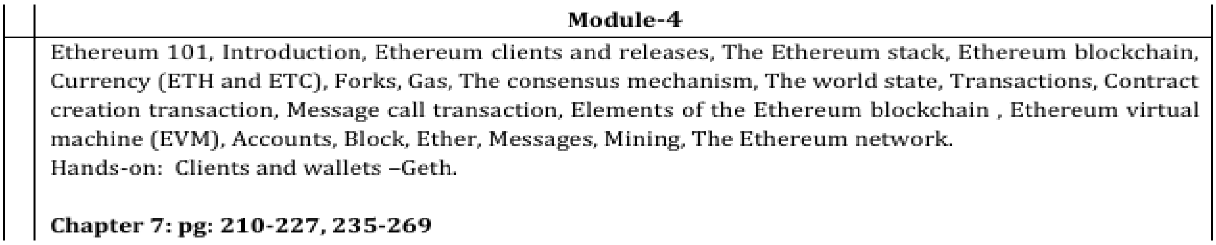


Module-4

Ethereum 101, Introduction, Ethereum clients and releases, The Ethereum stack, Ethereum blockchain, Currency (ETH and ETC), Forks, Gas, The consensus mechanism, The world state, Transactions, Contract creation transaction, Message call transaction, Elements of the Ethereum blockchain , Ethereum virtual machine (EVM), Accounts, Block, Ether, Messages, Mining, The Ethereum network.
Hands-on: Clients and wallets –Geth.

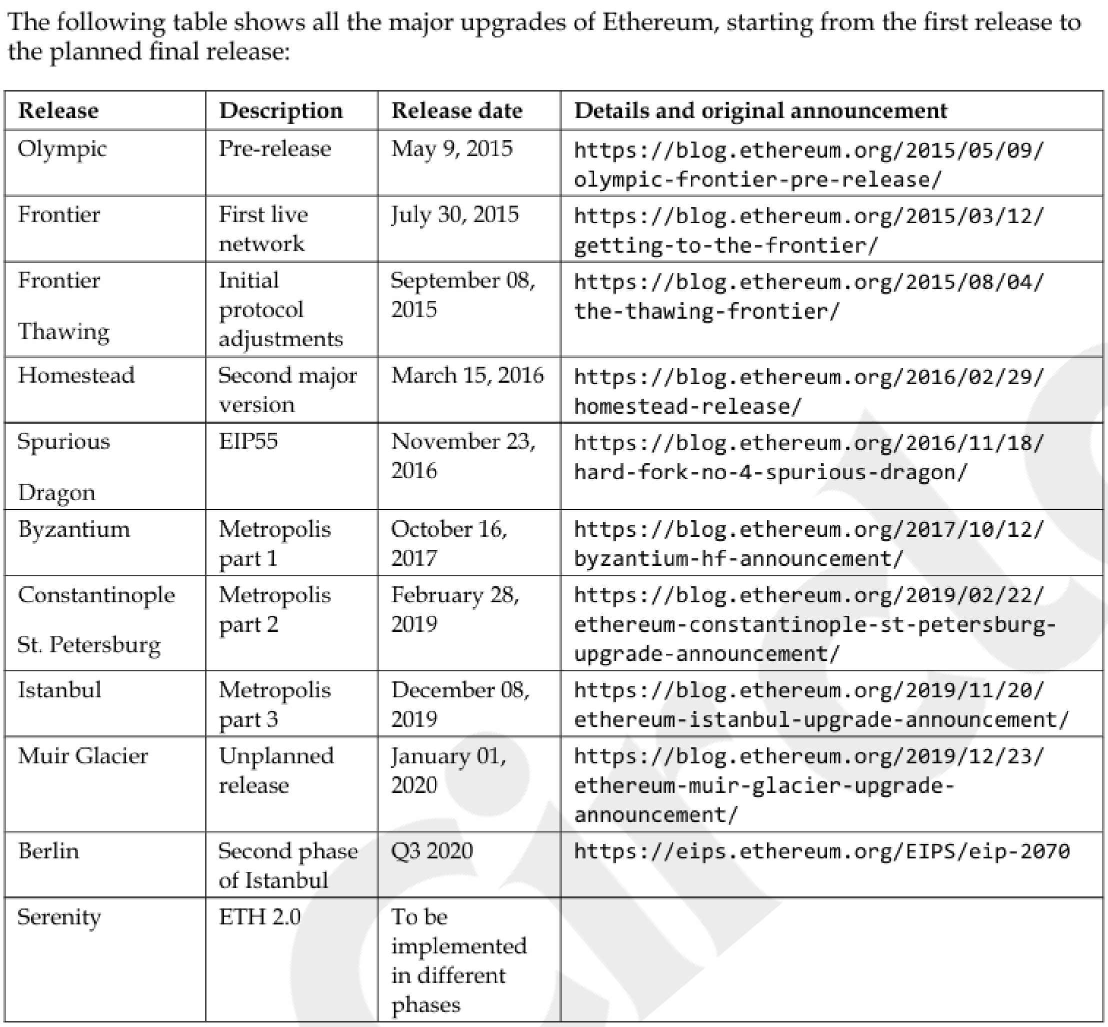
Chapter 7: pg: 210-227, 235-269

Introduction:

- Vitalik Buterin (https://vitalik.ca) conceptualized Ethereum (https://ethereum.org) in November, 2013.
- The key idea proposed was the development of a Turing-complete language that allows the development of arbitrary programs (smart contracts) for blockchain and decentralized applications.
- In other words, **Ethereum** allows developers to create and run applications on its blockchain using smart contracts.

Ethereum Clients and Releases:

- Various Ethereum clients have been developed using different languages and currently most popular are **go-Ethereum** and **parity.**
- go-Ethereum was developed using Golang, whereas parity was built using **Rust**.
- There are other clients available too, but usually, the go-Ethereum client known as geth is sufficient for all purposes. Mist is a user-friendly Graphical User Interface (GUI) wallet that runs geth in the background to sync with the network.
- The first version of Ethereum, called **Olympic**, was released in **May, 2015.** Two months later, a version of Ethereum called Frontier was released in July.
- Another version named **Homestead** with various improvements was released in **March, 2016.**
- The latest Ethereum release is called **Muir Glacier**, which delays the difficulty bomb (https://eips.ethereum.org/EIPS/eip-2384). A major release before that was Istanbul, which included changes around privacy and scaling capabilities.



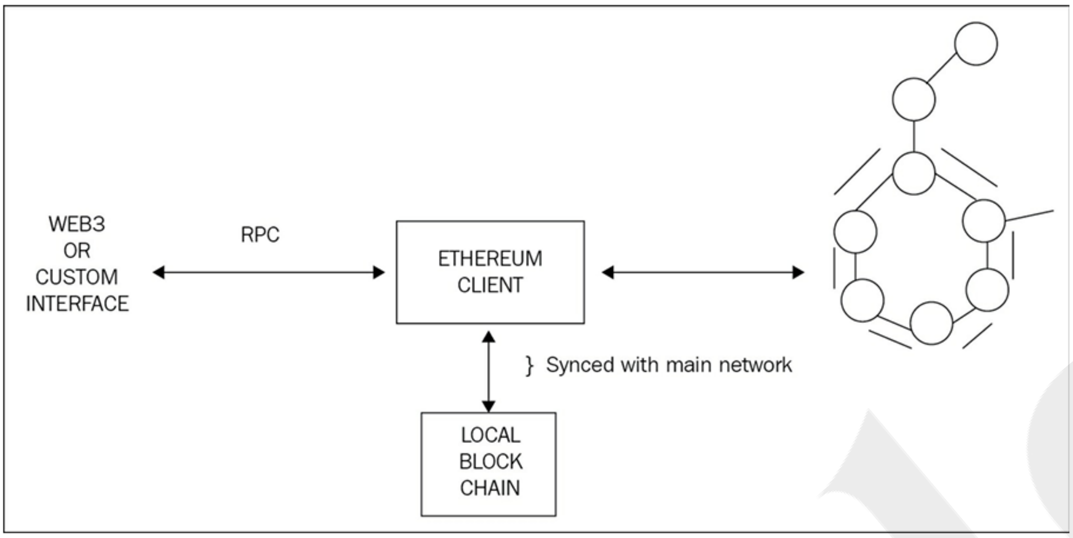
The following table shows all the major upgrades of Ethereum, starting from the first release to the planned final release:

Release	Description	Release date	Details and original announcement
Olympic	Pre-release	May 9, 2015	https://blog.ethereum.org/2015/05/09/olympic-frontier-pre-release/
Frontier	First live network	July 30, 2015	https://blog.ethereum.org/2015/03/12/getting-to-the-frontier/
Frontier Thawing	Initial protocol adjustments	September 08, 2015	https://blog.ethereum.org/2015/08/04/the-thawing-frontier/
Homestead	Second major version	March 15, 2016	https://blog.ethereum.org/2016/02/29/homestead-release/
Spurious Dragon	EIP55	November 23, 2016	https://blog.ethereum.org/2016/11/18/hard-fork-no-4-spurious-dragon/
Byzantium	Metropolis part 1	October 16, 2017	https://blog.ethereum.org/2017/10/12/byzantium-hf-announcement/
Constantinople St. Petersburg	Metropolis part 2	February 28, 2019	https://blog.ethereum.org/2019/02/22/ethereum-constantinople-st-petersburg-upgrade-announcement/
Istanbul	Metropolis part 3	December 08, 2019	https://blog.ethereum.org/2019/11/20/ethereum-istanbul-upgrade-announcement/
Muir Glacier	Unplanned release	January 01, 2020	https://blog.ethereum.org/2019/12/23/ethereum-muir-glacier-upgrade-announcement/
Berlin	Second phase of Istanbul	Q3 2020	https://eips.ethereum.org/EIPS/eip-2070
Serenity	ETH 2.0	To be implemented in different phases	

The final planned release of Ethereum is called Serenity and is envisaged to introduce the final version of the PoS based-blockchain, replacing PoW.

The Ethereum Stack

- The Ethereum stack consists of various components. At the core, there is the Ethereum blockchain running on the P2P Ethereum network.
- Secondly, there's an Ethereum client (usually geth) that runs on the nodes and connects to the peer-to-peer Ethereum network from where blockchain is downloaded and stored locally.
- It provides various functions, such as mining and account management. The local copy of the blockchain is synchronized regularly with the network.
- Another component is the web3.js library that allows interaction with geth via the **Remote Procedure Call (RPC)** interface. This can be visualized in the following diagram

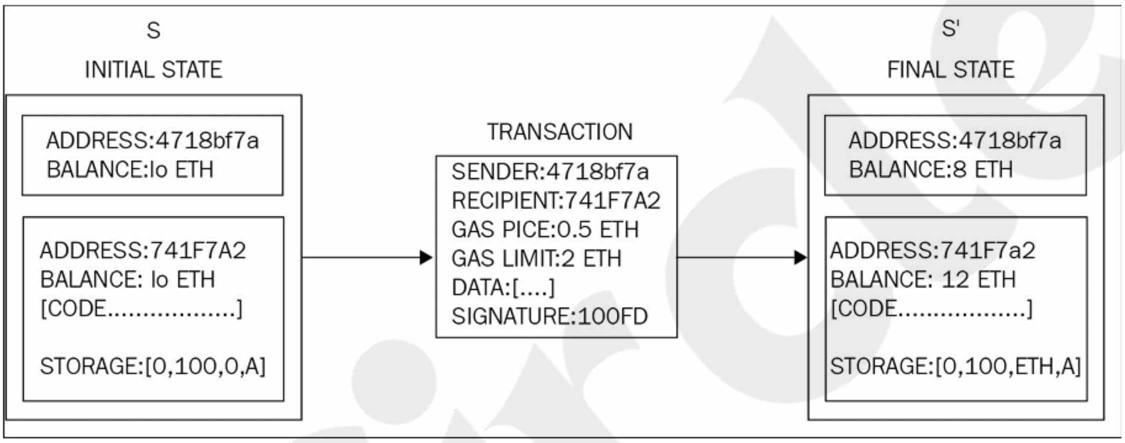


The Ethereum stack is essentially a suite of tools and technologies designed to develop and interact with Ethereum-based applications. It's the foundation for building decentralized applications (dApps) on the Ethereum blockchain. Here's a breakdown:

1. **Ethereum Virtual Machine (EVM)**: This is the computation engine that executes smart contracts on the blockchain. Every Ethereum node runs the EVM.
2. **Smart Contracts**: These are self-executing contracts with predefined rules written in code. They are usually written in programming languages like Solidity and then deployed to the blockchain.
3. **Solidity**: The most popular programming language for writing Ethereum smart contracts. It's high-level and specifically tailored for the Ethereum Virtual Machine.
4. **Web3.js**: A JavaScript library that enables developers to interact with the Ethereum blockchain. Through this, developers can send transactions, fetch blockchain data, and interact with smart contracts.
5. **Ganache**: A personal blockchain for Ethereum development. It helps developers simulate the blockchain environment locally, making it easier to test and deploy contracts.
6. **Truffle**: A development framework for Ethereum that simplifies tasks like compiling smart contracts, deploying them, and testing them. It's widely used in dApp development.
7. **Infura**: A cloud-based service that provides scalable access to Ethereum nodes. Developers use it to connect their dApps to the blockchain without needing to run their own node.
8. **MetaMask**: A browser extension that acts as a wallet and interface for Ethereum applications. It allows users to interact with dApps by managing keys and signing transactions.

Ethereum Blockchain:

- Ethereum, just like any other blockchain, can be visualized as a transaction-based state machine. This is mentioned in the Ethereum yellow paper written by Dr. Gavin Wood.
- The idea is that a genesis state is transformed into a final state by executing transactions incrementally. The final transformation is then accepted as the absolute undisputed version of the state.
- In the following diagram, the Ethereum state transition function is shown, where a transaction execution has resulted in a state transition.



Ethereum State transition function

- In the preceding example, a transfer of 2 Ether from Address 4718bf7a to Address 741f7a2 is initiated. The initial state represents the state before the transaction execution and the final state is what the morphed state looks like

Currency (ETH and ETC)

ETH (Ethereum)

- **ETH** is the native cryptocurrency of the **Ethereum blockchain**.
- It is used to **pay transaction (gas) fees**, deploy smart contracts, stake for rewards, and power Dapps.
- Ethereum is the **second-largest blockchain** after Bitcoin.
- **It is Highly adopted** in DeFi, NFTs, DAOs, and gaming.
- It runs on **Proof of Stake (PoS)** since *The Merge* in 2022.
- Symbol: ETH

CTC (Creditcoin)

- **CTC** is the native currency of the **Creditcoin blockchain**.
- Focused on building a **decentralized credit history system**.
- It is used to record loan transactions between lenders and borrowers on-chain.
- It aims to help **underbanked regions** by enabling access to decentralized finance.
- It runs on its own blockchain, separate from Ethereum.
- Symbol: CTC

Feature	**ETH**	**CTC**
Blockchain	Ethereum	Creditcoin
Main Use	Gas fees, dApps, DeFi, staking	Recording loans, building credit score
Ecosystem Size	Very large	Niche, growing
Consensus	Proof of Stake	Proof of Work (may shift to PoS)
Popularity	Top 2 crypto	Mid-tier token

Forks

A **fork** occurs when the blockchain splits into two separate paths. This happens when:

- There's a disagreement about rules (consensus protocol)
- A software upgrade changes block validation rules
- Temporary desync happens among miners or validators

There are **two main types**:

1. Soft Fork

- **Definition:** A backward-compatible upgrade. Old nodes still accept new blocks, but may not understand all new features.
- **Effect:** One chain continues; old clients follow along until they upgrade.

Example: Ethereum's *EIP-150* upgrade introduced gas cost changes—older clients still validated blocks, just without full functionality.

2. Hard Fork

- **Definition:** A non-backward-compatible upgrade. All nodes must upgrade, or they'll run on a separate chain.
- **Effect:** Can lead to two different chains if there's no agreement.

Example: The DAO Hack (2016)

- $60M in ETH was stolen due to a smart contract bug.
- Ethereum developers proposed a **hard fork** to reverse the hack.
- Some community members opposed this (wanted "code is law").
- **Result:** Two chains:
  - **Ethereum (ETH):** Reverted the hack.
  - **Ethereum Classic (ETC):** Continued the original chain

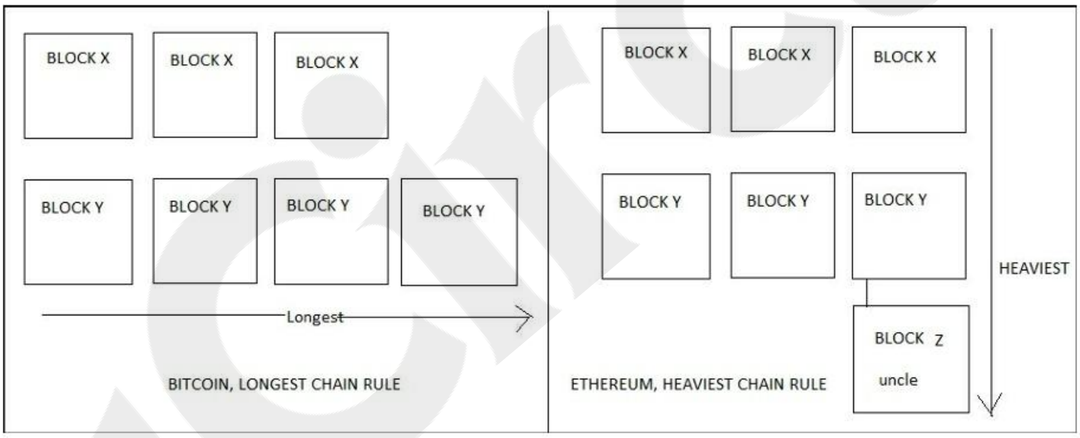
- With the latest release of homestead, due to major protocol upgrades, it resulted in a hard fork. The protocol was upgraded at block number 1,150,000, resulting in the migration from the first version of Ethereum known as Frontier to the second version of Ethereum called homestead. A recent unintentional fork that occurred on November 24, 2016, at 14:12:07 UTC was due to a bug in the geth client's journaling mechanism. Network fork occurred at block number 2,686,351. This bug resulted in geth failing to revert empty account deletions in the case of the empty out-of-gas exception. This was not an issue in parity (another popular Ethereum client). This means that from block number 2686351, the Ethereum blockchain is split into two, one running with parity clients and the other with geth. This issue was resolved with the release of geth version 1.5.3

Gas

- Another key concept in Ethereum is that of gas.
- All transactions on the Ethereum blockchain are required to cover the cost of computation they are performing. The cost is covered by something called **gas or crypto fuel**, which is a new concept introduced by Ethereum.
- This gas as execution fee is paid upfront by the transaction originators. The fuel is consumed with each operation.
- Each operation has a predefined amount of gas associated with it. Each transaction specifies the amount of gas it is willing to consume for its execution.
- If it runs out of gas before the execution is completed, any operation performed by the transaction up to that point is rolled back.
- If the transaction is successfully executed, then any remaining gas is refunded to the transaction originator. This concept should not be confused with mining fee, which is a different concept that is used to pay gas as a fee to the miners.

The consensus mechanism

- The consensus mechanism in Ethereum is based on the GHOST protocol originally proposed by Zohar and Sompolinsky in December 2013.
- Ethereum uses a simpler version of this protocol, where the chain that has most computational effort spent on it in order to build it is identified as the definite version.
- Another way of looking at it is to find the longest chain, as the longest chain must have been built by consuming adequate mining effort. **Greedy Heaviest Observed Subtree (GHOST)** was first introduced as a mechanism to alleviate the issues arising out of fast block generation times that led to stale or orphan blocks.
- In **GHOST,** stale blocks are added in calculations to figure out the longest and heaviest chain of blocks. Stale blocks are called **Uncles or Ommers** in Ethereum. The following diagram shows a quick comparison between the longest and heaviest chain:

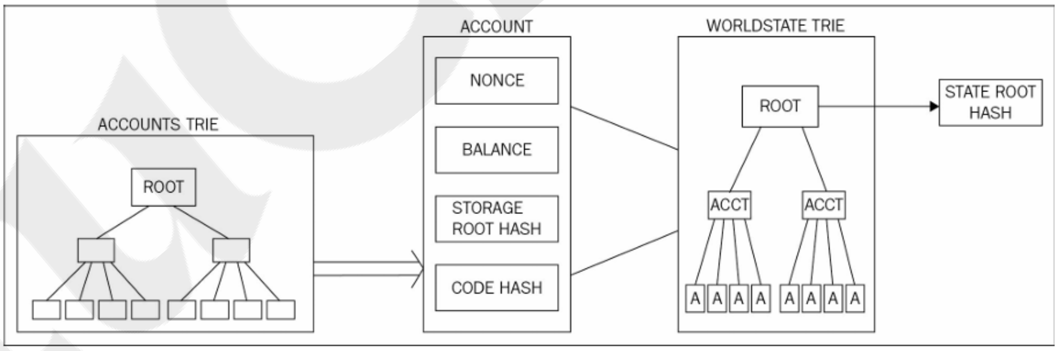


Longest versus heaviest chain

The world state:

- The world state in Ethereum represents the global state of the Ethereum blockchain.
- It is basically a mapping between Ethereum addresses and account states. The addresses are 20 bytes long.
- This mapping is a data structure that is serialized using Recursive Length Prefix (RLP). RLP is a specially developed encoding scheme that is used in Ethereum to serialize binary data for storage or transmission over the network and also to save the state in a Patricia tree.
- The RLP function takes an item as an input, which can be a string or a list of items, and produces raw bytes that are suitable for storage and transmission over the network.

- RLP does not encode data; instead, its main purpose is to encode structures. The account state The account state consists of four fields: nonce, balance, storageroot and codehash and is described in detail here.
  1. Nonce-This is a value that is incremented every time a transaction is sent from the address. In case of contract accounts, it represents the number of contracts created by the account. Contract accounts are one of the two types of accounts that exist in Ethereum;
  2.Balance -This value represents the number of Weis which is the smallest unit of the currency (Ether) in Ethereum held by the address.
  3. Storageroot This field represents the root node of a Merkle Patricia tree that encodes the storage contents of the account.
  4.Codehash This is an immutable field that contains the hash of the smart contract code that is associated with the account. In the case of normal accounts, this field contains the Keccak 256-bit hash of an empty string. This code is invoked via a message call.
- The world state and its relationship with accounts trie, accounts, and block header can be visualized in the following diagram. It shows the account data structure in the middle of the diagram, which contains a storage root hash derived from the root node of the account storage trie shown on the left.
- The account data structure is then used in the world state trie, which is a mapping between addresses and account states.
- Finally, the root node of the world state trie is hashed using the Keccak 256-bit algorithm and made part of the block header data structure, which is shown on the right-hand side of the diagram as state root hash.



- Accounts trie (storage contents of account), account tuple, world state trie, and state root hash and their relationship Accounts trie is basically a Merkle Patricia tree used to encode the storage contents of an account.
- The contents are stored as a mapping between keccak 256-bit hashes of 256-bit integer keys to the RLP-encoded 256-bit integer values.

Transactions

- A transaction in Ethereum is a digitally signed data packet using a private key that contains the instructions that, when completed, either result in a message call or contract creation.

- Transactions can be divided into two types based on the output they produce:

  a. Message call transactions: This transaction simply produces a message call that is used to pass messages from one account to another.

  b. Contract creation transactions: As the name suggests, these transactions result in the creation of a new contract. This means that when this transaction is executed successfully, it creates an account with the associated code. Both of these transactions are composed of a number of common fields, which are described here.
- **Nonce**- Nonce is a number that is incremented by one every time a transaction is sent by the sender. It must be equal to the number of transactions sent and is used as a unique identifier for the transaction. A nonce value can only be used once.
- **gasPrice**- The gasPrice field represents the amount of Wei required in order to execute the transaction.
- **gasLimit**- The gasLimit field contains the value that represents the maximum amount of gas that can be consumed in order to execute the transaction.
- **Value-** Value represents the total number of Wei to be transferred to the recipient; in the case of a contract account, this represents the balance that the contract will hold.
- **Signature-** Signature is composed of three fields, namely v, r, and s. These values represent the digital signature (R, S) and some information that can be used to recover the public key (V).
- Also of the transaction from which the sender of the transaction can also be determined. The signature is based on **ECDSA scheme** and makes use of the SECP256k1 curve.

  Ethereum Signatures: V, R, S Explained
- When you **sign a transaction** in Ethereum using your private key, the signature produced consists of **three components**:
- **Signature Format:**

  ECDSASIGN(message, private_key) = (V, R, S)

  1. R (32 bytes)
- **What it is:**
- Derived from a point on the elliptic curve.
- A random number k is chosen → point (x, y) = k * G (where G is the generator).
- The x coordinate of this point becomes **R**.

  Constraints:
- Must be greater than 0 and less than n (secp256k1 curve order).

2. S (32 bytes)

What it is:

- Calculated using the formula:
- ini
-
- S = (k^{-1} * (hash + private_key * R)) mod n
- Represents the proof that the signer knows the private key corresponding to the public key.

Constraint:

- Also must be within the range of the curve.

V (1 byte)

What it is:

- Also called the **recovery ID**.
- Helps recover the public key from the signature (using ECDSARECOVER function).

Ethereum-specific detail:

- ECDSASIGN (Message, Private Key) = (V, R, S) Init The Init field is used only in transactions that are
- In **secp256k1**, recovery ID is either 0 or 1.
- **Ethereum adds 27**, so V is usually **27 or 28**.
- Why add 27? It helps distinguish Ethereum signatures from other formats (a legacy choice).

What is ECDSA RECOVER?

- A **precompiled contract** in Ethereum (0x01) that:
- Takes (V, R, S) and the message hash as input.
- Recovers the **public key** that created the signature.
- From this public key, Ethereum derives the address.
- This is how Ethereum **verifies who signed** a transaction — without storing public keys explicitly.

Real-World Use in Ethereum

- When a transaction is signed and broadcasted:
- The transaction data is hashed.
- The private key signs the hash using ECDSA.
- The resulting V, R, and S are appended to the transaction.
- Ethereum nodes use ECDSARECOVER to:

- Reconstruct the public key from the signature.
- Hash the public key to get the Ethereum address.
- Check if this address matches the transaction’s sender.

TL;DR

• **Part**	• **Size**	• **Purpose**
• V	• 1 byte	• Recovery ID (helps reconstruct public key)
• R	• 32 bytes	• X coordinate of a point on the curve (part of signature)
• S	• 32 bytes	• Calculated to prove private key knowledge (part of signature)

- These 3 values together prove a transaction was signed by the holder of a specific Ethereum address's private key — without ever revealing that key.

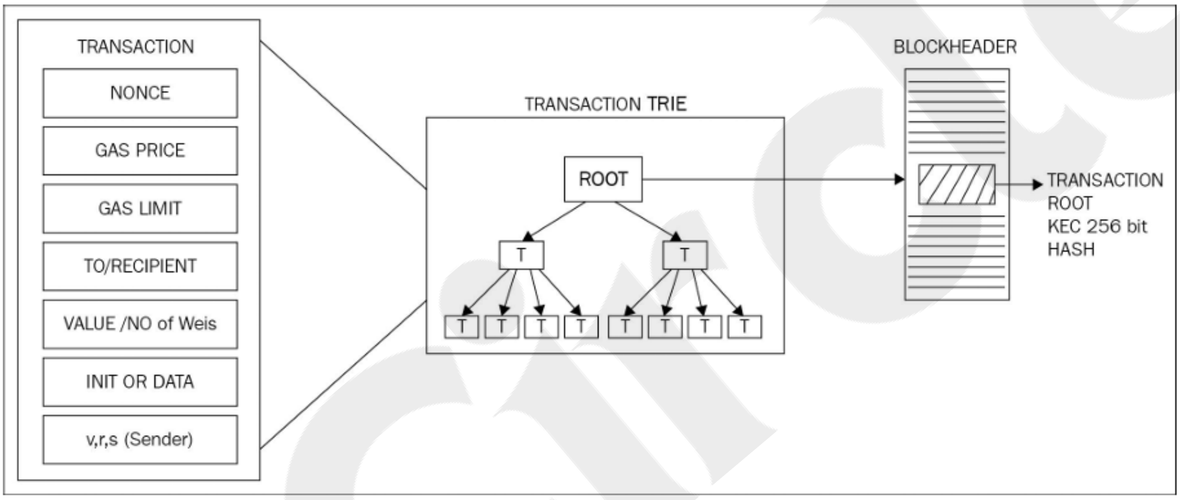
Init

- The Init field is used only in transactions that are intended to create contracts. This represents a byte array of unlimited length that specifies the EVM code to be used in the account initialization process.
- The code contained in this field is executed only once, when the account is created for the first time, and gets destroyed immediately after that.
- Init also returns another code section called body, which persists and runs in response to message calls that the contract account may receive. These message calls may be sent via a transaction or an internal code execution

Data

- If the transaction is a message call, then the data field is used instead of init, which represents the input data of the message call. It is also unlimited in size and is organized as a byte array.
- This can be visualized in the following diagram, where a transaction is a tuple of the fields mentioned earlier, which is then included in a transaction trie (a modified Merkle-Patricia tree) composed of the transactions to be included.
- Finally, the root node of transaction trie is hashed using a Keccak 256-bit algorithm and is included in the block header along with a list of transactions in the block. Transactions can be found in either transaction pools or blocks.

- When a mining node starts its operation of verifying blocks, it starts with the highest paying transactions in the transaction pool and executes them one by one.
- When the gas limit is reached or no more transactions are left to be processed in the transaction pool, the mining starts. In this process, the block is repeatedly hashed until a valid nonce is found that, once hashed with the block, results in a value less than the difficulty target.
- Once the block is successfully mined, it will be broadcasted immediately to the network, claiming success, and will be verified and accepted by the network. This process is similar to Bitcoin's mining process discussed in the previous chapter. The only difference is that Ethereum's Proof of Work algorithm is ASIC-resistant, known as **Ethash**, where finding a nonce requires large memory.



Relationship between transaction, transaction trie and block header

Contract creation transaction

There are a few essential parameters that are required when creating an account. These parameters are listed as follows:

- Sender
- Original transactor
- Available gas
- Gas price
- Endowment, which is the amount of ether allocated initially
- A byte array of arbitrary length Initialization EVM code
- Current depth of the message call/contract-creation stack (current depth means the number of items that are already there in the stack)

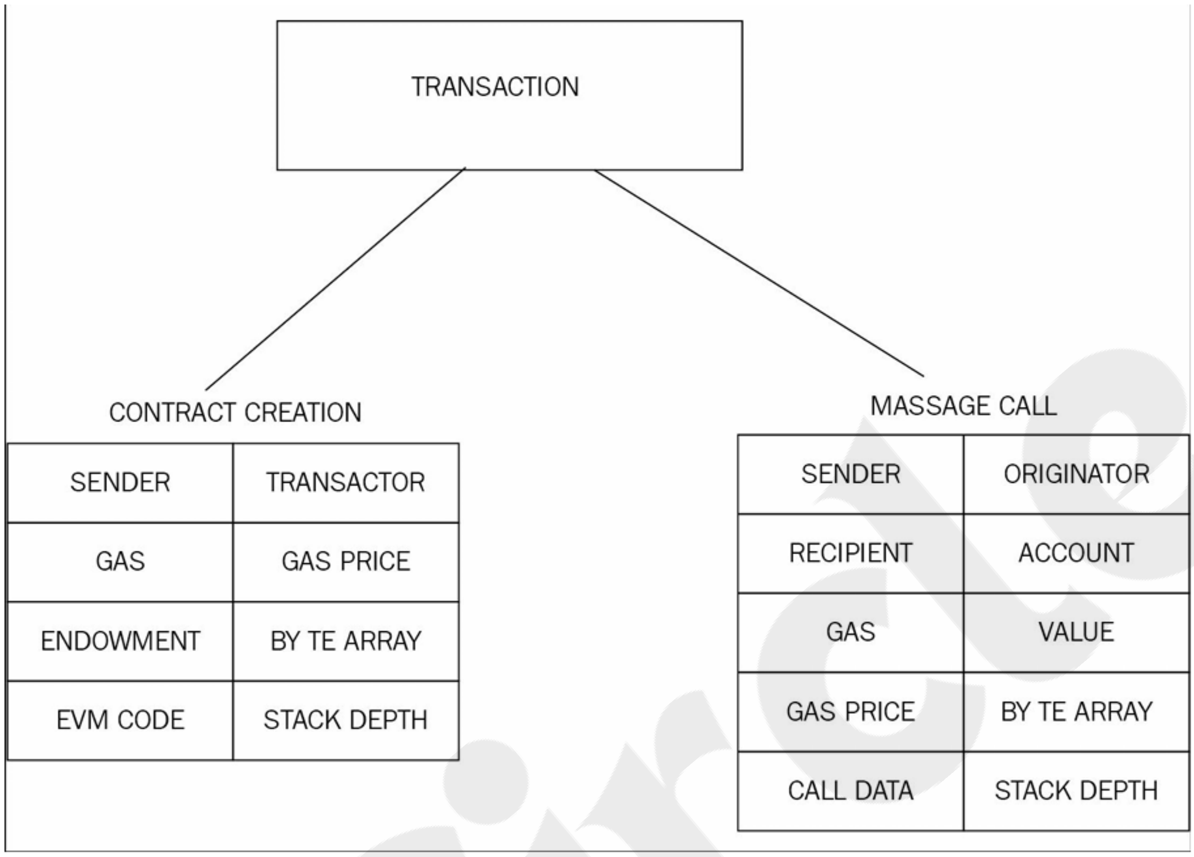
- Addresses generated as a result of contract creation transaction are 160-bit in length. Precisely, as defined in the yellow paper, they are the rightmost 160-bits of the Keccak hash of the RLP encoding of the structure containing only the sender and the nonce. Initially, the nonce in the account is set to zero. The balance of the account is set to the value passed to the contract. Storage is also set to empty. Code hash is Keccak 256-bit hash of the empty string.

- The account is initialized when the EVM code (Initialization EVM code) is executed. In the case of any exception during code execution, such as not having enough gas, the state does not change.
- If the execution is successful, then the account is created after the payment of appropriate gas costs. The current version of Ethereum (homestead) specifies that the result of contract transaction is either a new contract with its balance, or no new contract is created with no transfer of value. This is in contrast to previous versions, where the contract could be created regardless of the contract code deployment being successful or not due to an out-of-gas exception.

Message call transaction

A message call requires several parameters for execution, which are listed as follows:

- Sender
- The transaction originator
- Recipient The account whose code is to be executed
- Available gas Value
- Gas price
- Arbitrary length byte array
- Input data of the call
- Current depth of the message call/contract creation stack
- Message calls result in state transition. Message calls also produce output data, which is not used if transactions are executed. In cases where message calls are triggered by VM code, the output produced by the transaction execution is used. In the following diagram, the segregation between two types of transaction is shown

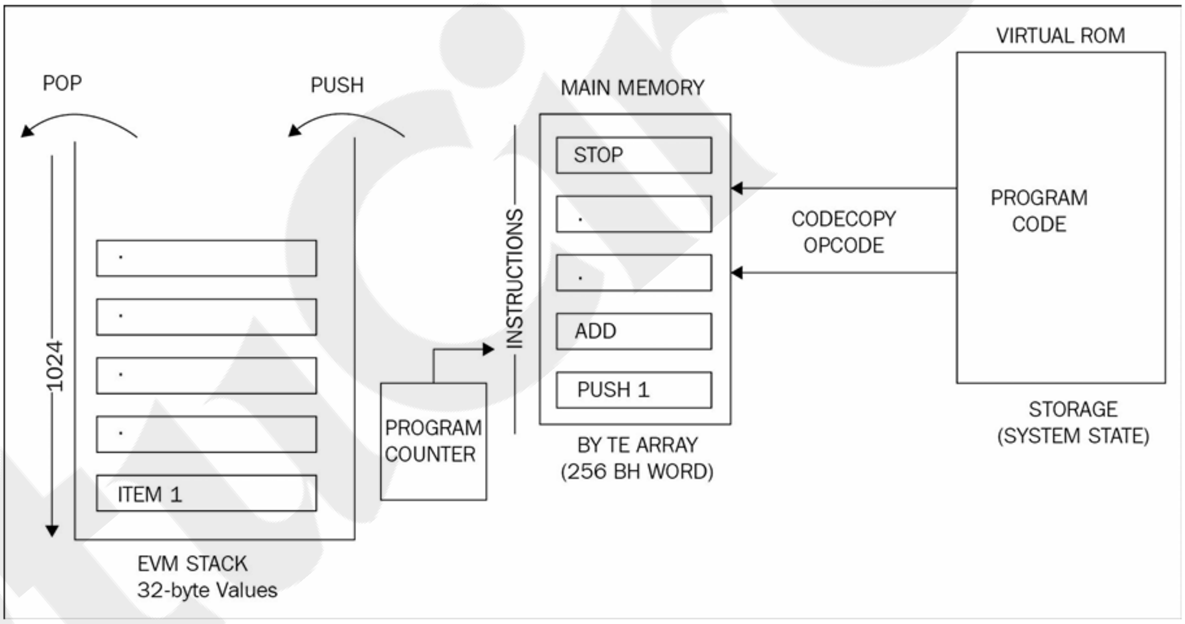


Types of transactions, required parameters for execution

Elements of the Ethereum blockchain

- Ethereum virtual machine (EVM) EVM is a simple stack-based execution machine that runs bytecode instructions in order to transform the system state from one state to another. The word size of the virtual machine is set to 256-bit.
- The stack size is limited to 1024 elements and is based on the LIFO (Last in First Out) queue. EVM is a Turing-complete machine but is limited by the amount of gas that is required to run any instruction. This means that infinite loops that can result in denial of service attacks are not possible due to gas requirements.
- EVM also supports exception handling in case exceptions occur, such as not having enough gas or invalid instructions, in which case the machine would immediately halt and return the error to the executing agent.
- EVM is a fully isolated and sandboxed runtime environment. The code that runs on the EVM does not have access to any external resources, such as a network or filesystem.
- A EVM is a stack-based architecture. EVM is big-endian by design and it uses 256-bit wide words. This word size allows for Keccak 256-bit hash and elliptic curve cryptography computations.

- There are two types of storage available to contracts and EVM. The first one is called memory, which is a **byte array**. When a contract finishes the code execution, the memory is cleared. It is akin to the concept of RAM.
- The other type, called **storage,** is permanently stored on the blockchain. It is a key value store. Memory is unlimited but constrained by gas fee requirements. The storage associated with the virtual machine is a word addressable word array that is non-volatile and is maintained as part of the system state.
- Keys and value are 32 bytes in size and storage. The program code is stored in a virtual read only memory (virtual ROM) that is accessible using the CODECOPY instruction. The CODECOPY instruction is used to copy the program code into the main memory. Initially, all storage and memory is set to zero in the EVM. The following diagram shows the design of the EVM where the virtual ROM stores the program code that is copied into main memory using CODECOPY.
- The main memory is then read by the EVM by referring to the program counter and executes instructions step by step. The program counter and EVM stack are updated accordingly with each instruction execution.



EVM operation

- EVM optimization is an active area of research and recent research has suggested that EVM can be optimized and tuned to a very fine degree in order to achieve high performance. Research into the possibility of using Web assembly (WASM) is underway already. WASM is developed by Google, Mozilla, and Microsoft and is now being designed as an open standard by the W3C community group. The aim of WASM is to be able to run machine code in the browser that will result in execution at native speed. Similarly, the aim of EVM 2.0 is to be able to run the EVM instruction set (Opcodes) natively in CPUs, thus making it faster and efficient.

Execution environment

There are some key elements that are required by the execution environment in order to execute the code. The key parameters are provided by the execution agent, for example, a transaction. These are listed as follows:

1. The address of the account that owns the executing code.
2. The address of the sender of the transaction and the originating address of this execution.
3. The gas price in the transaction that initiated the execution.
4. Input data or transaction data depending on the type of executing agent. This is a byte array; in the case of a message call, if the execution agent is a transaction, then the transaction data is included as input data.
5. The address of the account that initiated the code execution or transaction sender. This is the address of the sender in case the code execution is initiated by a transaction; otherwise, it's the address of the account.
6. The value or transaction value. This is the amount in **Wei.** If the execution agent is a transaction, then it is the transaction value.
7. The code to be executed presented as a byte array that the iterator function picks up in each execution cycle.
8. The block header of the current block
9. The number of message calls or contract creation transactions currently in execution. In other words, this is the number of CALLs or CREATEs currently in execution.

The execution environment can be visualized as a tuple of nine elements, as follows:

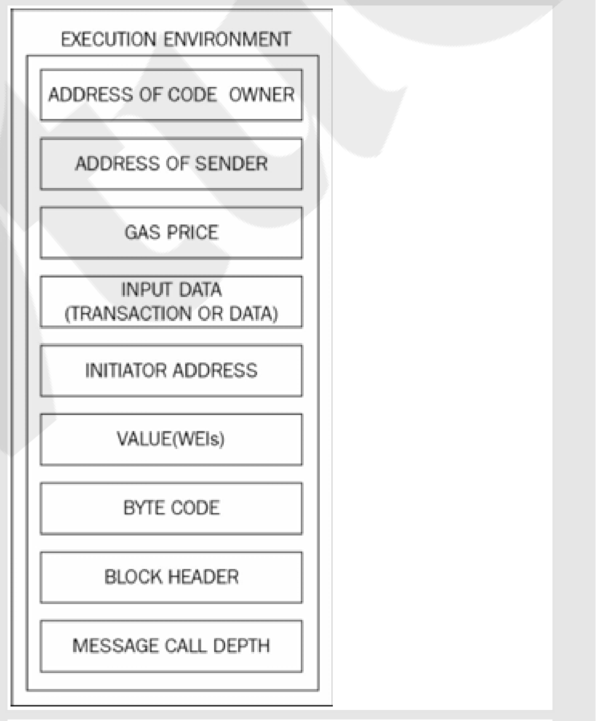


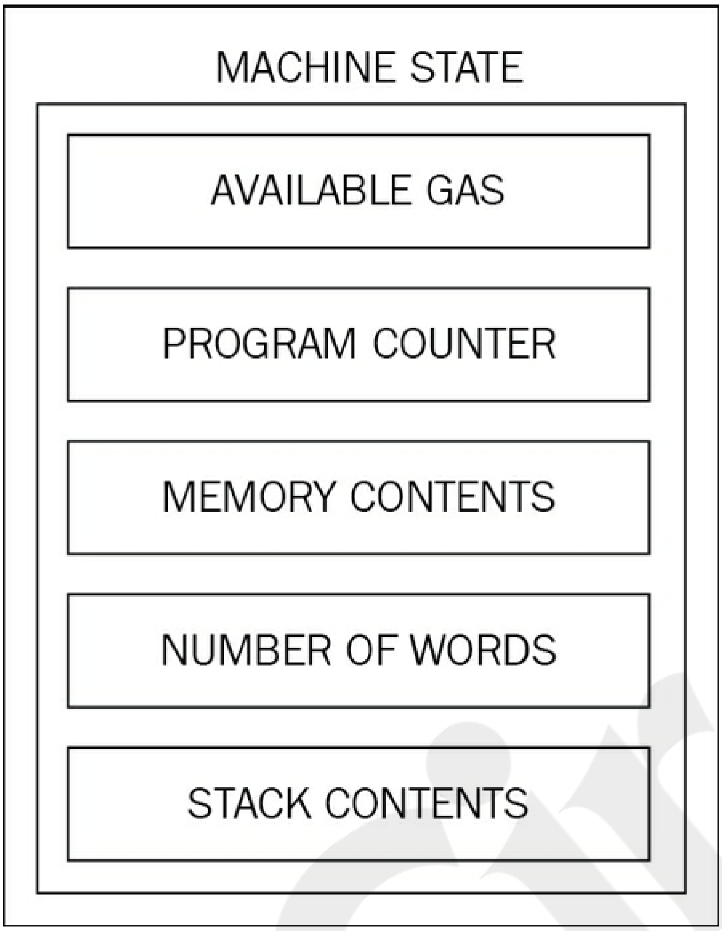
Fig: Execution environment tuple

Machine state

- Machine state is also maintained internally by the EVM. Machine state is updated after each execution cycle of EVM.
- An iterator function runs in the virtual machine, which outputs the results of a single cycle of the state machine.
- Machine state is a tuple that consist of the following elements:
    - Available gas
    - The program counter, which is a positive integer up to 256
    - Memory contents
    - Active number of words in memory
    - Contents of the stack
    - The EVM is designed to handle exceptions and will halt (stop execution) in case any of the following exceptions occur:
    - Not having enough gas required for execution Invalid instructions Insufficient stack items Invalid destination of jump op codes Invalid stack size (greater than 1024)

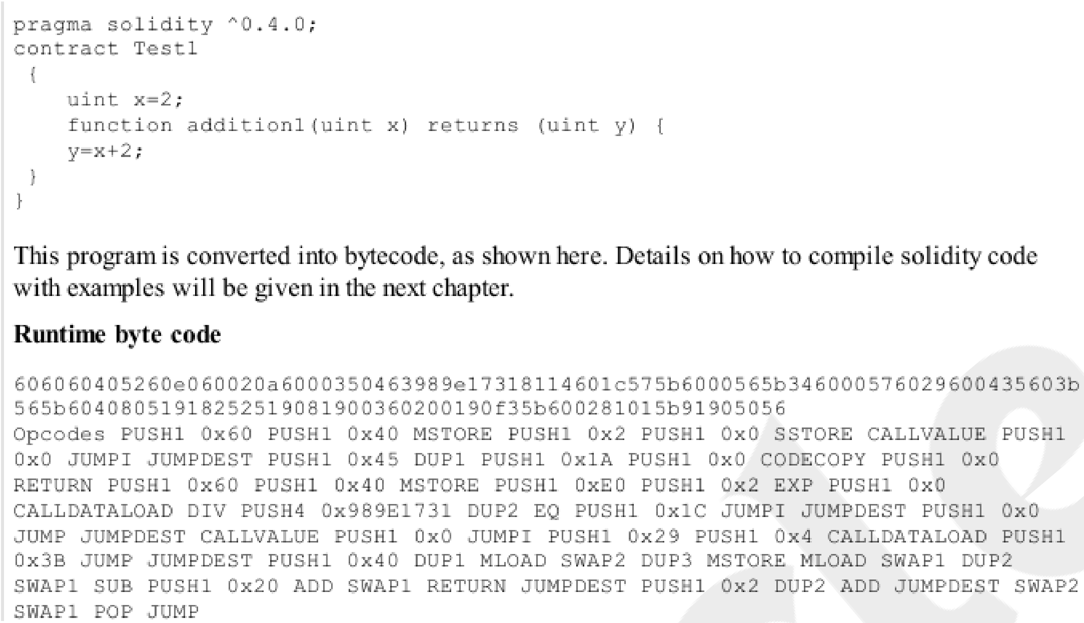
The iterator function

- The iterator function mentioned earlier performs various important functions that are used to set the next state of the machine and eventually the world state.
- These functions include the following: It fetches the next instruction from a byte array where the machine code is stored in the execution environment. It adds/removes (PUSH/POP) items from the stack accordingly.
- Gas is reduced according to the gas cost of the instructions/Opcodes. It increments the program counter (PC). Machine state can be viewed as a tuple shown in the following diagram



Machine state tuple

- the virtual machine is also able to halt in normal conditions if STOP or SUICIDE or RETURN Opcodes are encountered during the execution cycle.
- Code written in a high-level language such as serpent, LLL, or Solidity is converted into the byte code that EVM understands in order for it to be executed by the EVM.
- Solidity is the high-level language that has been developed for Ethereum with JavaScript such as syntax to write code for smart contracts.
- Once the code is written, it is compiled into byte code that's understandable by the EVM using the Solidity compiler called solc.
- LLL (Lisp-like Low-level language) is another language that is used to write smart contract code. Serpent is a Python-like high-level language that can be used to write smart contracts for Ethereum. For example, a simple program in solidity is shown as follows:



```
pragma solidity ^0.4.0;
contract Test1
 {
    uint x=2;
    function addition1(uint x) returns (uint y) {
    y=x+2;
 }
}
```

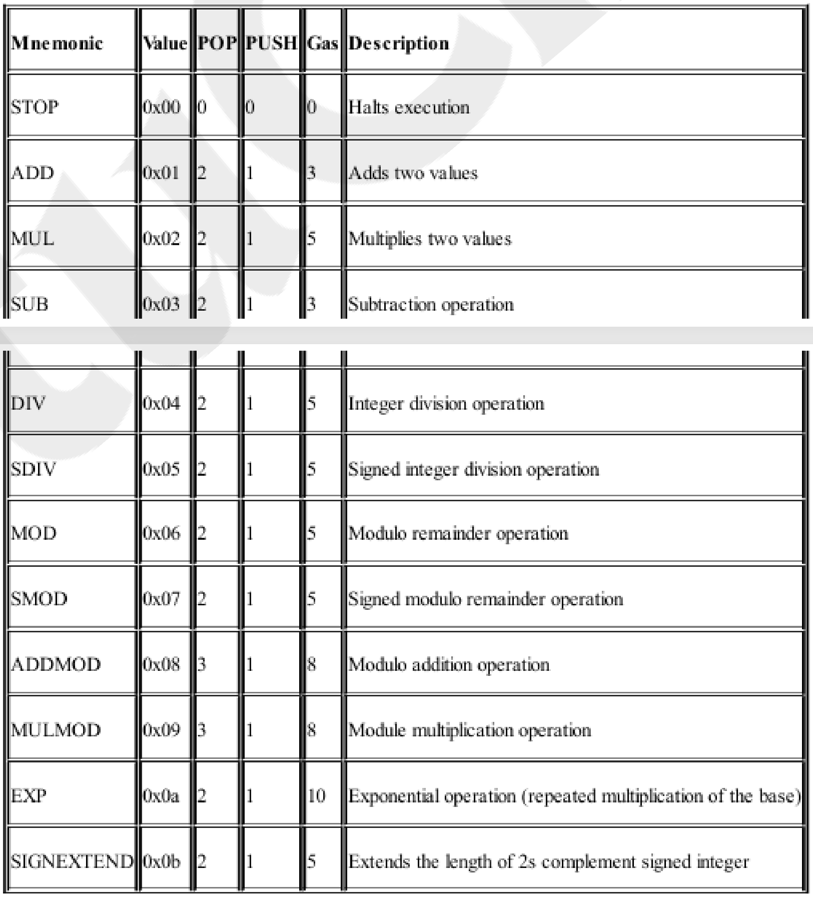
This program is converted into bytecode, as shown here. Details on how to compile solidity code with examples will be given in the next chapter.

Runtime byte code

```
606060405260e060020a6000350463989e17318114601c575b6000565b346000576029600435603b
565b60408051918252519081900360200190f35b600281015b91905056
Opcodes PUSH1 0x60 PUSH1 0x40 MSTORE PUSH1 0x2 PUSH1 0x0 SSTORE CALLVALUE PUSH1
0x0 JUMPI JUMPDEST PUSH1 0x45 DUP1 PUSH1 0x1A PUSH1 0x0 CODECOPY PUSH1 0x0
RETURN PUSH1 0x60 PUSH1 0x40 MSTORE PUSH1 0xE0 PUSH1 0x2 EXP PUSH1 0x0
CALLDATALOAD DIV PUSH4 0x989E1731 DUP2 EQ PUSH1 0x1C JUMPI JUMPDEST PUSH1 0x0
JUMP JUMPDEST CALLVALUE PUSH1 0x0 JUMPI PUSH1 0x29 PUSH1 0x4 CALLDATALOAD PUSH1
0x3B JUMP JUMPDEST PUSH1 0x40 DUP1 MLOAD SWAP2 DUP3 MSTORE MLOAD SWAP1 DUP2
SWAP1 SUB PUSH1 0x20 ADD SWAP1 RETURN JUMPDEST PUSH1 0x2 DUP2 ADD JUMPDEST SWAP2
SWAP1 POP JUMP
```

Opcodes and their meaning

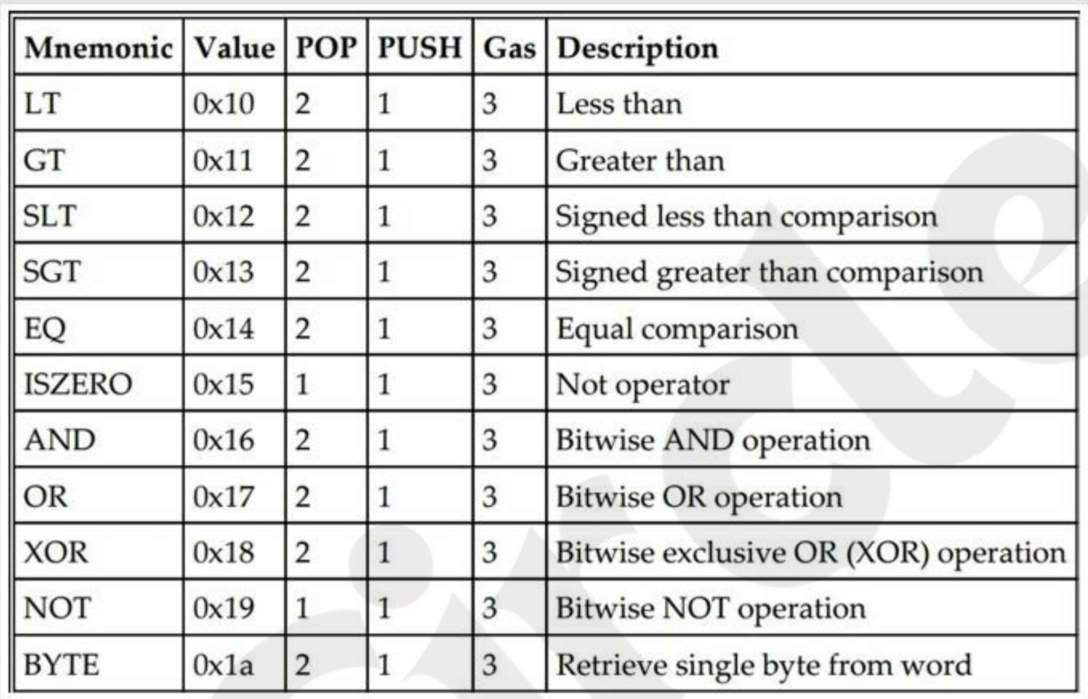
There are different opcodes that have been introduced in the EVM. Opcodes are divided into multiple categories based on the operation they perform. The list of opcodes with their meaning and usage is presented here. Arithmetic operations All arithmetic in EVM is modulo 2^256. This group of opcodes is used to perform basic arithmetic operations. The value of these operations starts from 0x00 up to 0x0b.



Mnemonic	Value	POP	PUSH	Gas	Description
STOP	0x00	0	0	0	Halts execution
ADD	0x01	2	1	3	Adds two values
MUL	0x02	2	1	5	Multiplies two values
SUB	0x03	2	1	3	Subtraction operation
DIV	0x04	2	1	5	Integer division operation
SDIV	0x05	2	1	5	Signed integer division operation
MOD	0x06	2	1	5	Modulo remainder operation
SMOD	0x07	2	1	5	Signed modulo remainder operation
ADDMOD	0x08	3	1	8	Modulo addition operation
MULMOD	0x09	3	1	8	Module multiplication operation
EXP	0x0a	2	1	10	Exponential operation (repeated multiplication of the base)
SIGNEXTEND	0x0b	2	1	5	Extends the length of 2s complement signed integer

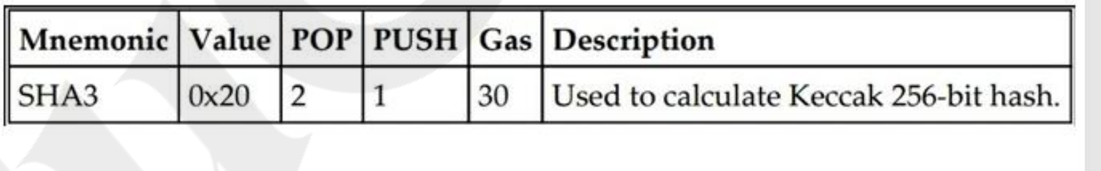
Note that STOP is not an arithmetic operation but is categorized in this list of arithmetic operations due to the range of values (0s) it falls in.

Logical operations Logical operations include operations that are used to perform comparisons and Boolean logic operations. The value of these operations is in the range of 0x10 to 0x1a



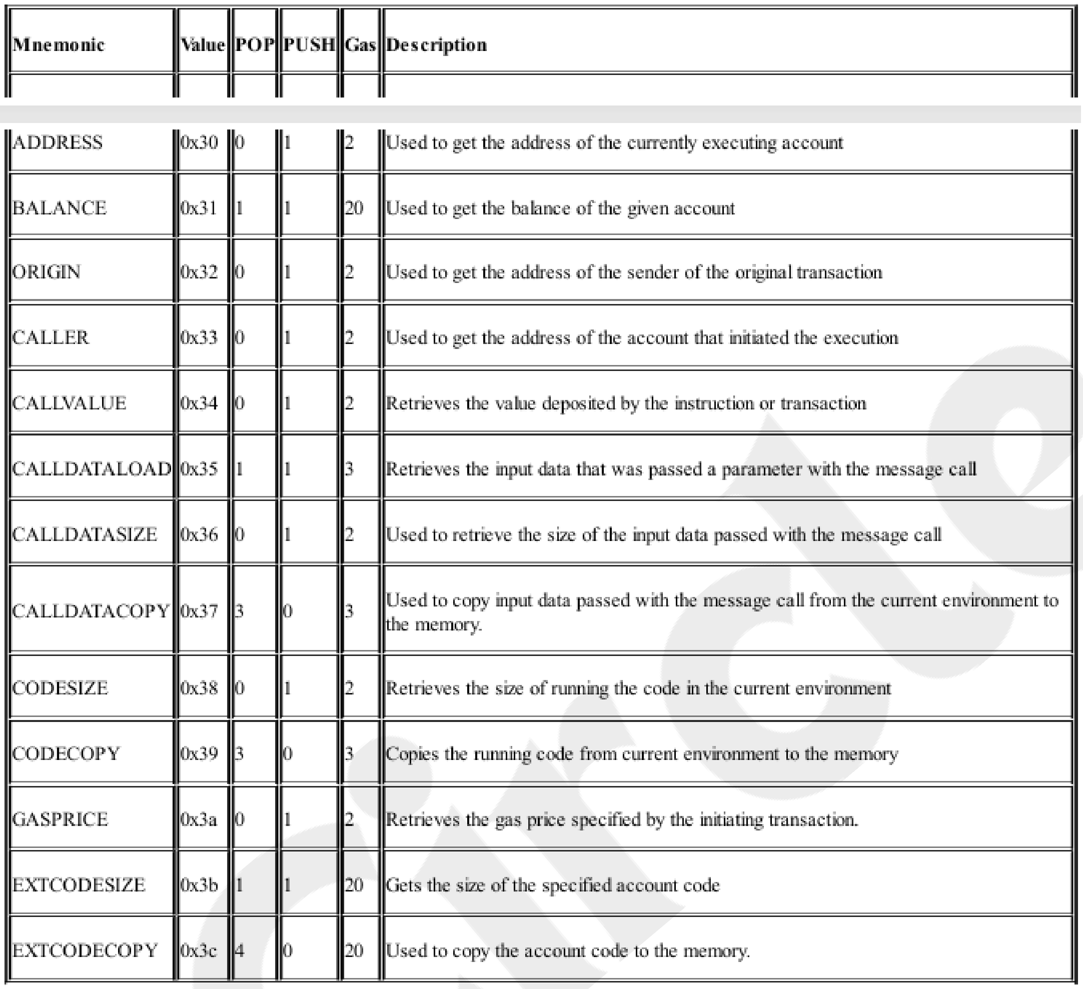
Mnemonic	Value	POP	PUSH	Gas	Description
LT	0x10	2	1	3	Less than
GT	0x11	2	1	3	Greater than
SLT	0x12	2	1	3	Signed less than comparison
SGT	0x13	2	1	3	Signed greater than comparison
EQ	0x14	2	1	3	Equal comparison
ISZERO	0x15	1	1	3	Not operator
AND	0x16	2	1	3	Bitwise AND operation
OR	0x17	2	1	3	Bitwise OR operation
XOR	0x18	2	1	3	Bitwise exclusive OR (XOR) operation
NOT	0x19	1	1	3	Bitwise NOT operation
BYTE	0x1a	2	1	3	Retrieve single byte from word

Cryptographic operations There is only one operation in this category named SHA3. It is worth noting that this is not the standard SHA3 standardized by NIST but the original Keccak implementation.



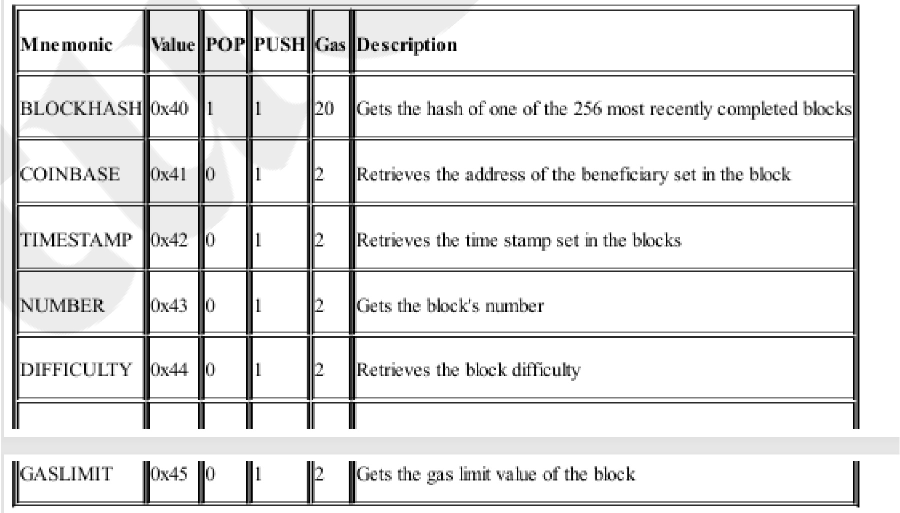
Mnemonic	Value	POP	PUSH	Gas	Description
SHA3	0x20	2	1	30	Used to calculate Keccak 256-bit hash.

Environmental information There are a total of 13 instructions in this category. These opcodes are used to provide information related to addresses, runtime environments, and data copy operations.

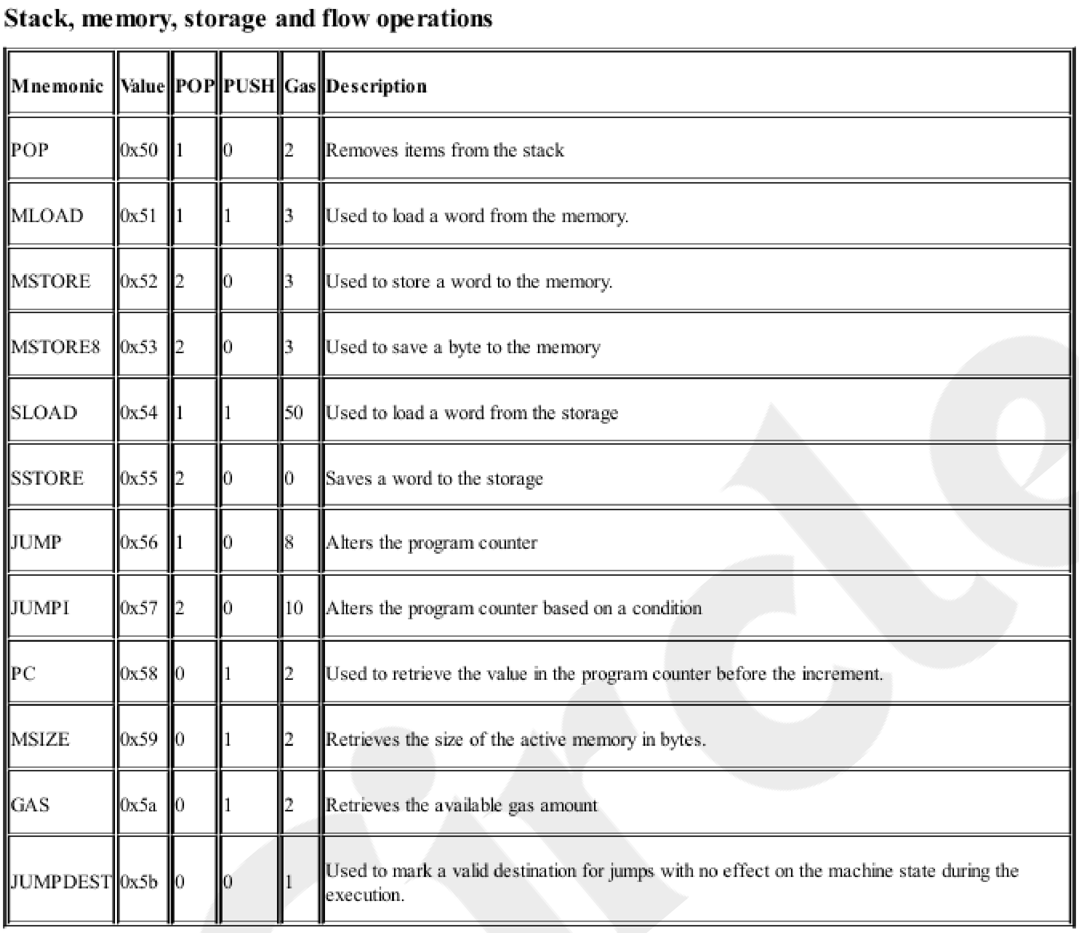


Mnemonic	Value	POP	PUSH	Gas	Description
ADDRESS	0x30	0	1	2	Used to get the address of the currently executing account
BALANCE	0x31	1	1	20	Used to get the balance of the given account
ORIGIN	0x32	0	1	2	Used to get the address of the sender of the original transaction
CALLER	0x33	0	1	2	Used to get the address of the account that initiated the execution
CALLVALUE	0x34	0	1	2	Retrieves the value deposited by the instruction or transaction
CALLDATALOAD	0x35	1	1	3	Retrieves the input data that was passed a parameter with the message call
CALLDATASIZE	0x36	0	1	2	Used to retrieve the size of the input data passed with the message call
CALLDATACOPY	0x37	3	0	3	Used to copy input data passed with the message call from the current environment to the memory.
CODESIZE	0x38	0	1	2	Retrieves the size of running the code in the current environment
CODECOPY	0x39	3	0	3	Copies the running code from current environment to the memory
GASPRICE	0x3a	0	1	2	Retrieves the gas price specified by the initiating transaction.
EXTCODESIZE	0x3b	1	1	20	Gets the size of the specified account code
EXTCODECOPY	0x3c	4	0	20	Used to copy the account code to the memory.

Block Information: This set of instructions is related to retrieving various attributes associated with a block:



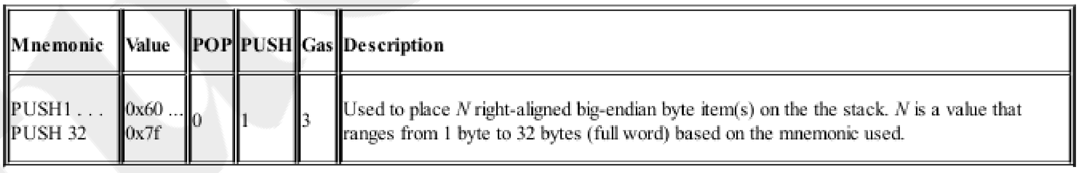
Mnemonic	Value	POP	PUSH	Gas	Description
BLOCKHASH	0x40	1	1	20	Gets the hash of one of the 256 most recently completed blocks
COINBASE	0x41	0	1	2	Retrieves the address of the beneficiary set in the block
TIMESTAMP	0x42	0	1	2	Retrieves the time stamp set in the blocks
NUMBER	0x43	0	1	2	Gets the block's number
DIFFICULTY	0x44	0	1	2	Retrieves the block difficulty
GASLIMIT	0x45	0	1	2	Gets the gas limit value of the block



Stack, memory, storage and flow operations

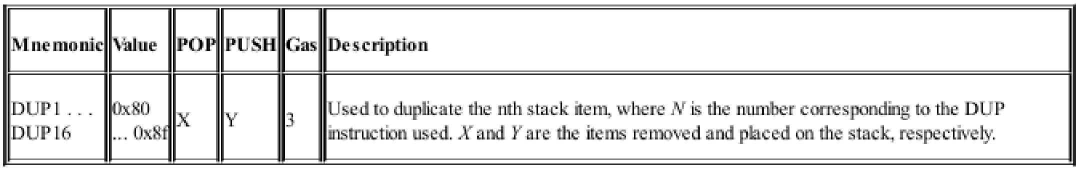
Mnemonic	Value	POP	PUSH	Gas	Description
POP	0x50	1	0	2	Removes items from the stack
MLOAD	0x51	1	1	3	Used to load a word from the memory.
MSTORE	0x52	2	0	3	Used to store a word to the memory.
MSTORE8	0x53	2	0	3	Used to save a byte to the memory
SLOAD	0x54	1	1	50	Used to load a word from the storage
SSTORE	0x55	2	0	0	Saves a word to the storage
JUMP	0x56	1	0	8	Alters the program counter
JUMPI	0x57	2	0	10	Alters the program counter based on a condition
PC	0x58	0	1	2	Used to retrieve the value in the program counter before the increment.
MSIZE	0x59	0	1	2	Retrieves the size of the active memory in bytes.
GAS	0x5a	0	1	2	Retrieves the available gas amount
JUMPDEST	0x5b	0	0	1	Used to mark a valid destination for jumps with no effect on the machine state during the execution.

Push operations These operations include PUSH operations that are used to place items on the stack. The range of these instructions is from 0x60 to 0x7f. There are 32 PUSH operations available in total in the EVM. PUSH operation, which reads from the byte array of the program code.



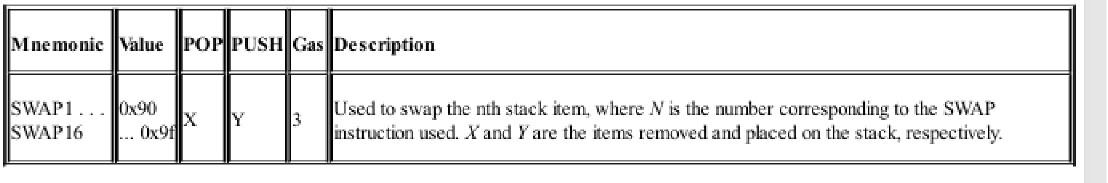
Mnemonic	Value	POP	PUSH	Gas	Description
PUSH1 . . . PUSH 32	0x60 ... 0x7f	0	1	3	Used to place *N* right-aligned big-endian byte item(s) on the the stack. *N* is a value that ranges from 1 byte to 32 bytes (full word) based on the mnemonic used.

Duplication operations As the name suggests, duplication operations are used to duplicate stack items. The range of values is from 0x80 to 0x8f. There are 16 DUP instructions available in the EVM. Items placed on the stack or removed from the stack also change incrementally with the mnemonic used; for example, DUP1 removes one item from the stack and places two items on the stack, whereas DUP16 removes 16 items from the stack and places 17 items.



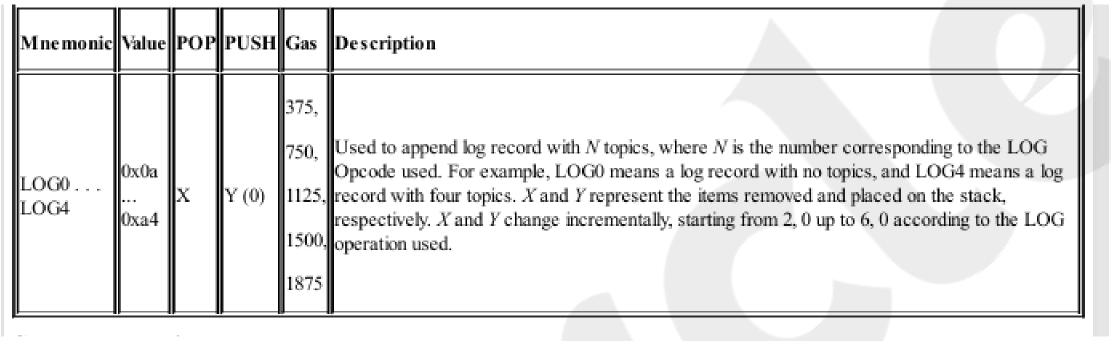
Mnemonic	Value	POP	PUSH	Gas	Description
DUP1 . . . DUP16	0x80 ... 0x8f	X	Y	3	Used to duplicate the nth stack item, where *N* is the number corresponding to the DUP instruction used. *X* and *Y* are the items removed and placed on the stack, respectively.

Exchange operations SWAP operations provide the ability to exchange stack items. There are 16 SWAP instructions available and with each instruction, the stack items are removed and placed incrementally up to 17 items depending on the type of Opcode used.



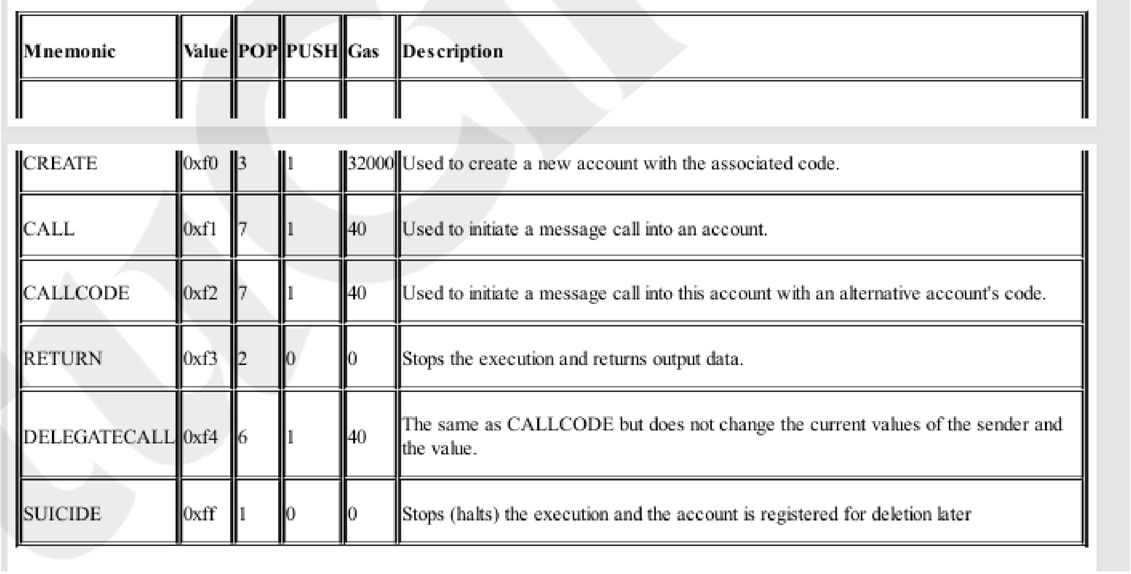
Mnemonic	Value	POP	PUSH	Gas	Description
SWAP1 . . . SWAP16	0x90 ... 0x9f	X	Y	3	Used to swap the nth stack item, where *N* is the number corresponding to the SWAP instruction used. *X* and *Y* are the items removed and placed on the stack, respectively.

Logging operations Logging operations provide opcodes to append log entries on the sub-state tuple's log series field. There are four log operations available in total and they range from value 0x0a to 0xa4.



Mnemonic	Value	POP	PUSH	Gas	Description
LOG0 . . . LOG4	0x0a ... 0xa4	X	Y (0)	375, 750, 1125, 1500, 1875	Used to append log record with *N* topics, where *N* is the number corresponding to the LOG Opcode used. For example, LOG0 means a log record with no topics, and LOG4 means a log record with four topics. *X* and *Y* represent the items removed and placed on the stack, respectively. *X* and *Y* change incrementally, starting from 2, 0 up to 6, 0 according to the LOG operation used.

System operations System operations are used to perform various system-related operations, such as account creation, message calling, and execution control. There are six Opcodes available in total in this category.



Mnemonic	Value	POP	PUSH	Gas	Description
CREATE	0xf0	3	1	32000	Used to create a new account with the associated code.
CALL	0xf1	7	1	40	Used to initiate a message call into an account.
CALLCODE	0xf2	7	1	40	Used to initiate a message call into this account with an alternative account's code.
RETURN	0xf3	2	0	0	Stops the execution and returns output data.
DELEGATECALL	0xf4	6	1	40	The same as CALLCODE but does not change the current values of the sender and the value.
SUICIDE	0xff	1	0	0	Stops (halts) the execution and the account is registered for deletion later

Precompiled contracts There are four precompiled contracts in Ethereum. Here is the list of these contracts and details:

1. **The elliptic curve public key recovery function ECDSARECOVER (Elliptic curve DSA recover function**) is available at address 1. It is denoted as ECREC and requires 3000 gas for execution. If the signature is invalid, then no output is returned by this function. Public key recovery is a standard mechanism by which the public key can be derived from the private key in elliptic curve cryptography. The ECDSA recovery function is shown as follows: ECDSARECOVER(H, V, R, S) = Public Key It takes four

inputs: H, which is a 32 byte hash of the message to be signed and V, R, and S, which represent the ECDSA signature with the recovery ID and produce a 64 byte public key. V, R, and S

2. **The SHA-256 bit hash function** The SHA-256 bit hash function is a precompiled contract that is available at address 2 and produces a SHA256 hash of the input. It is almost like a pass-through function. Gas requirement for SHA-256 (SHA256) depends on the input data size. The output is a 32-byte value

3. **The RIPEMD-160 bit hash function** The RIPEMD-160 bit hash function is used to provide RIPEMD 160-bit hash and is available at address 3. The output of this function is a 20-byte value. Gas requirement, similar to SHA-256, is dependent on the amount of input data.

4. **The identity function** the identity function is available at address 4 and is denoted by the ID. It simply defines output as input; in other words, whatever input is given to the ID function, it will output the same value. Gas requirement is calculated by a simple formula: 15 + 3 [Id/32] where Id is the input data. This means that at a high level, the gas requirement is dependent on the size of the input data albeit with some calculation performed, as shown in the preceding equation. All the previously mentioned precompiled contracts can become native extensions and can be included in the EVM opcodes in the future

Accounts

Accounts are one of the main building blocks of the Ethereum blockchain. The state is created or updated as a result of the interaction between accounts. Operations performed between and on the accounts represent state transitions. State transition is achieved using what's called the Ethereum state transition function, which works as follows:

1. Confirm the transaction validity by checking the syntax, signature validity, and nonce.

2. Transaction fee is calculated and the sending address is resolved using the signature. Furthermore, sender's account balance is checked and subtracted accordingly and nonce is incremented. An error is returned if the account balance is not enough.

3. Provide enough ether (gas price) to cover the cost of the transaction. This is charged per byte incrementally according to the size of the transaction.

4. In this step, the actual transfer of value occurs. The flow is from the sender's account to receiver's account. The account is created

automatically if the destination account specified in the transaction does not exist yet. Moreover, if the destination account is a contract, then the contract code is executed. This also depends on the amount of gas available. If enough gas is available, then the contract code will be executed fully; otherwise, it will run up to the point where it runs out of gas.

5. In cases of transaction failure due to insufficient account balance or gas, all state changes are rolled back with the exception of fee payment, which is paid to the miners.

6. Finally, the remainder (if any) of the fee is sent back to the sender as change and fee is paid to the miners accordingly. At this point, the function returns the resulting state.

Types of accounts

There are two types of accounts in Ethereum: **Externally owned accounts Contract accounts** The first is externally owned accounts **(EOAs)** and the other is **contract accounts.**

- EOAs are similar to accounts that are controlled by a private key in bitcoin.
- Contract accounts are the accounts that have code associated with them along with the private key.
- An EOA has ether balance, is able to send transactions, and has no associated code, whereas a Contract Account (CA) has ether balance, associated code, and the ability to get triggered and execute code in response to a transaction or a message.
- It is worth noting that due to the Turing-completeness property of the Ethereum blockchain, the code within contract accounts can be of any level of complexity.
- The code is executed by EVM by each mining node on the Ethereum network.
- In addition, contract accounts are able to maintain their own permanent state and can call other contracts. It is envisaged that in the serenity release, the distinction between externally owned accounts and contract accounts may be eliminated.

Block

- blocks are the main building blocks of a blockchain. Ethereum blocks consist of various components, which are described as follows:
  1.The block header
  2.The transactions list
  3.The list of headers of **Ommers or Uncles**
  The transaction list is simply a list of all transactions included in the block. In addition, the list of headers of Uncles is also included in the block. The most important and complex part is the **block header**

Block header

Block headers are the most critical and detailed components of an Ethereum block. The header contains valuable information,

PARENT HASH

This is the Keccak 256-bit hash of the parent (previous) block's header.

OMMERS HASH

This is the Keccak 256-bit hash of the list of Ommers (Uncles) blocks included in the block.

BENEFICIARY

Beneficiary field contains the 160-bit address of the recipient that will receive the mining reward once the block is successfully mined.

STATE ROOT

The state root field contains the Keccak 256-bit hash of the root node of the state trie. It is calculated after all transactions have been processed and finalized.

TRANSACTIONS ROOT

The transaction root is the Keccak 256-bit hash of the root node of the transaction trie. Transaction trie represents the list of transactions included in the block.

RECEIPTS ROOT

The receipts root is the keccak 256 bit hash of the root node of the transaction receipt trie. This trie is composed of receipts of all transactions included in the block. Transaction receipts are generated after each transaction is processed and contain useful post transaction information.

LOGS BLOOM

The logs bloom is a bloom filter that is composed of the logger address and log topics from the log entry of each transaction receipt of the included transaction list in the block.

DIFFICULTY

The difficulty level of the current block.

NUMBER

The total number of all previous blocks; the genesis block is block zero.

GAS LIMIT

Th***e*** field contains the value that represents the limit set on the gas consumption per block.

GAS USED

The field contains the total gas consumed by the transactions included in the block.

TIMESTAMP

Timestamp is the epoch Unix time of the time of block initialization.

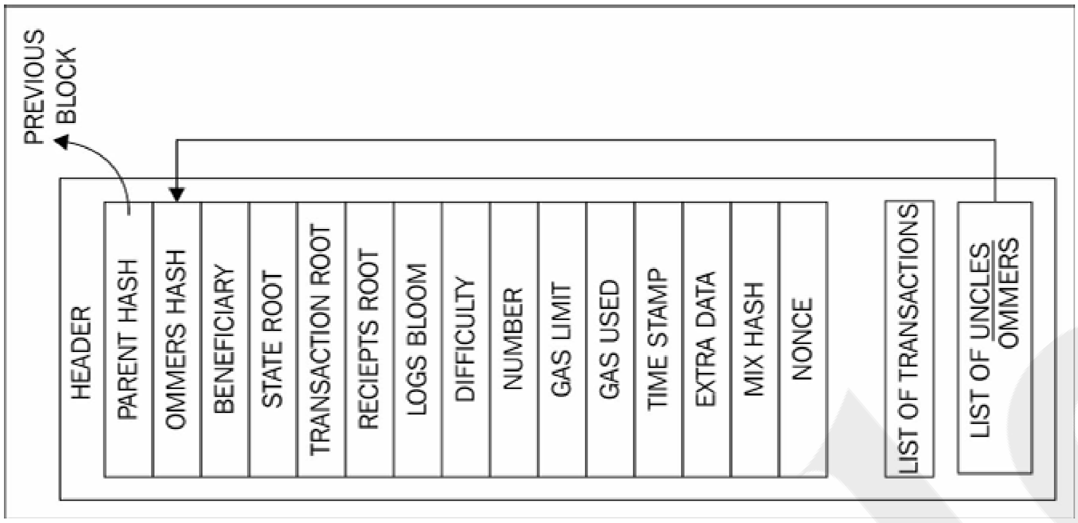
EXTRA DATA Extra data field can be used to store arbitrary data related to the block.

MIXHASH

Mixhash field contains a 256-bit hash that once combined with the nonce is used to prove that adequate computational effort has been spent in order to create this block.

NONCE

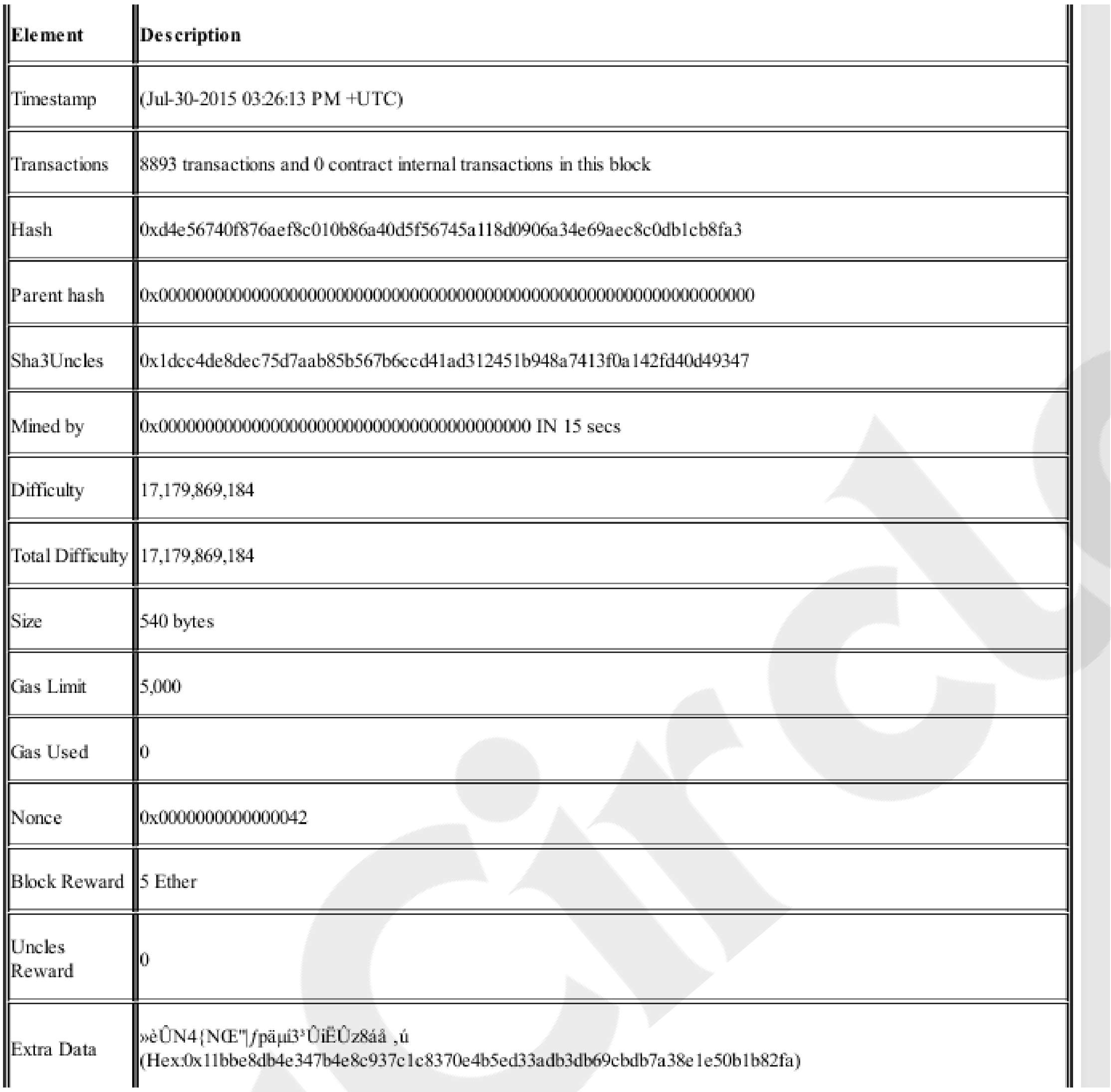
Nonce is a 64-bit hash (a number) that is used to prove, in combination with the mixhash field, that adequate computational effort has been spent in order to create this block. The following figure shows the detailed structure of the block and block header



Detailed diagram of block structure with block header

The genesis block

The genesis block varies slightly with regard to the data it contains and the way it has been created from a normal block.

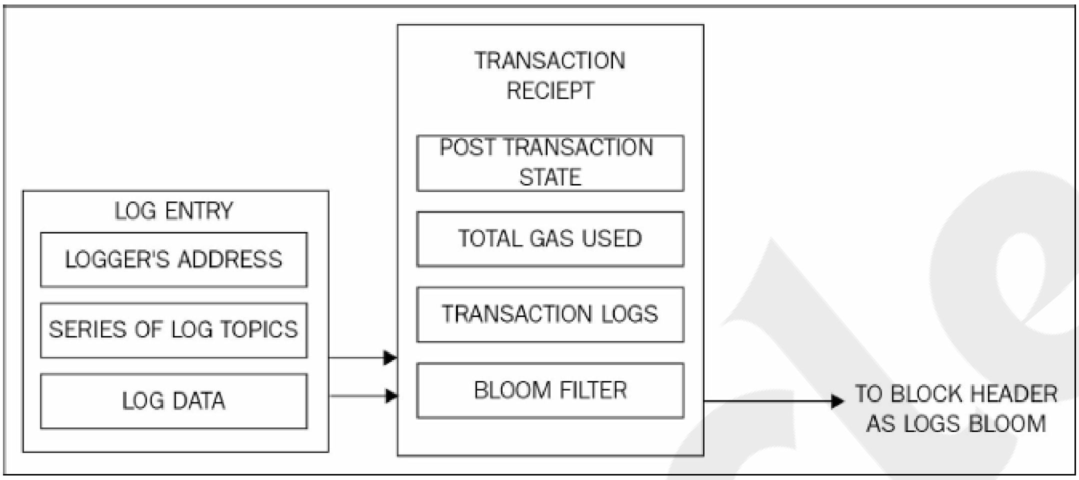


Element	Description
Timestamp	(Jul-30-2015 03:26:13 PM +UTC)
Transactions	8893 transactions and 0 contract internal transactions in this block
Hash	0xd4e56740f876aef8c010b86a40d5f56745a118d0906a34e69aec8c0db1cb8fa3
Parent hash	0x0000000000000000000000000000000000000000000000000000000000000000
Sha3Uncles	0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
Mined by	0x0000000000000000000000000000000000000000 IN 15 secs
Difficulty	17,179,869,184
Total Difficulty	17,179,869,184
Size	540 bytes
Gas Limit	5,000
Gas Used	0
Nonce	0x0000000000000042
Block Reward	5 Ether
Uncles Reward	0
Extra Data	»èÛN4{NŒ"∣ƒpäµi3³ÛiËÛz8áâ ,ú (Hex:0x11bbe8db4e347b4e8c937c1c8370e4b5ed33adb3db69cbdb7a38e1e50b1b82fa)

Transaction receipts Transaction receipts are used as a mechanism to store the state after a transaction has been executed. In other words, these structures are used to record the outcome of the transaction execution. It is produced after the execution of each transaction. All receipts are stored in an index-keyed trie. Hash (Keccak 256-bit) of the root of this trie is placed in the block header as the receipts root. It is composed of four elements that are described here.

1. **The post-transaction** state This item is a trie structure that holds the state after the transaction has executed. It is encoded as a byte array.
2. **Gas used** This item represents the total amount of gas used in the block that contains the transaction receipt. The value is taken immediately after the transaction execution is completed. The total gas used is expected to be a non-negative integer.
3. **Set of logs** This field shows the set of log entries created as a result of transaction execution. Log entries contain the logger's address, a series of log topics, and the log data.
4. **The bloom filter** A bloom filter is created from the information contained in the set of logs .A log entry is reduced to a hash of 256 bytes, which is then

embedded in the header of the block as the logs bloom. Log entry is composed of the logger's address and log topics and log data. Log topics are encoded as a series of 32 byte data structures. Log data is made up of a few bytes of data. This process can be visualized in the following diagram

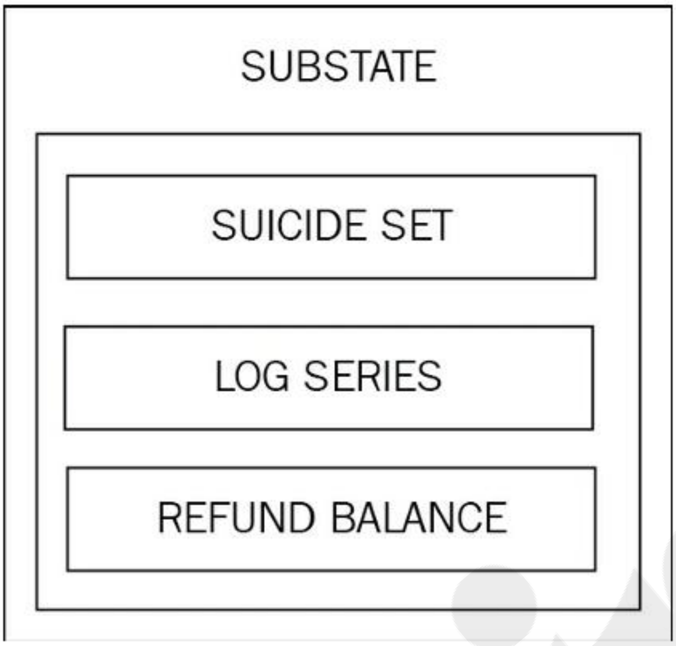


Transaction receipts and logs bloom

Transaction validation and execution Transactions are executed after verifying the transactions for validity. Initial tests are listed as follows:

- **A transaction** must be well-formed and RLP-encoded without any additional trailing bytes The digital signature used to sign the transaction is valid Transaction nonce must be equal to the sender's account's current nonce Gas limit must not be less than the gas used by the transaction The sender's account contains enough balance to cover the execution cost
- **The transaction sub state** A transaction sub-state is created during the execution of the transaction that is processed immediately after the execution completes. This transaction sub-state is a tuple that is composed of three items.
- **Suicide set** This element contains the list of accounts that are disposed of after the transaction is executed.
- **Log series** This is an indexed series of checkpoints that allow the monitoring and notification of contract calls to the entities external to the Ethereum environment, such as application frontends. It works like a trigger mechanism that is executed every time a specific function is invoked or a specific event occurs. Logs are created in response to events occurring in the smart contract. It can also be used as a cheaper form of storage.

- **Refund balance** This is the total price of gas in the transaction that initiated the execution. Refunds are not immediately executed; instead, they are used to partially offset the total execution cost.



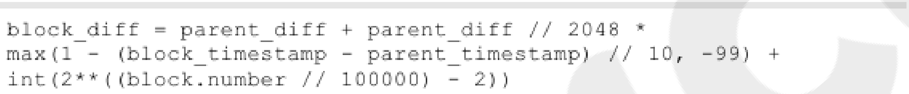
Sub-state tuple

The block validation mechanism

An Ethereum block is considered valid if it passes the following checks:

- **Consistent with Uncles and transactions.** This means that all Ommers (Uncles) satisfy the property that they are indeed Uncles and also if the Proof of Work for Uncles is valid.
- If the previous block (parent) exists and is valid.
- If the timestamp of the block is valid. This basically means that the current block's timestamp must be higher than the parent block's timestamp. Also, it should be less than 15 minutes into the future. All block times are calculated in epoch time (Unix time). If any of these checks fails, the block will be rejected.
  - **Block finalization** Block finalization is a process that is run by miners in order to validate the contents of the block and apply rewards. It results in four steps being executed. These steps are described here in detail.
  - **Ommers validation** Validate Ommers (stale blocks also called Uncles). In the case of mining, determine Ommers. The validation process of the headers of stale blocks checks whether the header is valid and the relationship of the Uncle with the current block satisfies the maximum depth of six blocks. A block can contain a maximum of two Uncles.

- **Transaction validation** Validate transactions. In the case of mining, determine transactions. The process involves checking whether the total gas used in the block is equal to the final gas consumption after the final transaction.
- **Reward application** Apply rewards, which means updating the beneficiary's account with a reward balance. In Ethereum, a reward is also given to miners for stale blocks, which is 1/32 of the block reward. Uncles that are included in the blocks also receive 7/8 of the total block reward. The current block reward is 5 Ether. A block can have a maximum of two Uncles.
- **State and nonce validation** Verify the state and nonce. In the case of mining, compute a valid state and nonce.
- **Block difficulty** Block difficulty is increased if the time between two blocks decreases, whereas it increases if the block time between two blocks decreases. This is required to maintain a roughly consistent block generation time. The difficulty adjustment algorithm in Ethereum's homestead release is shown as follows.

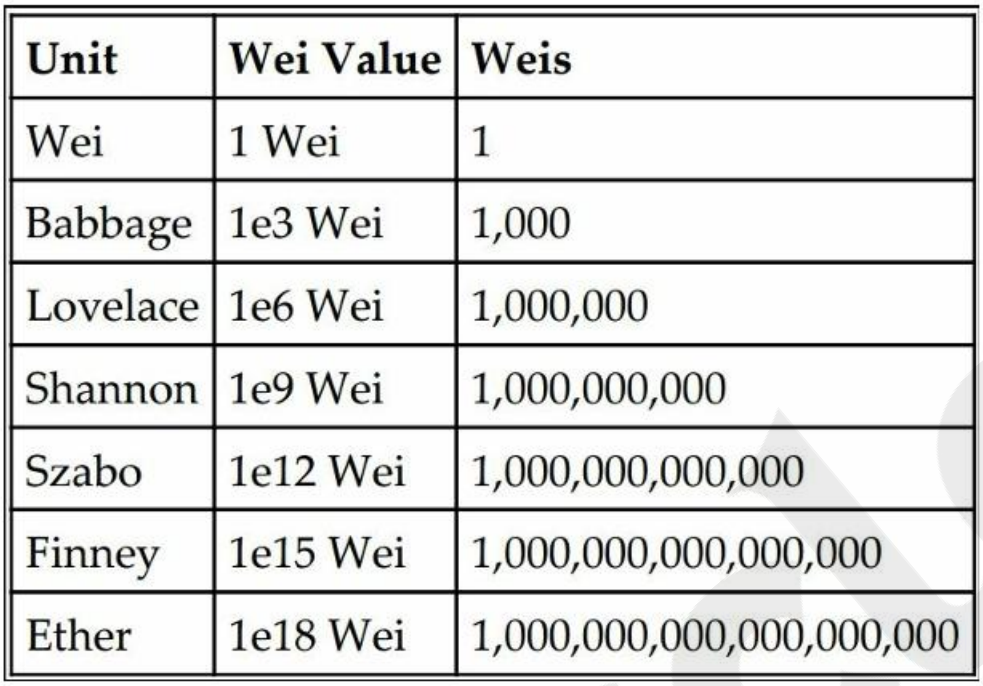


```
block_diff = parent_diff + parent_diff // 2048 *
max(1 - (block_timestamp - parent_timestamp) // 10, -99) +
int(2**((block.number // 100000) - 2))
```

The preceding algorithm means that, if the time difference between the generation of the parent block and the current block is less than 10 seconds, the difficulty goes up. If the time difference is between 10 to 19 seconds, the difficulty level remains the same. Finally, if the time difference is 20 seconds or more, the difficultly level decreases. This decrease is proportional to the time difference. In addition to timestamp-difference-based difficulty adjustment, there is also another part (shown in the last line of the preceding algorithm) that increases the difficulty exponentially after every 100,000 blocks. This is the so called difficulty time bomb or Ice age introduced in the Ethereum network, which will make it very hard to mine on the Ethereum blockchain at some point in the future. This will encourage users to switch to Proof of Stake as mining on the POW chain will eventually become prohibitively difficult. According to the latest update and estimates based on the algorithm, the block generation time will become significantly high during the second half of the year 2017 and in 2021, it will become so high that it will be virtually impossible to mine on the POW chain. This way, miners will have no choice but to switch to the Proof of Stake scheme proposed by Ethereum called Casper.

Ether

Ether is minted by miners as a currency reward for the computational effort they spend in order to secure the network by verifying and with validation transactions and blocks. Ether is used within the Ethereum blockchain to pay for the execution of contracts on the EVM. Ether is used to purchase gas as crypto fuel, which is required in order to perform computation on the Ethereum blockchain. The denomination table is shown as follows



Unit	Wei Value	Weis
Wei	1 Wei	1
Babbage	1e3 Wei	1,000
Lovelace	1e6 Wei	1,000,000
Shannon	1e9 Wei	1,000,000,000
Szabo	1e12 Wei	1,000,000,000,000
Finney	1e15 Wei	1,000,000,000,000,000
Ether	1e18 Wei	1,000,000,000,000,000,000

Fees are charged for each computation performed by the EVM on the blockchain

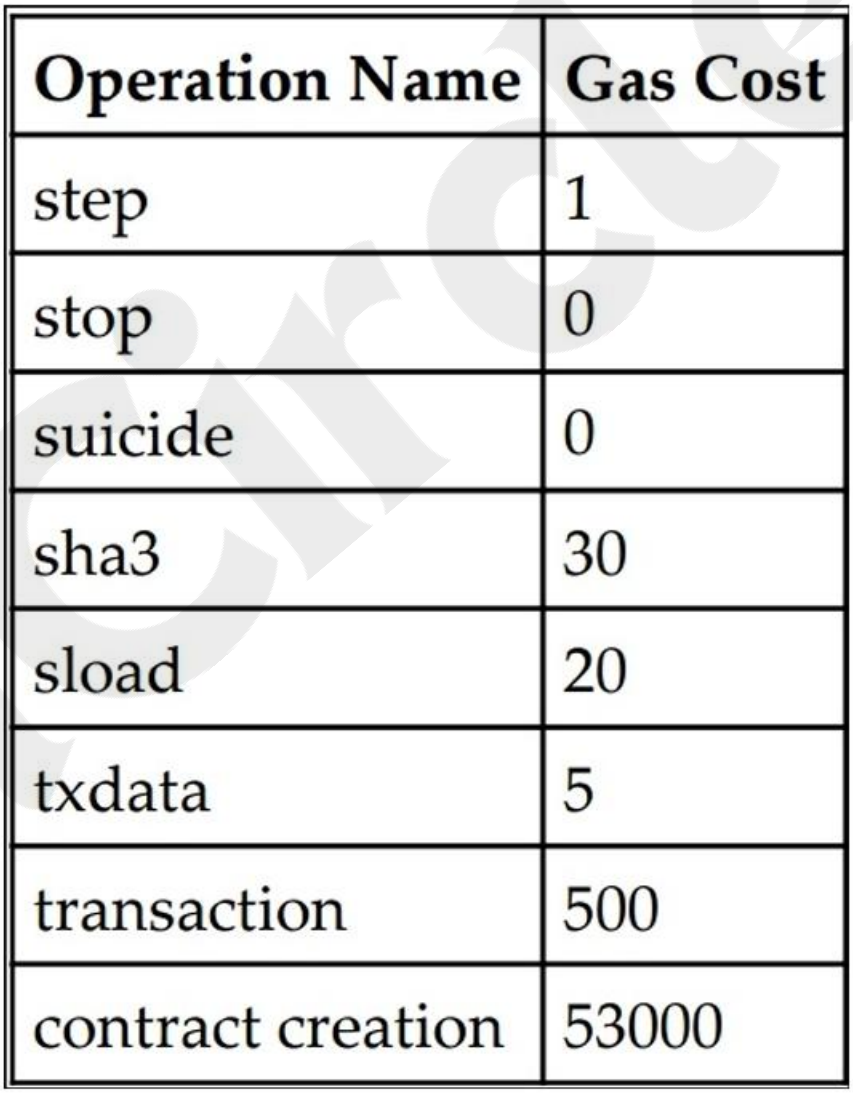
Gas

- Gas is required to be paid for every operation performed on the ethereum blockchain. This is a mechanism that ensures that infinite loops cannot cause the whole blockchain to stall due to the Turing-complete nature of the EVM.
- A transaction fee is charged as some amount of Ether and is taken from the account balance of the transaction originator. A fee is paid for transactions to be included by miners for mining.
- If this fee is too low, the transaction may never be picked up; the more the fee, the higher are the chances that the transactions will be picked up by the miners for inclusion in the block.
- Conversely, if the transaction that has an appropriate fee paid is included in the block by miners but has too many complex operations to perform, it can result in an out-of gas exception if the gas cost is not enough. In this case, the transaction will fail but will still be made part of the block and the transaction originator will not get any refund. Transaction cost can be estimated using the following formula:

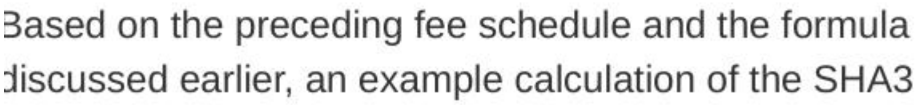
  Total cost = gasUsed * gasPrice

Here, gasUsed is the total gas that is supposed to be used by the transaction during the execution and gasPrice is specified by the transaction originator as an incentive to the miners to include the transaction in the next block. This is specified in Ether. Each EVM opcode has a fee assigned to it. It is an estimate because the gas used can be more or less than the value specified by the transaction originator originally.

- For example, if computation takes too long or the behavior of the smart contract changes in response to some other factors, then the transaction execution may perform more or less operations than originally intended and can result in consuming more or fewer gas. If the execution runs out of gas, everything is immediately rolled back; otherwise, if the execution is successful and there is some remaining gas, then it is returned to the transaction originator. Each operation costs some gas; a high level fee schedule of a few operations is shown as an example here



Operation Name	Gas Cost
step	1
stop	0
suicide	0
sha3	30
sload	20
txdata	5
transaction	500
contract creation	53000



Based on the preceding fee schedule and the formula discussed earlier, an example calculation of the SHA3

operation can be calculated as follows: SHA3 costs 30 gas Current gas price is 25 GWei, which is 0.000000025 Ether Multiplying both: 0.000000025 * 30 = 0.00000075 Ether In total, 0.00000075 Ether is the total gas that will be charged. Fee schedule Gas is charged in three scenarios as a prerequisite to the execution of an operation: The computation of an operation For contract creation or message call Increase in the usage of memory.

Messages

- Messages, as defined in the yellow paper, are the data and value that are passed between two accounts.
- A message is a data packet passed between two accounts. This data packet contains data and value (amount of ether). It can either be sent via a smart contract (autonomous object) or from an external actor (externally owned account) in the form of a transaction that has been digitally signed the sender.
- Contracts can send messages to other contracts. Messages only exist in the execution environment and are never stored. Messages are similar to transactions; however, the main difference is that they are produced by the contracts, whereas transactions are produced by entities external (externally owned accounts) to the Ethereum environment.
- A message consists of the components mentioned here:
  1. Sender of the message
  2. Recipient of the message
  3. Amount of Wei to transfer and message to the contract address
  4. Optional data field (Input data for the contract)
  5. Maximum amount of gas that can be consumed Messages are generated when CALL or DELEGATECALL Opcodes are executed by the contracts.

Calls

- A call does not broadcast anything to the blockchain; instead, it is a local call to a contract function and runs locally on the node. It is almost like a local function call. It does not consume any gas as it is a read-only operation. It is akin to a dry run.
- Calls are executed locally on a node and generally do not result in any state change. As defined in the yellow paper, this is the act of passing a message from one account to another. If the destination account has an associated EVM code, then the virtual machine will start upon the receipt of the message to perform the required operations.
- If the message sender is an autonomous object, then the call passes any data returned from the virtual machine operation.

State is altered by transactions. These are created by external factors and are signed and then broadcasted to the Ethereum network

Mining

- Mining is the process by which new currency is added to the blockchain. This is an incentive for the miners to validate and verify blocks made up of transactions. The mining process helps secure the network by verifying computations.
- At a theoretical level, a miner performs the following functions:
  1. Listens for the transactions broadcasted on the Ethereum network and determines the transactions to be processed.
  2. Determines stale blocks called Uncles or Ommers and includes them in the block.
  3. Updates the account balance with the reward earned from successfully mining the block.
  4. Finally, a valid state is computed and block is finalized, which defines the result of all state transitions. The current method of mining is based on Proof of Work, which is similar to that of bitcoin.
- When a block is deemed valid, it has to satisfy not only the general consistency requirements, but it must also contain the Proof of Work for a given difficulty. The Proof of Work algorithm is due to be replaced with the Proof of Stake algorithm with the release of serenity.
- Considerable research work has been carried out in order to build the Proof of Stake algorithm suitable for the Ethereum Network. An Algorithm named Casper has been developed, which will replace the existing Proof of Work algorithm in Ethereum. This is a security deposit based on the economic protocol where nodes are required to place a security deposit before they can produce blocks. Nodes have been named bonded validators in Casper, whereas the act of placing the security deposit is named bonding.

Ethash

- Ethash is the name of the Proof of Work algorithm used in Ethereum. Originally, this was proposed as the Dagger-Hashimoto algorithm, but much has changed since the first implementation and the PoW algorithm has now evolved into what's known as Ethash now. Similar to bitcoin, the core idea behind mining is to find a nonce that once hashed the result in a predetermined difficulty level. Initially, the difficulty was low when Ethereum was new and even CPU and single GPU mining was profitable to a certain extent, but that is no longer the case. Now either pooled mining is profitable, or large GPU mining farms are used for mining purposes.

- Ethash is a memory-hard algorithm, which makes it difficult to be implemented on specialized hardware. As in bitcoin, ASICs have been developed, which have resulted in mining centralization over the years, but memory-hard Proof of Work algorithms are one way of thwarting this threat and Ethereum implements Ethash to discourage ASIC development for mining. This algorithm requires choosing subsets of a fixed resource called DAG (Directed Acyclic Graph) depending on the nonce and block headers. DAG is around 2 GB in size and changes every 30000 blocks.
- Mining can only start when DAG is completely generated the first time a mining node starts. The time between every 30000 blocks is around 5.2 days and is called epoch. This DAG is used as a seed by the Proof of Work algorithm called Ethash. According to current specifications, the epoch time is defined as 30,000 blocks. The current reward scheme is 5 Ether for successfully finding a valid nonce. In addition to receiving 5 Ethers, the successful miner also receives the cost of the gas consumed within the block and an additional reward for including stale blocks (Uncles) in the block.
- A maximum of two Uncles are allowed per block and are rewarded 7/8 of the normal block reward. In order to achieve a 12 second block time, block difficulty is adjusted at every block. The rewards are directly proportional to the miner's hash rate, which basically means how fast a miner can hash. Mining can be performed by simply joining the Ethereum network and running an appropriate client. The key requirement is that the node should be fully synced with the main network before mining can start.

Different methods of Mining

- **CPU mining** Even though not profitable on the main net, CPU mining is still valuable on the test network or even a private network to experiment with mining and contract deployment. Private and test networks will be discussed with practical examples in the next chapter. A geth example is shown on how to start CPU mining here. Geth can be started with mine switch in order to start mining.

```
geth --mine --minerthreads <n>
```

CPU mining can also be started using the web 3 geth console. Geth console can be started by issuing the following command:

```
geth attach
```

After this, the miner can be started by issuing the following command, which will return true if successful, or false otherwise. Take a look at the following command:

```
Miner.start(4)
True
```

The preceding command will start the miner with four threads. Take a look at the following command:

```
Miner.stop
True
```

GPU mining

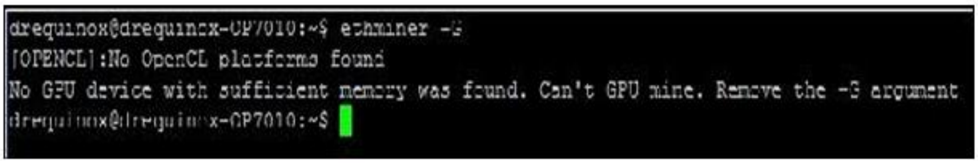
At a basic level, GPU mining can be performed easily by running two commands

```
geth --rpc
```

Once geth is up and running and the blockchain is fully downloaded, Ethminer can be run in order to start mining. Ethminer is a standalone miner that can also be used in the farm mode to contribute to mining pools.

```
ethminer -G
```

Running with G switch assumes that the appropriate graphics card is installed and configured correctly. If no appropriate graphics cards are found, ethminer will return an error, as shown in the following screenshot.



```
drequinox@drequinox-OP7010:~$ ethminer -G
[OPENCL]:No OpenCL platforms found
No GPU device with sufficient memory was found. Can't GPU mine. Remove the -G argument
drequinox@drequinox-OP7010:~$
```

Error in case no appropriate GPUs can be found

Once the graphics cards are installed and configured correctly, the process can be started by issuing the ethminer -G command. Ethminer can also be used to run benchmarking, as shown in the following screenshot. There are two modes that can be invoked for benchmarking. It can either be CPU or GPU. The commands are shown here.

CPU BENCHMARKING

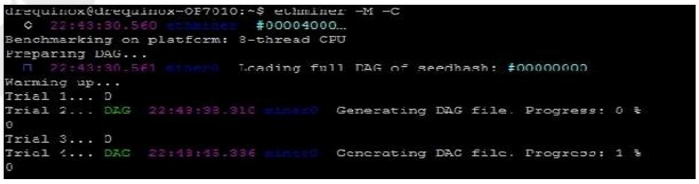
$ ethminer -M -C

GPU BENCHMARKING

$ ethminer -M -G

The GPU device to be used can also be specified in the command line: **$ ethminer -M -G --opencl-device 1**

As GPU mining is implemented using OpenCL AMD, chipset-based GPUs tend to work faster as compared to NVidia GPUs. Due to the high memory requirements (DAG creation), FPGAs and ASICs will not provide any major advantage over GPUs. This is done on purpose in order to discourage the development of specialized hardware for mining



```
drequinox@drequinox-OP7010:~$ ethminer -M -C
  ◇  22:43:30.560 ethminer  #00004000…
Benchmarking on platform: 8-thread CPU
Preparing DAG...
  ⊓  22:43:30.561 miner0  Loading full DAG of seedhash: #00000000
Warming up...
Trial 1... 0
Trial 2... DAG  22:43:38.310 miner0  Generating DAG file. Progress: 0 %
0
Trial 3... 0
Trial 4... DAG  22:43:46.336 miner0  Generating DAG file. Progress: 1 %
0
```

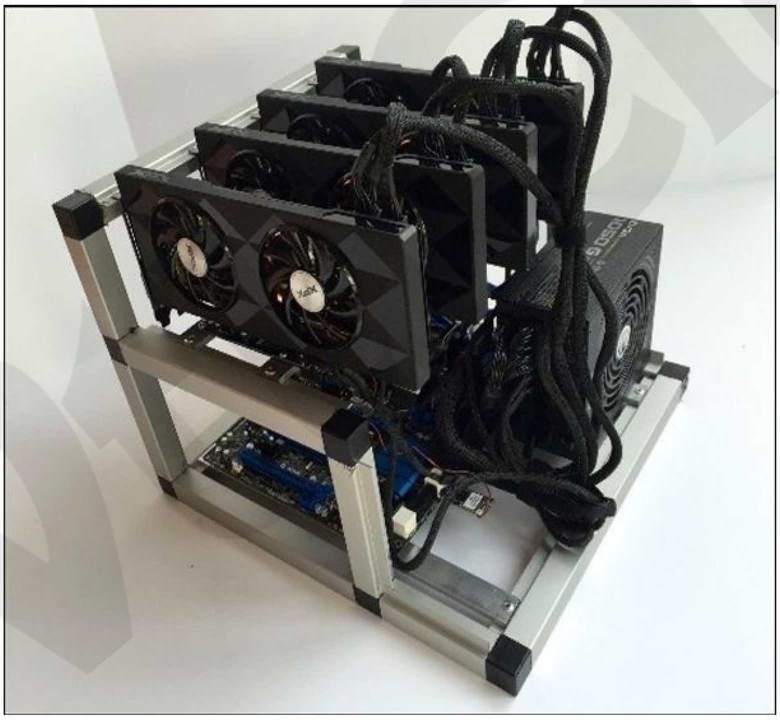
Mining rigs

- As difficulty increased over time for mining Ether, mining rigs with multiple GPUs were starting to be built by the miners. A mining rig usually contains around five GPU cards, and all of them work in parallel for mining, thus improving the chances of finding valid nonces for mining.
- Mining rigs can be built with some effort and are also available commercially from various vendors.

MOTHERBOARD A specialized motherboard with multiple PCI-E x1 or x16 slots, for example, BIOSTAR Hi-Fi or ASRock H81, is required.

SSD HARD DRIVE An SSD hard drive is required. The SSD drive is recommended because of its much faster performance over the analog equivalent. This will be mainly used to store the blockchain.

GPU The GPU is the most important component of the rig as it is the main workhorse that will be used for mining. For example, it can be a Sapphire AMD Radeon R9 380 with 4 GB RAM. Linux Ubuntu's latest version is usually chosen as the operating system for the rig. There is also another variant of Linux available, called EthOS (available at http://ethosdistro.com/), that is especially built for Ethereum mining and supports mining operations natively. Finally, mining software such as Ethminer and geth are installed. Additionally, some remote monitoring and administration software is also installed so that rigs can be monitored and managed remotely, if required. It is also important to put appropriate air conditioning or cooling mechanisms in place as running multiple GPUs can generate a lot of heat. This also necessitates the need for using an appropriate monitoring software that can alert users if there are any problems with the hardware, for example, if the GPUs are overheating



A mining rig for Ethereum for sale at eBay

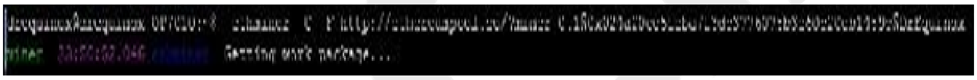
Mining pools

There are many online mining pools that offer Ethereum mining.

Ethminer can be used to connect to a mining pool using the following command. Each pool publishes its own instructions, but generally, the process of connecting to a pool is similar. An example from ethereumpool.co is shown here

```
ethminer -C -F http://ethereumpool.co/?
```

```
miner=0.1@0x024a20cc5feba7f3dc3776075b3e60
c20eb1459c@DrEquinox
```



```
drequinox@drequinox OP7010:~$ ethminer -C -F http://ethereumpool.co/?miner=0.1@0x024a20cc5feba7f3dc3776075b3e60c20eb1459c@DrEquinox
miner  23:50:02.046 ethminer  Getting work package...
```

Screenshot of ethminer

The Ethereum network

The Ethereum network is a peer-to-peer network where nodes participate in order to maintain the blockchain and contribute to the consensus mechanism. Networks can be divided into three types, based on requirements and usage.

1. MainNet: MainNet is the current live network of ethereum. The current version of MainNet is homestead.
2. TestNet: TestNet is also called Ropsten and is the test network for the Ethereum blockchain. This blockchain is used to test smart contracts and DApps before being deployed to the production live blockchain. Moreover, being a test network, it allows experimentation and research.
3. Private net(s) As the name suggests, this is the private network that can be created by generating a new genesis block. This is usually the case in

distributed ledger networks, where a private group of entities start their own blockchain and use it as a permissioned blockchain.

Supporting protocols There are various supporting protocols that are in development in order to support the complete decentralized ecosystem. This includes **whisper and Swarm** protocols. In addition to the contracts layer, which is the core blockchain layer, there are additional layers that need to be decentralized in order to achieve a complete decentralized ecosystem. This includes decentralized storage and decentralized messaging. Whisper, being developed for ethereum, is a decentralized messaging protocol, whereas Swarm is a decentralized storage protocol. Both of these technologies are being developed currently and have been envisaged to provide the basis for a fully decentralized web.

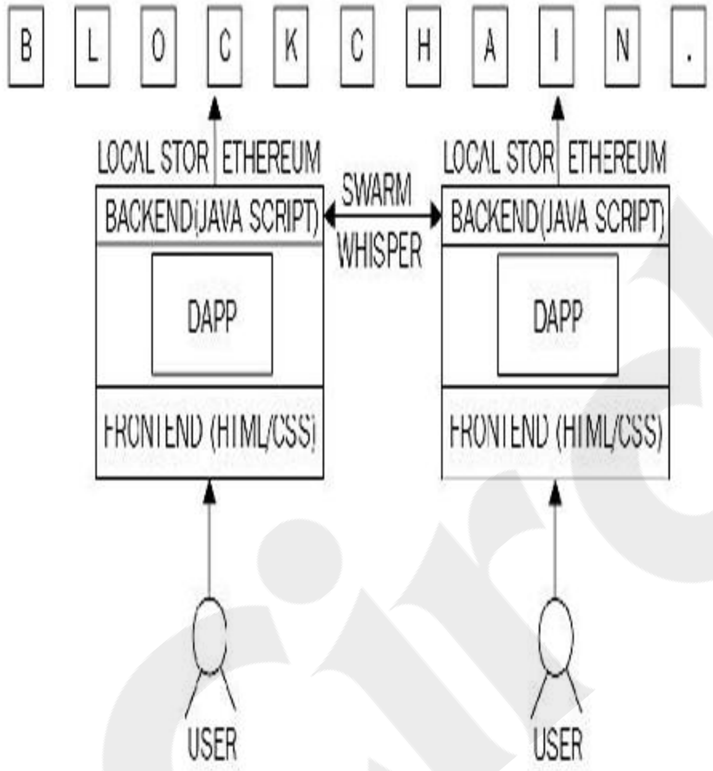
WHISPER

Whisper provides decentralized peer-to-peer messaging capabilities to the ethereum network. In essence, whisper is a communication protocol that nodes use in order to communicate with each other. The data and routing of messages are encrypted within whisper communications. Moreover, it is designed to be used for smaller data transfers and in scenarios where real-time communication is not required. Whisper is also designed to provide a communication layer that cannot be traced and provides "dark communication" between parties. Blockchain can be used for communication, but that is expensive and consensus is not really required for messages exchanged between nodes. Therefore, whisper can be used as a protocol that allows Whisper is already available with geth and can be enabled using the --shh option while running the geth ethereum client.

SWARM

Swarm is being developed as a distributed file storage platform. It is a decentralized, distributed, and peer-to peer storage network. Files in this network are addressed by the hash of their content. This is in contrast to the traditional centralized services, where storage is available at a central location only. This is developed as a native base layer service for the Ethereum web 3.0 stack. Swarm is integrated with DevP2P, which is the multiprotocol network layer of Ethereum. Swarm is envisaged to provide a DDOS (Distributed Denial of service)-resistant and fault tolerant distributed storage layer for Ethereum Web 3.0. Both whisper and Swarm are under development and, even though Proof of Concept and alpha code has been released for Swarm, there is no stable production version available yet.

The following figure gives a high level overview of how Swarm and whisper fit together and work with blockchain:



Diagrams shows blockchain, whisper and Swarm

Applications developed on Ethereum

There are various implementations of DAOs and smart contracts in Ethereum, most notably, the **DAO,** which was recently hacked and required a hard fork in order for funds to be recovered. The DAO was created to serve as a decentralized platform to collect and distribute investments**. Augur** is another DAPP that has been implemented on Ethereum, which is a decentralized prediction market. Various other decentralized applications are listed on http://dapps.ethercasts.com/.

Scalability and security issues Scalability in any blockchain is a fundamental issue. Security is also of paramount importance. Issues such as privacy and confidentiality have caused some adaptability issues, especially in the financial sector. However, a great deal of research is being conducted in these areas**.**

Hands On- CLIENTS AND WALLETS

As Ethereum is under heavy development and evolution, there are many components, clients, and tools that have been developed and introduced over the last few years. The following is a list of all main components, client software, and tools that are available with Ethereum. This list is provided in order to reduce the ambiguity around many tools and clients available for Ethereum. The list provided here also explains the usage and significance of various components.

Geth This is the Go implementation of the Ethereum client. Eth This is the C++ implementation of the Ethereum client**.**

Pyethapp This is the Python implementation of the Ethereum client.

Parity This implementation is built using Rust and developed by EthCore.

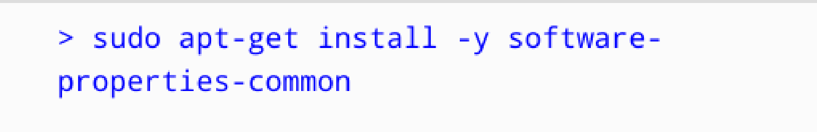
EthCore is a company that works on the development of the parity client. Parity can be downloaded from https://ethcore.io/parity.html.

Light clients SPV clients download only a small subset of the blockchain. This allows low resource devices, such as mobile phones, embedded devices, or tablets, to be able to verify the transactions. A complete ethereum blockchain and node are not required in this case and SPV clients can still validate the execution of transactions. SPV clients are also called light clients. This idea is similar to bitcoin SPV clients. There is a wallet available from Jaxx (https://jaxx.io/), which can be installed on iOS and Android, which provides the SPV (Simple Payment Verification) functionality.

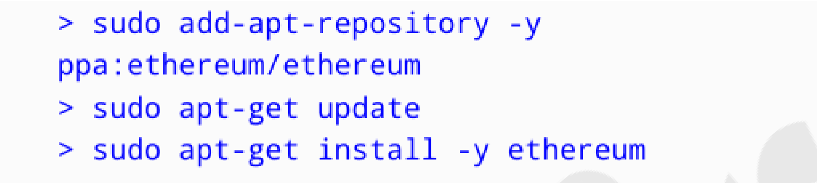
Installation

The following installation procedure describes the installation of various Ethereum clients on Ubuntu systems. Instructions for other operating systems are available on Ethereum Wikis. As Ubuntu systems will be used in examples later on, only installation on Ubuntu has been described here. Geth client can be installed by using the following command on an Ubuntu system

Geth client can be installed by using the following command on an Ubuntu system

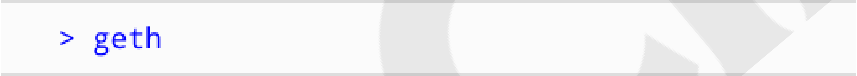


```
> sudo apt-get install -y software-
properties-common
```



```
> sudo add-apt-repository -y
ppa:ethereum/ethereum
> sudo apt-get update
> sudo apt-get install -y ethereum
```

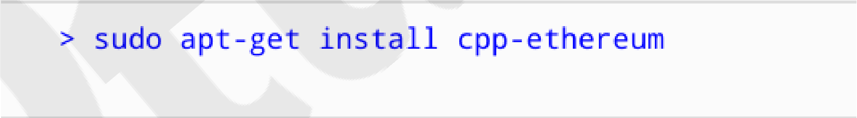
After installation is completed. Geth can be launched simply by issuing the geth command at the command prompt, as it comes preconfigured with all the required parameters to connect to the live Ethereum network (mainnet)



```
> geth
```

ETH INSTALLATION

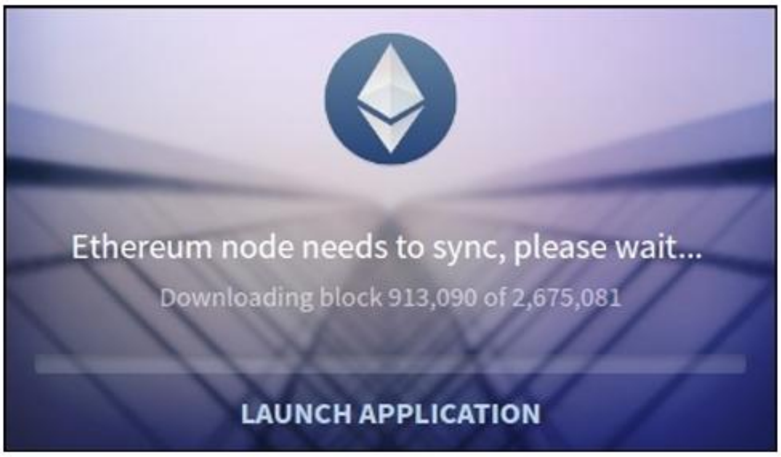
Eth is the C++ implementation of the Ethereum client and can be installed using the following command on Ubuntu.



```
> sudo apt-get install cpp-ethereum
```

MIST BROWSER

Mist browser is a user-friendly interface for end users with a feature-rich graphical user interface that is used to browse DAPPS and for account management and contract management. Mist installation is covered in the next chapter. When Mist is launched for the first time, it will initialize geth in the background and will sync with the network. It can take from a few hours to a few days depending on the speed and type of the network to fully synchronize with the network. If TestNet is used, then syncing completes relatively faster as the size of TestNet (Ropsten) is not as big as MainNet.



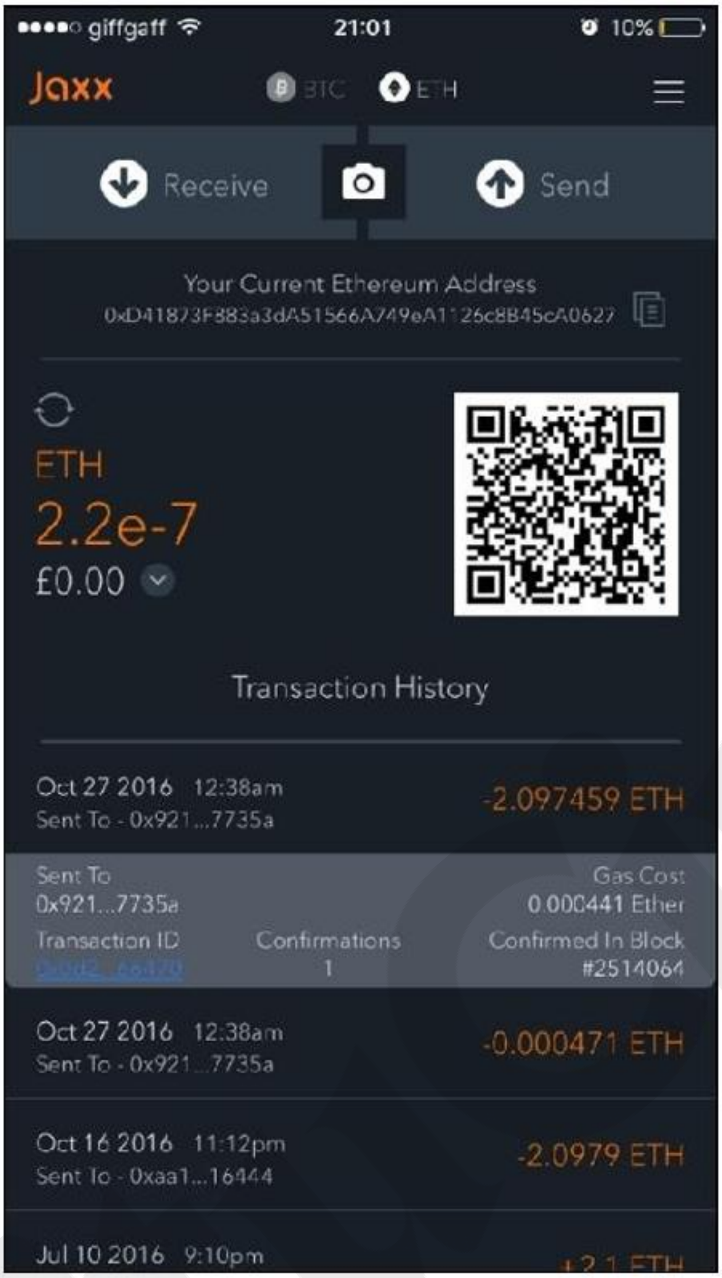
Ethereum node needs to sync, please wait...

Downloading block 913,090 of 2,675,081

Mist browser starting up and syncing with the main network Mist browser is not a wallet; in fact, it is a browser of DAPPS and provides a user-friendly user interface for the creation and management of contracts, accounts, and browsing decentralized applications. Ethereum wallet is a DAPP that is released with Mist.

Wallet is a generic program that can store private keys and associated accounts and, based on the addresses stored within it, it can compute the existing balance of Ether associated with the addresses by querying the blockchain.

Other wallets include but are not limited to MyEtherWallet, which is an open source ether wallet developed in JavaScript. MyEtherWallet runs in the client browser. This is available at https://www.myetherwallet.com. Icebox is developed by Consensys. This is a cold storage browser that provides secure storage of Ether. It depends on whether the computer on which Icebox is run is connected to the Internet or not. Various wallets are available for ethereum for desktop, mobile, and web platforms. A popular Ethereum iOS Wallet named Jaxx is shown in the following image:



Your Current Ethereum Address
0xD41873F883a3dA51566A749eA1126c8B45cA0627

ETH
2.2e-7
£0.00

Transaction History

Oct 27 2016 12:38am
Sent To - 0x921...7735a
-2.097459 ETH

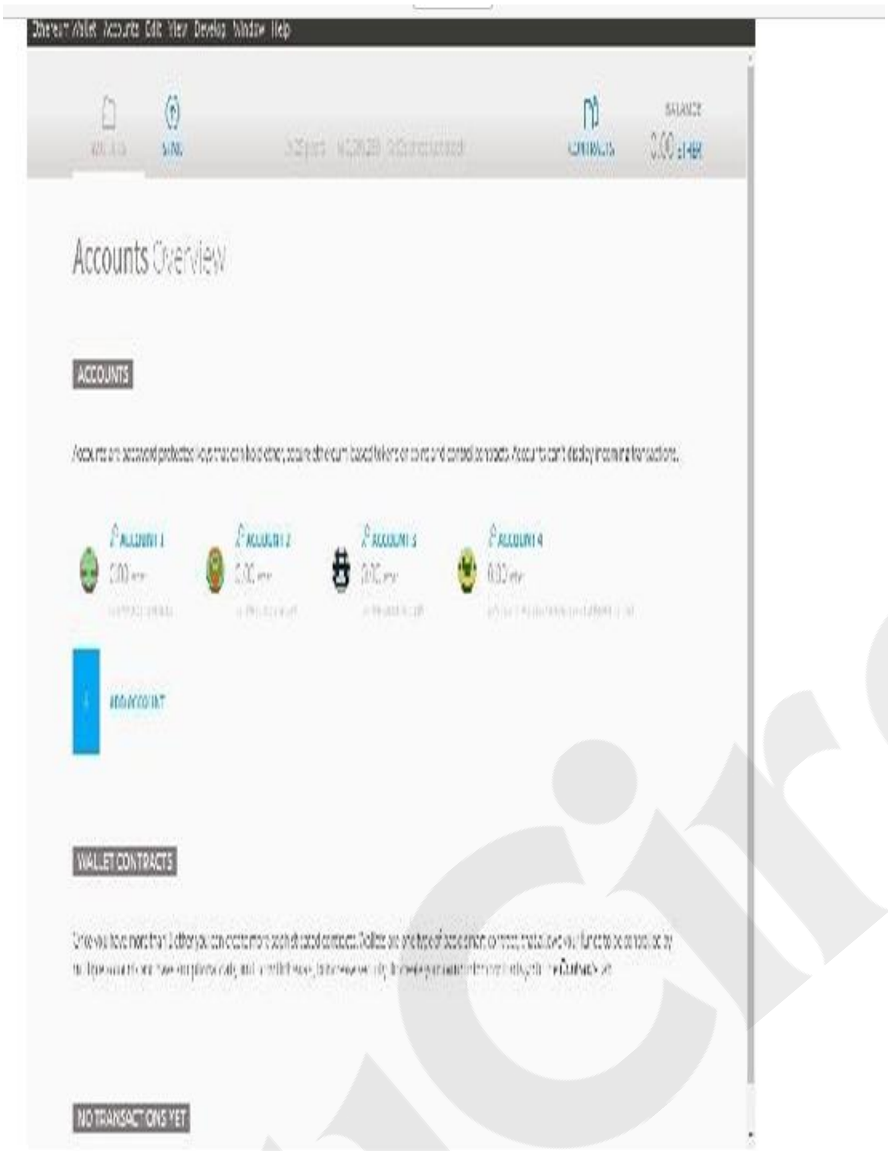
Sent To
0x921...7735a
Gas Cost
0.000441 Ether
Transaction ID
[illegible]
Confirmations
1
Confirmed In Block
#2514064

Oct 27 2016 12:38am
Sent To - 0x921...7735a
-0.000471 ETH

Oct 16 2016 11:12pm
Sent To - 0xaa1...16444
-2.0979 ETH

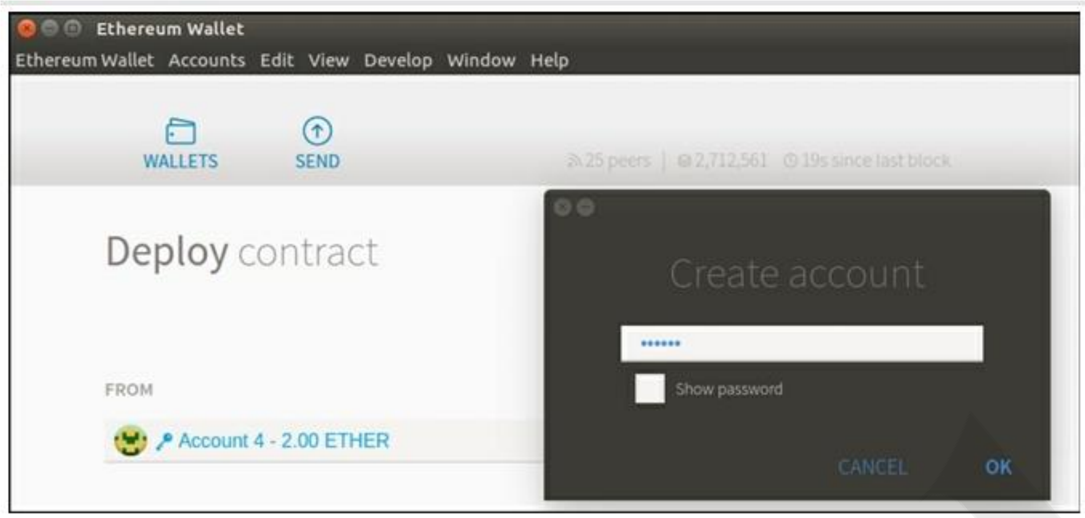
Jul 10 2016 9:10pm
+2.1 ETH

Jaxx Ethereum wallet for iOS showing transactions and current balance Once the blockchain is synchronized, Mist will launch and show the following interface. In this example, four accounts are displayed with no balance:



Mist browser

A new accounts can be created in a number of ways. In the Mist browser, it can be created by clicking on the Accounts menu and selecting the New account or by clicking on the Add account option in the Mist Accounts Overview screen.

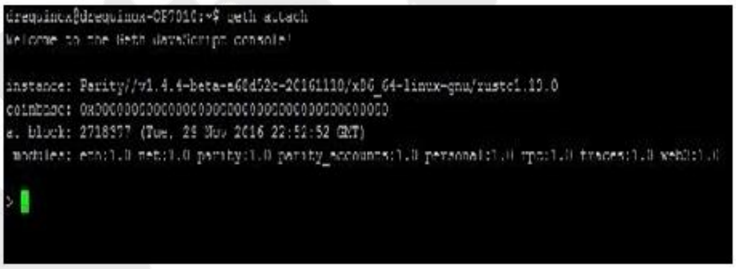


Add new account

The account will need a password to be set, as shown in the preceding figure; once the account is set up, it will be displayed in the accounts overview section of the Mist browser.

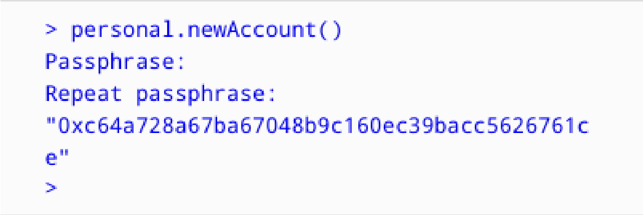
Accounts can also be added via the command line using the geth or parity command-line interface. This process is shown in the next section.

THE GETH CONSOLE The geth JavaScript console can be used to perform various functions. For example, an account can be created by attaching geth. Geth can be attached with the running daemon, as shown in the following figure



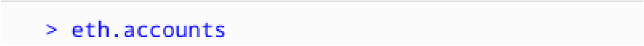
```
drequinox@drequinox-OP7010:~$ geth attach
Welcome to the Geth JavaScript console!

instance: Parity//v1.4.4-beta-a68d52c-20161110/x86_64-linux-gnu/rustc1.13.0
coinbase: 0x0000000000000000000000000000000000000000
at block: 2718377 (Tue, 29 Nov 2016 22:52:52 GMT)
 modules: eth:1.0 net:1.0 parity:1.0 parity_accounts:1.0 personal:1.0 rpc:1.0 traces:1.0 web3:1.0

>
```

Once geth is successfully attached with the running instance of the ethereum client (in this case, parity), it will display command prompt '>', which provides an interactive command line interface to interact with the ethereum client using JavaScript notations. For example, a new account can be added using the following command in the geth console:

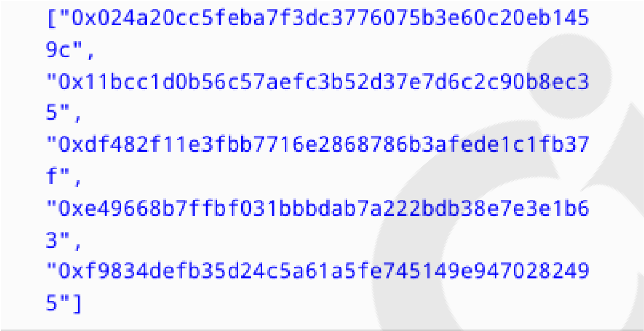


```
> personal.newAccount()
Passphrase:
Repeat passphrase:
"0xc64a728a67ba67048b9c160ec39bacc5626761c
e"
>
```

The list of accounts can also be displayed similarly



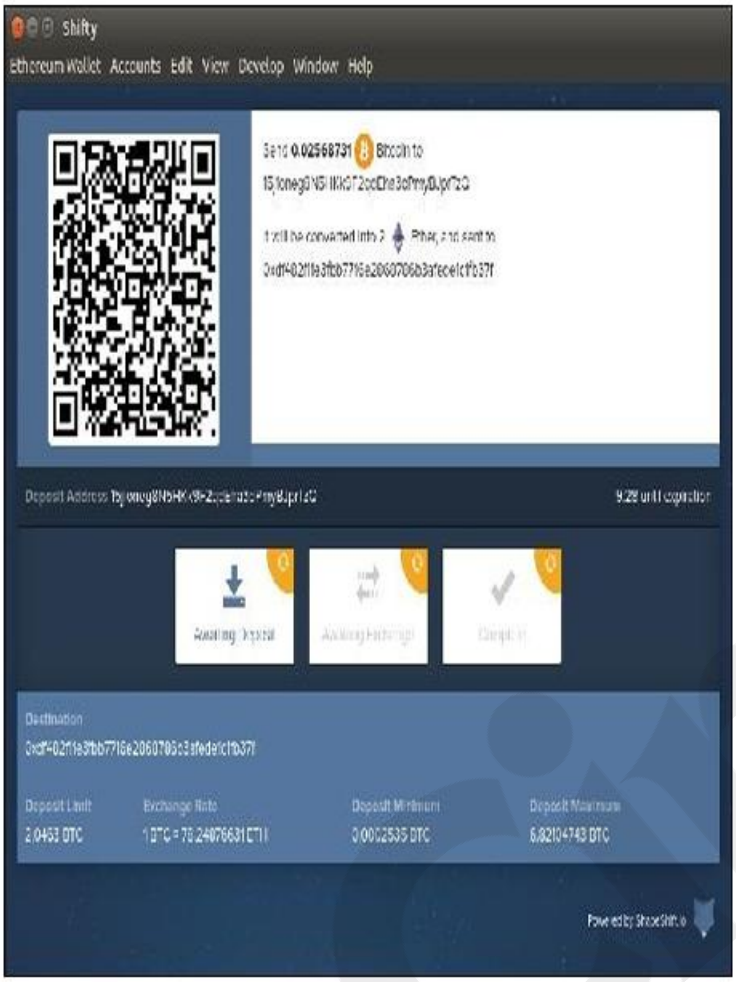
```
> eth.accounts
```



```
["0x024a20cc5feba7f3dc3776075b3e60c20eb145
9c",
"0x11bcc1d0b56c57aefc3b52d37e7d6c2c90b8ec3
5",
"0xdf482f11e3fbb7716e2868786b3afede1c1fb37
f",
"0xe49668b7ffbf031bbbdab7a222bdb38e7e3e1b6
3",
"0xf9834defb35d24c5a61a5fe745149e947028249
5"]
```

FUNDING THE ACCOUNT WITH BITCOIN

This option is available with the Mist browser by clicking on the account and then selecting the option to fund the account. The backend engine used for this operation is shapeshift.io and can be used to fund the account from bitcoin or other currencies, including the fiat currency option as well. Once the exchange is completed, the transferred Ether will be available in the account.



Send 0.02568731 Bitcoin to
15jonegGN5HKK9T2zeEhe3aPmyBJpr7zG

It will be converted into 2 Ether, and sent to
0xdf402f1fe3fbb7716e2060706b3afede1cf1b37f

Deposit Address 15jonegGN5HKK9T2zeEhe3aPmyBJpr7zG

9:28 until expiration

Destination
0xdf402f1fe3fbb7716e2060706b3afede1cf1b37f

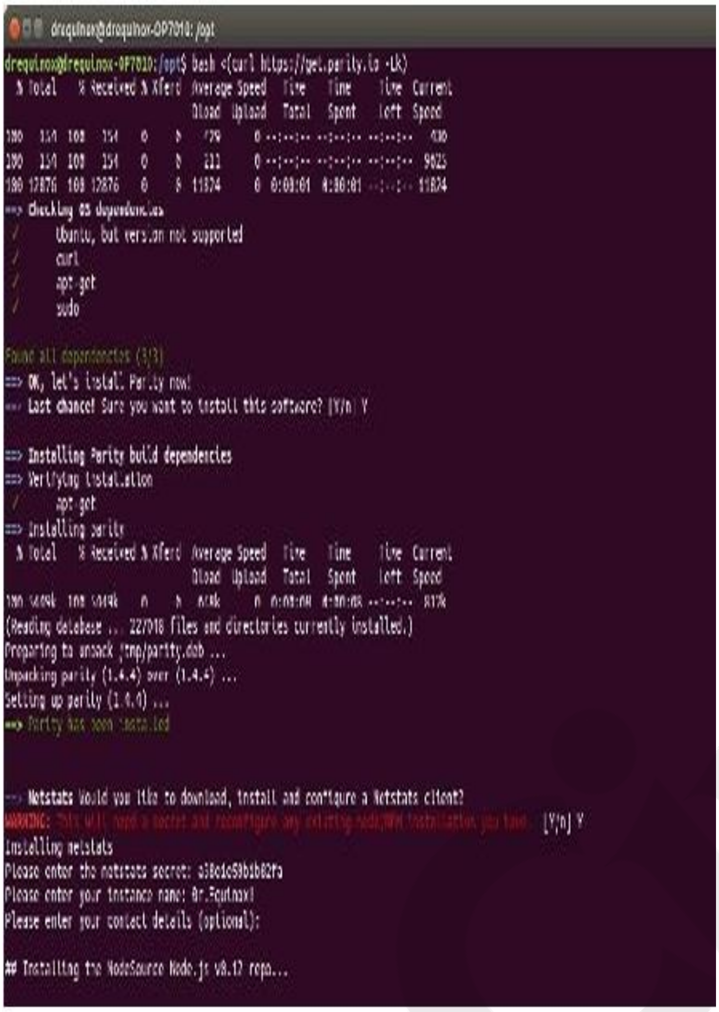
Deposit Limit	Exchange Rate	Deposit Minimum	Deposit Maximum
2.0453 BTC	1 BTC = 78.24076631 ETH	0.0002535 BTC	6.82104743 BTC

Parity installation

Parity is another implementation of the Ethereum client. It has been written using the Rust programming language. The main aim behind the development of parity is high performance, small footprint, and reliability. Parity can be installed using the following commands on an Ubuntu or Mac system

```
bash <(curl https://get.parity.io -Lk)
```

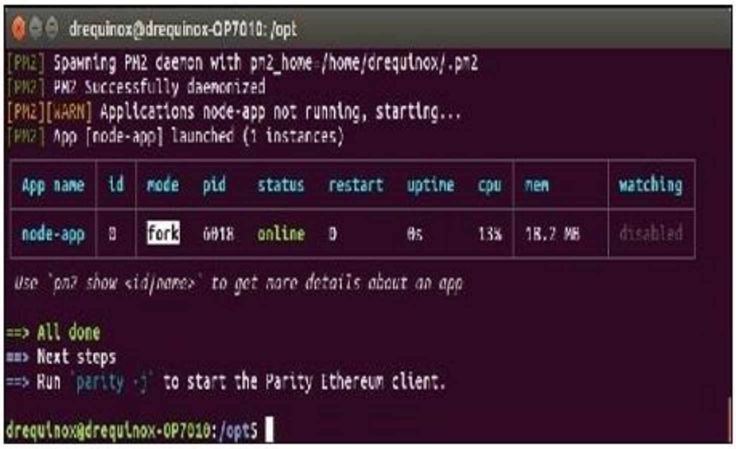
This will initiate the download and installation of the parity client. After the installation of parity is completed, the installer will also offer the installation of the netstats client. The netstat client is a daemon that runs in the background and collects important statistics and displays them on stats.ethdev.com. A sample installation of parity is shown in the following screenshot



```
drequinox@drequinox-OP7010: /opt
drequinox@drequinox-OP7010:/opt$ bash <(curl https://get.parity.io -Lk)
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   154  100   154    0     0    729      0 --:--:-- --:--:-- --:--:--   430
100   154  100   154    0     0    211      0 --:--:-- --:--:-- --:--:--  9025
100 12876  100 12876    0     0  11874      0  0:00:01  0:00:01 --:--:-- 11874
==> Checking OS dependencies
 ✓      Ubuntu, but version not supported
 ✓      curl
 ✓      apt-get
 ✓      sudo

Found all dependencies (3/3)
==> OK, let's install Parity now!
==> Last chance! Sure you want to install this software? [Y/n] Y

==> Installing Parity build dependencies
==> Verifying installation
 ✓      apt-get
==> Installing parity
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 5404k  100 5404k    0     0   613k      0  0:00:08  0:00:08 --:--:--  817k
(Reading database ... 227018 files and directories currently installed.)
Preparing to unpack /tmp/parity.deb ...
Unpacking parity (1.4.4) over (1.4.4) ...
Setting up parity (1.4.4) ...
==> Parity has been installed


==> Netstats Would you like to download, install and configure a Netstats client?
WARNING: This will need a secret and reconfigure any existing node/NPM installation you have. [Y/n] Y
Installing netstats
Please enter the netstats secret: a38e1e59b1b82fa
Please enter your instance name: Dr.Equinox!
Please enter your contact details (optional):

## Installing the NodeSource Node.js v8.12 repo...
```

Once the installation is completed successfully, the following message is displayed. Ethereum parity node can then be started using `parity -j`. If compatibility with geth is required in order to use Ethereum wallet (Mist browser) with parity, then the `parity -geth` command should be used to run parity. This will run

parity in compatibility mode with the geth client and will consequently allow Mist to run on top of parity.

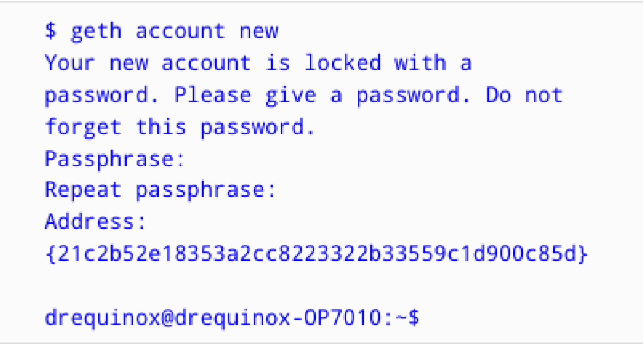


```
drequinox@drequinox-OP7010: /opt
[PM2] Spawning PM2 daemon with pm2_home=/home/drequinox/.pm2
[PM2] PM2 Successfully daemonized
[PM2][WARN] Applications node-app not running, starting...
[PM2] App [node-app] launched (1 instances)
┌──────────┬────┬──────┬──────┬────────┬─────────┬────────┬─────┬─────────┬──────────┐
│ App name │ id │ mode │ pid  │ status │ restart │ uptime │ cpu │ mem     │ watching │
├──────────┼────┼──────┼──────┼────────┼─────────┼────────┼─────┼─────────┼──────────┤
│ node-app │ 0  │ fork │ 6018 │ online │ 0       │ 0s     │ 13% │ 18.2 MB │ disabled │
└──────────┴────┴──────┴──────┴────────┴─────────┴────────┴─────┴─────────┴──────────┘
 Use `pm2 show <id|name>` to get more details about an app

==> All done
==> Next steps
==> Run `parity -j` to start the Parity Ethereum client.

drequinox@drequinox-OP7010:/opt$
```

Parity installation

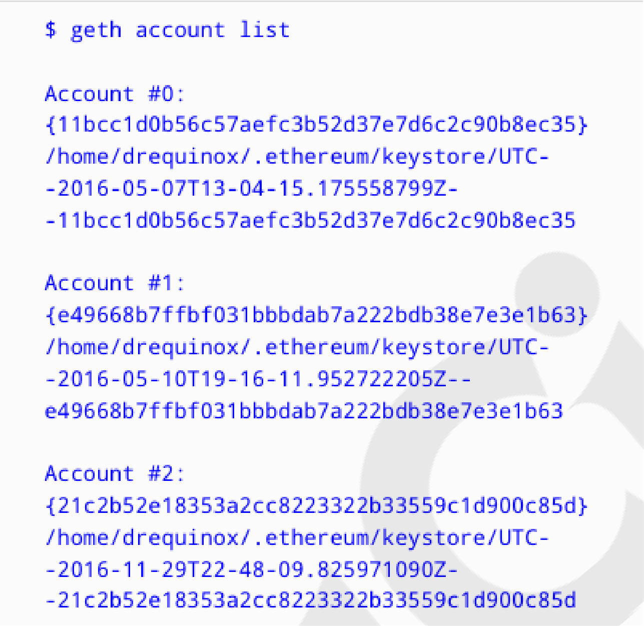
The client can optionally be listed on https://ethstats.net/.

GETH



```
$ geth account new
Your new account is locked with a
password. Please give a password. Do not
forget this password.
Passphrase:
Repeat passphrase:
Address:
{21c2b52e18353a2cc8223322b33559c1d900c85d}

drequinox@drequinox-OP7010:~$
```

The list of accounts can be shown using geth using the following command:



```
$ geth account list

Account #0:
{11bcc1d0b56c57aefc3b52d37e7d6c2c90b8ec35}
/home/drequinox/.ethereum/keystore/UTC-
-2016-05-07T13-04-15.175558799Z-
-11bcc1d0b56c57aefc3b52d37e7d6c2c90b8ec35

Account #1:
{e49668b7ffbf031bbbdab7a222bdb38e7e3e1b63}
/home/drequinox/.ethereum/keystore/UTC-
-2016-05-10T19-16-11.952722205Z--
e49668b7ffbf031bbbdab7a222bdb38e7e3e1b63

Account #2:
{21c2b52e18353a2cc8223322b33559c1d900c85d}
/home/drequinox/.ethereum/keystore/UTC-
-2016-11-29T22-48-09.825971090Z-
-21c2b52e18353a2cc8223322b33559c1d900c85d
```