amount of ETH to transfer from sender to recipient (in WEI, a denomination of ETH)
- `data` – optional field to include arbitrary data
- `gasLimit` – the maximum amount of gas units that can be consumed by the transaction. Units of gas represent computational steps
- `maxPriorityFeePerGas` - the maximum price of the consumed gas to be included as a tip to the validator
- `maxFeePerGas` - the maximum fee per unit of gas willing to be paid for the transaction (inclusive of `baseFeePerGas` and `maxPriorityFeePerGas`)

Example Transaction

```
{
  from:
"0xEA674fdDe714fd979de3EdF0F56AA9716B898ec
8",
  to:
"0xac03bb73b6a9e108530aff4df5077c2b3d481e5
a",
  gasLimit: "21000",
  maxFeePerGas: "300",
  maxPriorityFeePerGas: "10",
  nonce: "0",
  value: "10000000000"
}
```

Some practical points about transaction selection

- Block producers choose which transactions to include in a block
- Block producers can add their own transactions to a block
- Block producers choose the order of transactions in a block

- Your transaction is in competition with other transactions for inclusion in the block

We will talk about the consequences of these points in our MEV lesson.

Transaction Processing

Before the transaction executes it needs to pass some validity tests

- The transaction follows the rules for well-formed RLP (recursive length prefix.)
- The signature on the transaction is valid.
- The nonce on the transaction is valid, i.e. it is equivalent to the sender account's current nonce.
- The gas_limit is greater than or equal to the intrinsic_gas used by the transaction.
- The sender's account balance contains the cost required in up-front payment.

(For details of RLP see docs)

View Functions and modifying state

From documentation

If the compiler's EVM target is Byzantium or newer (default) the opcode `STATICCALL` is used when `view` functions are called, which enforces the state to stay unmodified as part of the EVM execution.
For library `view` functions `DELEGATECALL` is used, because there is no combined `DELEGATECALL` and `STATICCALL`.
This means library `view` functions do not have run-time checks that prevent state modifications.

The following statements are considered modifying the state:

1. Writing to state variables.
2. Emitting events.
3. Creating other contracts.
4. Using `selfdestruct`.
5. Sending Ether via calls.
6. Calling any function not marked `view` or `pure`.

7. Using low-level calls.
8. Using inline assembly that contains certain opcodes.

Note : Getter methods are automatically marked `view`.

The EVM

The EVM is a stack machine , the stack has a maximum size of 1024.

Stack items have a size of 256 bits; in fact, the EVM is a 256-bit word machine (this facilitates Keccak256 hash scheme and elliptic-curve computations).

During execution 2 areas are available for variables

- memory - a transient memory which does not persist between transactions
- storage - part of a Merkle Patricia storage trie associated with the contract’s account, part of the global state

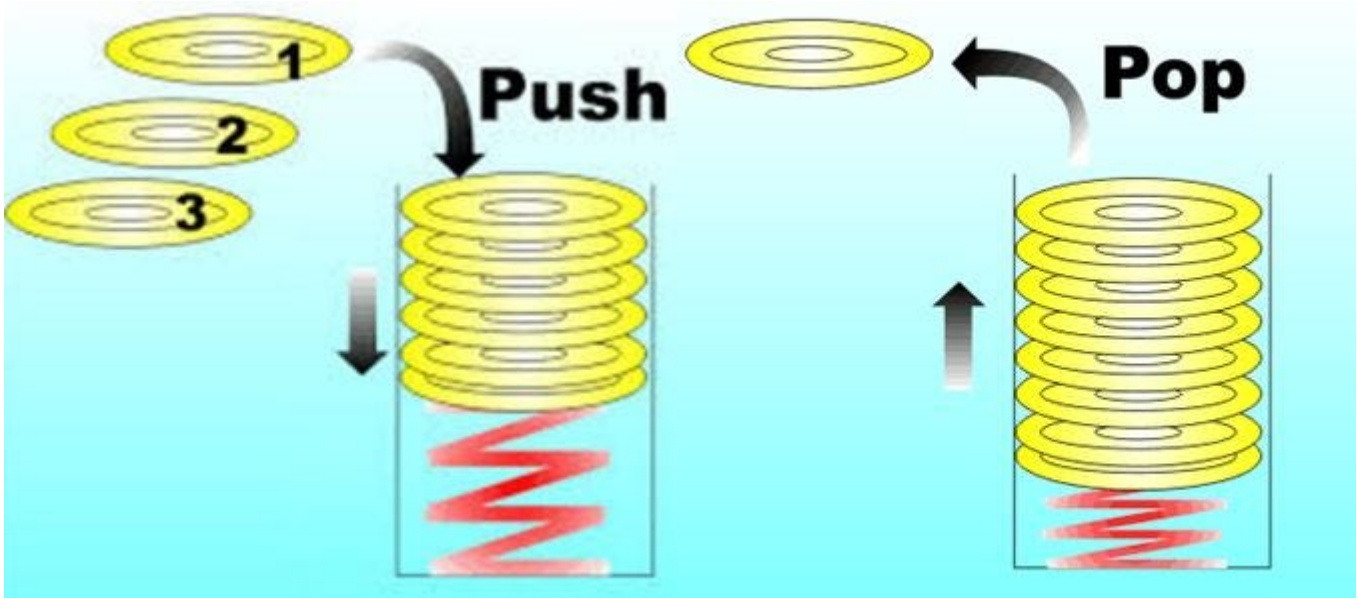
Data areas

Data can be stored in

- Stack
- Calldata
- Memory
- Storage
- Code
- Logs

EVM State transition

The Stack



The top 16 items can be manipulated or accessed at once (or stack too deep error)
See this article : All About Stack

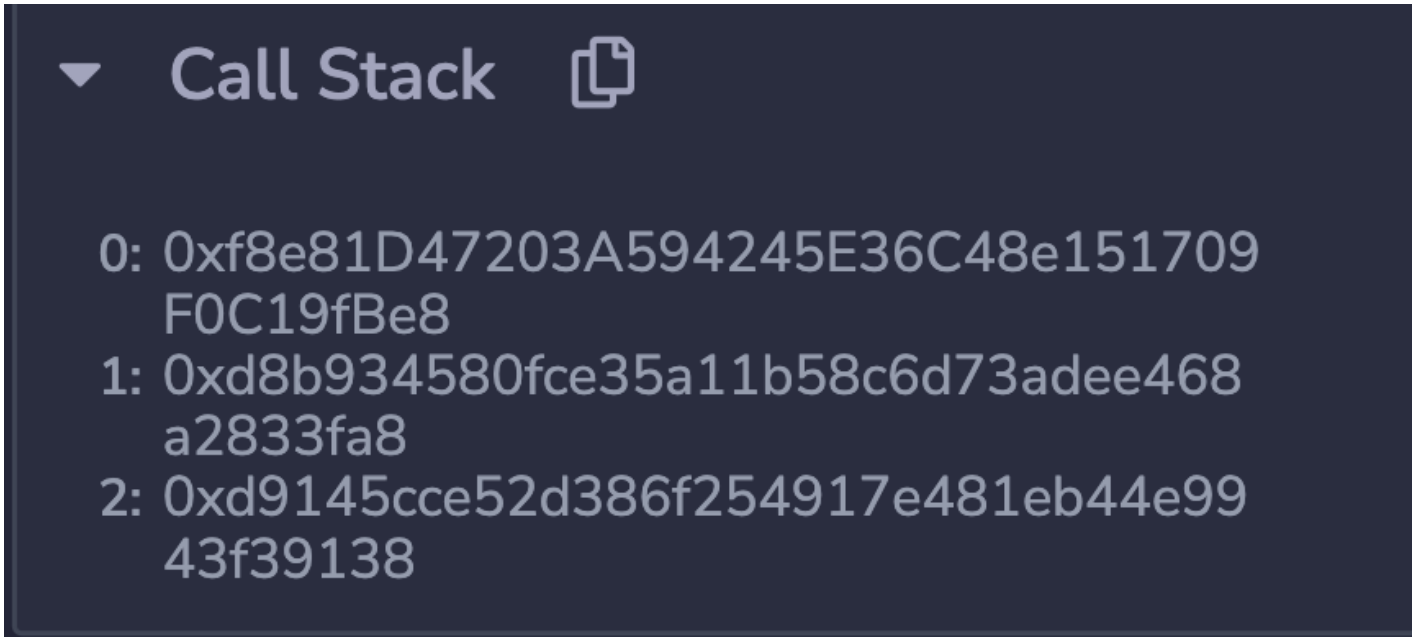
Don't confuse this with the call stack

Call Stack

The "call stack" is the current execution environment of the contract where the EVM is running.
It is an array of addresses of contracts being called, and these addresses are added or removed from the call stack as the CALL opcodes are executed.
The last address on the call stack, or the address at the top of the stack, is the contract where the EVM is currently executing opcodes.

The contract address at the bottom of the call stack is the one that was initially called by the EOA.

This is in Remix



Initial contract was
0xf8e81D47203A594245E36C48e151709F0C19fBe8
2nd contract was
0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8
3rd contract was
0xd9145CCE52D386f254917e481eB44e9943F39138

See also this question

Memory

Memory is a byte-array. Memory starts off zero-size, but can be expanded in 32-byte chunks by simply accessing or storing memory at indices greater than its current size.
Since memory is contiguous, it does save gas to keep it packed and shrink its size, instead of having large patches of zeros.

- MLOAD loads a word from memory into the stack.
- MSTORE saves a word to memory.
- MSTORE8 saves a byte to memory.

Memory Expansion

From Explanation

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".
If you use <=724 bytes of memory the second part of the equation is 0

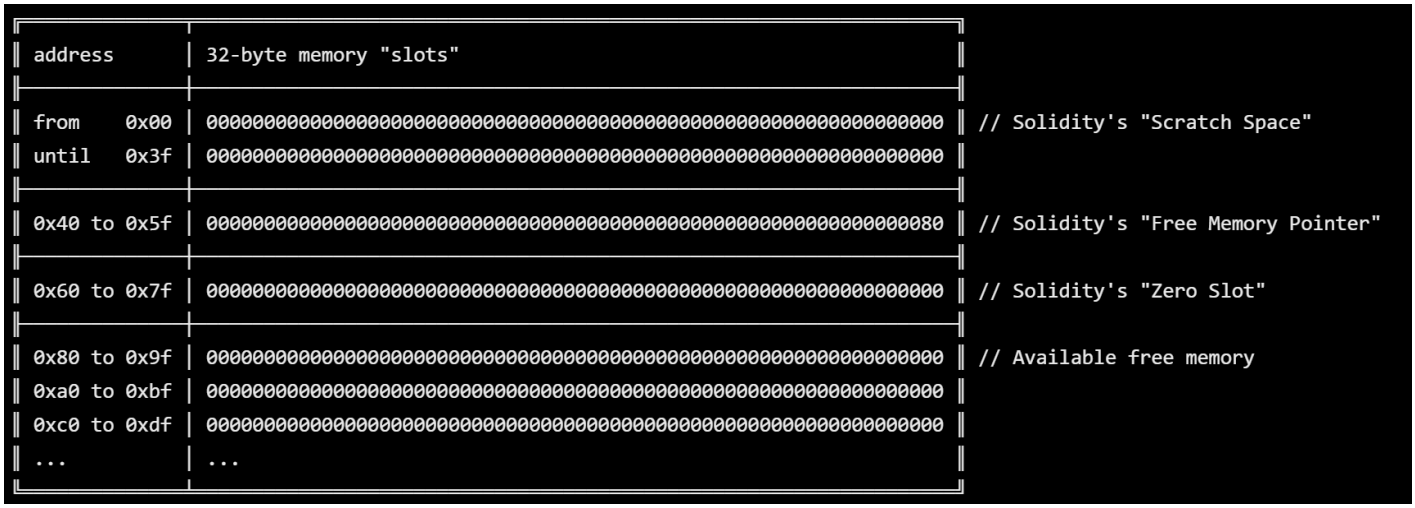
$$C_{\text{mem}}(a) \equiv G_{\text{memory}} \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor$$

(From the yellow paper)
"Note also that Cmem is the memory cost function (the expansion function being the difference between the cost before and after). It is a polynomial, with the higher-order coefficient divided

and floored, and thus linear up to 704B of memory used, after which it costs substantially more"

Memory is a byte array. This means 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.



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.

Storage

See Documentation

It is useful when thinking about the storage to think about fixed size and dynamic sized variables.

FiatTokenV2_1 <<Contract>> 0xa2327a938febf5fec13bacfb16ae10ecbc4cbdcf			
slot	type: <inherited contract>.variable (bytes)		
0	unallocated (12)	address: Ownable._owner (20)	
1	unallocated (11)	bool: Pausable.paused (1)	address: Pausable.pauser (20)
2	unallocated (12)	address: Blacklistable.blacklister (20)	
3	mapping(address=>bool): Blacklistable.blacklisted (32)		
4	string: FiatTokenV1.name (32)		
5	string: FiatTokenV1.symbol (32)		
6	unallocated (31)	uint8: FiatTokenV1.decimals (1)	
7	string: FiatTokenV1.currency (32)		
8	unallocated (11)	bool: FiatTokenV1.initialized (1)	address: FiatTokenV1.masterMinter (20)
9	mapping(address=>uint256): FiatTokenV1.balances (32)		
10	mapping(address=>mapping(address=>uint256)): FiatTokenV1.allowed (32)		
11	uint256: FiatTokenV1.totalSupply_ (32)		
12	mapping(address=>bool): FiatTokenV1.minters (32)		
13	mapping(address=>uint256): FiatTokenV1.minterAllowed (32)		
14	unallocated (12)	address: Rescuable._rescuer (20)	
15	bytes32: EIP712Domain.DOMAIN_SEPARATOR (32)		
16	mapping(address=>mapping(bytes32=>bool)): EIP3009._authorizationStates (32)		
17	mapping(address=>uint256): EIP2612._permitNonces (32)		
18	unallocated (31)	uint8: FiatTokenV2._initializedVersion (1)	

For fixed size variables, data is stored contiguously item after item starting with the first state variable,

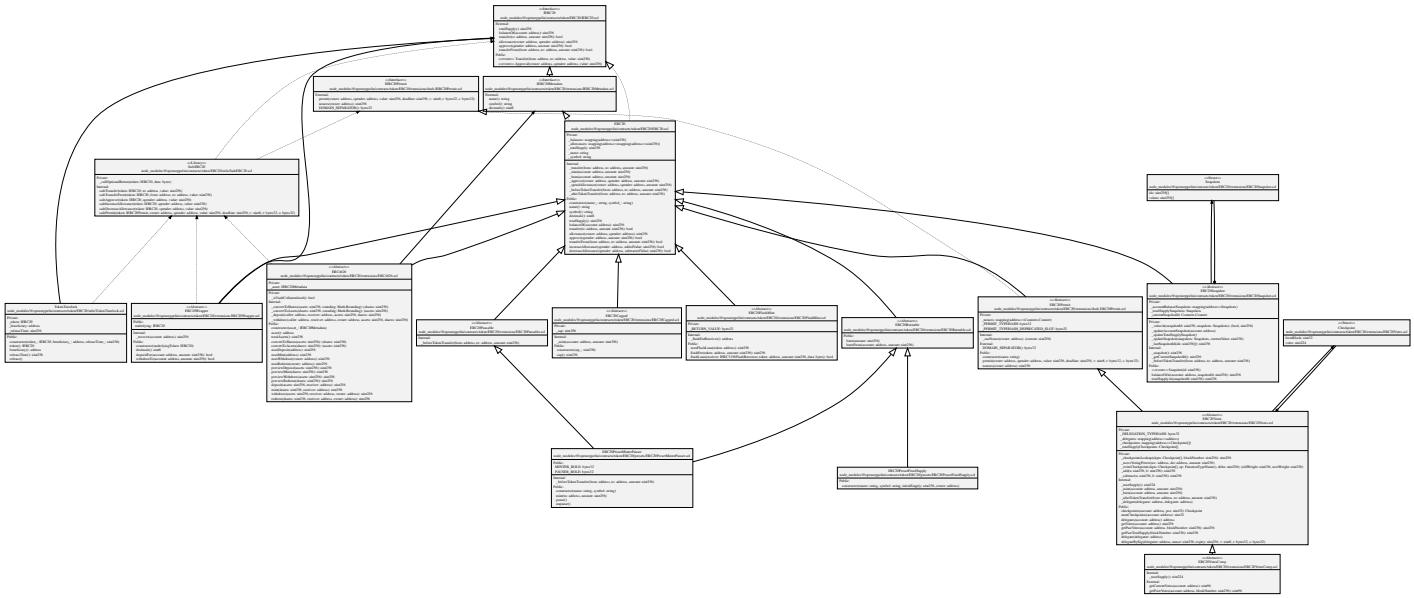
which is stored in slot 0. For each variable, a size in bytes is determined according to its type.

For variable length items such as arrays and mappings, the storage slot contains a pointer to another area of storage where the variable starts. For the details see the documentation

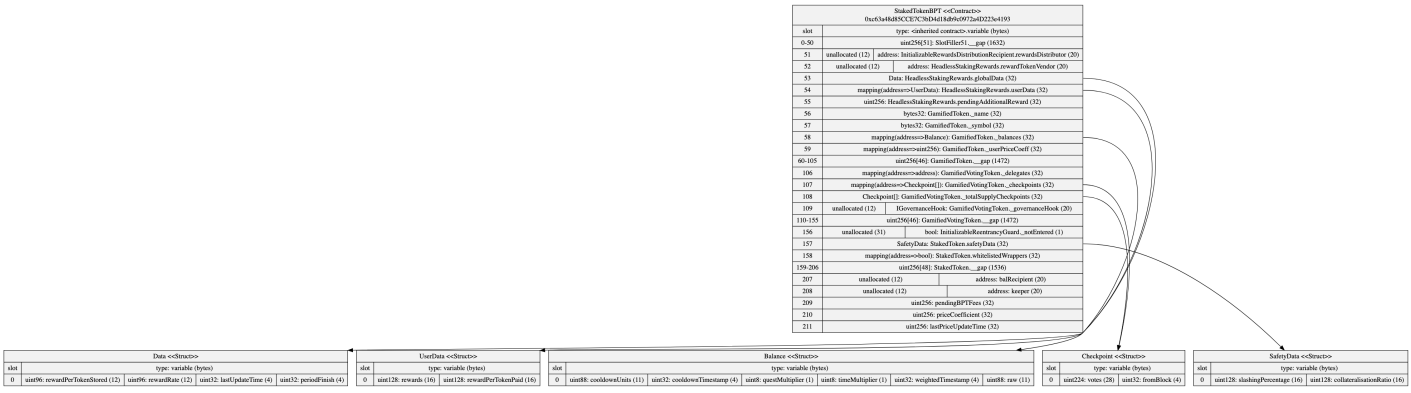
Visualisation Tool

See repo

It provides visualisation of storage, plus UML diagrams for the contract.



FiatTokenV2_1 <<Contract>> 0xa2327a938febf5fec13bacfb16ae10ecbc4cbdcf			
slot	type: <inherited contract>.variable (bytes)		
0	unallocated (12)	address: Ownable._owner (20)	
1	unallocated (11)	bool: Pausable.paused (1)	address: Pausable.pauser (20)
2	unallocated (12)	address: Blacklistable.blacklister (20)	
3	mapping(address=>bool): Blacklistable.blacklisted (32)		
4	string: FiatTokenV1.name (32)		
5	string: FiatTokenV1.symbol (32)		
6	unallocated (31)	uint8: FiatTokenV1.decimals (1)	
7	string: FiatTokenV1.currency (32)		
8	unallocated (11)	bool: FiatTokenV1.initialized (1)	address: FiatTokenV1.masterMinter (20)
9	mapping(address=>uint256): FiatTokenV1.balances (32)		
10	mapping(address=>mapping(address=>uint256)): FiatTokenV1.allowed (32)		
11	uint256: FiatTokenV1.totalSupply_ (32)		
12	mapping(address=>bool): FiatTokenV1.minters (32)		
13	mapping(address=>uint256): FiatTokenV1.minterAllowed (32)		
14	unallocated (12)	address: Rescuable._rescuer (20)	
15	bytes32: EIP712Domain.DOMAIN_SEPARATOR (32)		
16	mapping(address=>mapping(bytes32=>bool)): EIP3009._authorizationStates (32)		
17	mapping(address=>uint256): EIP2612._permitNonces (32)		
18	unallocated (31)	uint8: FiatTokenV2._initializedVersion (1)	



Code Execution

OpCodes

- Stack-manipulating opcodes (POP, PUSH, DUP, SWAP)
- Arithmetic/comparison/bitwise opcodes (ADD, SUB, GT, LT, AND, OR)
- Environmental opcodes (CALLER, CALLVALUE, NUMBER)
- Memory-manipulating opcodes (MLOAD, MSTORE, MSTORE8, MSIZE)
- Storage-manipulating opcodes (SLOAD, SSTORE)
- Program counter related opcodes (JUMP, JUMPI, PC, JUMPDEST)
- Halting opcodes (STOP, RETURN, REVERT, INVALID, SELFDESTRUCT)

https://www.ethervm.io/

Machine State

The machine state is a tuple consisting of five elements:

1. gas_available
2. program_counter
3. memory_contents A series of zeroes of size 2^{256}
4. memory_words.count
5. stack_contents

There is also the current operation to be executed

Gas Refunds

Since EIP-3529 gas refunds are not given for self destructing contracts, and the amount of refund for storage has been reduced.

References

DEVCON1: Understanding the Ethereum Blockchain Protocol - Vitalik Buterin

Mastering Ethereum by Andreas Antonopoulos

White paper

Beige Paper

Yellow Paper

Noxx Articles about the EVM

EVM Languages

- Solidity
  - The most popular programming language for Ethereum contracts
- LLL
  - Low-level Lisp-like Language
- Vyper
  - A language with overflow-checking, numeric units but without unlimited loops
- Yul / Yul+
  - An intermediate language that can be compiled to bytecode for different backends. Support for EVM 1.0, EVM 1.5 and Ewasm is planned, and it is designed to be a usable common denominator of all three platforms.
- FE
  - Statically typed language Inspired by Rust and Python
- Huff see article
  - Low level language
- Pyramid Scheme (experimental)

    - A Scheme compiler into EVM that follows the SICP compilation approach
- Flint
    - A language with several security features: e.g. asset types with a restricted set of atomic operations
- LLLL
    - An LLL-like compiler being implemented in Isabelle/HOL
- HAseembly-evm
    - An EVM assembly implemented as a Haskell DSL
- Bamboo (experimental)
    - A language without loops

Vyper

- Pythonic programming language
- Strong typing
- Small and understandable compiler code
- Deliberately has less features than Solidity with the aim of making contracts more secure and easier to audit. Vyper does not support

  - Modifiers
  - Inheritance
  - Inline assembly
  - Function overloading
  - Operator overloading
  - Recursive calling
  - Infinite-length loops

FE

See :Repo

- Statically typed language for the Ethereum Virtual Machine (EVM).
- Inspired by Python and Rust.
- Aims to be easy to learn -- even for developers who are new to the Ethereum ecosystem.

- Fe development is still in its early stages, the language had its alpha release in January 2021.

Features

- Bounds and overflow checking
- Decidability by limitation of dynamic program behavior
- More precise gas estimation (as a consequence of decidability)
- Static typing
- Pure function support
- Restrictions on reentrancy
- Static looping
- Module imports
- Standard library
- Usage of YUL IR to target both EVM and eWASM
- WASM compiler binaries for enhanced portability and in-browser compilation of Fe contracts
- Implementation in a powerful, systems-oriented language (Rust) with strong safety guarantees

to reduce risk of compiler bugs

References

EVM languages