11

Ethereum 101

This chapter is an introduction to the Ethereum blockchain. We will introduce the fundamentals and various theoretical concepts behind Ethereum. A discussion on the different components, protocols, and algorithms relevant to the Ethereum blockchain is also presented.

We cover different elements of Ethereum such as transactions, accounts, the world state, the **Ethereum Virtual Machine** (**EVM**), and the relevant protocols so that readers can develop strong technical foundations before exploring more advanced concepts in later chapters. The main topics we will explore in this chapter are as follows:

- The Ethereum network
- Components of the Ethereum ecosystem
- The Ethereum Virtual Machine (EVM)
- Smart contracts
- Trading and investment

Let's begin with a brief overview of the foundation, architecture, and use of the Ethereum blockchain.

Ethereum – an overview

Vitalik Buterin (`https://vitalik.ca`) conceptualized Ethereum (`https://ethereum.org`) in November, 2013. The core idea proposed was the development of a Turing-complete language that allows the development of arbitrary programs (smart contracts) for blockchain and **Decentralized Applications** (**DApps**). This concept is in contrast to Bitcoin, where the scripting language is limited and only allows necessary operations.

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 difficulty bomb is a difficulty adjustment mechanism that eventually forces all miners to stop mining on Ethereum 1 and move to Ethereum 2. Once activated, over time, it makes mining on Ethereum 1.x so prohibitively slow that mining becomes infeasible and results in a so-called "ice age". In other words, this mechanism exponentially increases the **Proof-of-Work** (**PoW**) mining difficulty to a level where block generation becomes impossible, thus forcing the miners to migrate to Ethereum 2.0's **Proof-of-Stake** (**PoS**) system.

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. This chapter covers Istanbul, which at the time of writing is the latest major upgrade of Ethereum. We will cover Serenity and relevant concepts in *Chapter 16*, *Serenity*.

A list of all releases as announced is maintained at `https://github.com/ethereum/go-ethereum/releases`.

The formal specification of Ethereum has been described in the *yellow paper*, which can be used to develop Ethereum implementations.

The yellow paper

The Ethereum yellow paper (`https://ethereum.github.io/yellowpaper/paper.pdf`) was written by *Dr. Gavin Wood*, the founder of Ethereum and Parity (`http://gavwood.com`), and serves as a formal specification of the Ethereum protocol. Anyone can implement an Ethereum client by following the protocol specifications defined in the paper.

While this paper can be somewhat challenging to read, especially for those who do not have a background in algebra or mathematics and are not familiar with mathematical notation, it contains a complete formal specification of Ethereum. This specification can be used to implement a fully compliant Ethereum client. Therefore, it appears necessary to understand this paper at least at a high level.

The list of all symbols with their meanings used in the paper is provided here with the anticipation that it will make reading the Ethereum yellow paper more accessible.

Useful mathematical symbols

The following table shows the mathematical symbols used in the yellow paper, along with their meaning:

Symbol	Meaning	Symbol	Meaning
≡	Is defined as	≤	Less than or equal to
=	Is equal to	σ	Sigma, the world state
≠	Is not equal to	μ	Mu, the machine state
‖…‖	Length of	Υ	Upsilon, Ethereum state transition function
∈	Is an element of	∏	Block-level state transition function
∉	Is not an element of	.	Sequence concatenation
∀	For all	∃	There exists
∪	Union	∧	Contract creation function
∧	Logical AND	Δ	Increment
:	Such that	⌊...⌋	Floor, lowest element
{}	Set	⌈...⌉	Ceiling, highest element
()	Function of tuple	|...|	Number of bytes
[]	Array indexing	⊕	Exclusive OR
∨	Logical OR	(a ,b)	Real numbers >= a and < b
>	Is greater than	∅	Empty set, null

+	Addition		
-	Subtraction		
Σ	Summation		
{	Describing various cases of if, otherwise		

Now, in the next and upcoming sections, we will introduce the Ethereum blockchain and its core elements.

The Ethereum blockchain

Ethereum, just like any other blockchain, can be visualized as a transaction-based state machine. This definition is mentioned in the Ethereum yellow paper written by Dr. Gavin Wood.

The core idea is that in the Ethereum blockchain, 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:

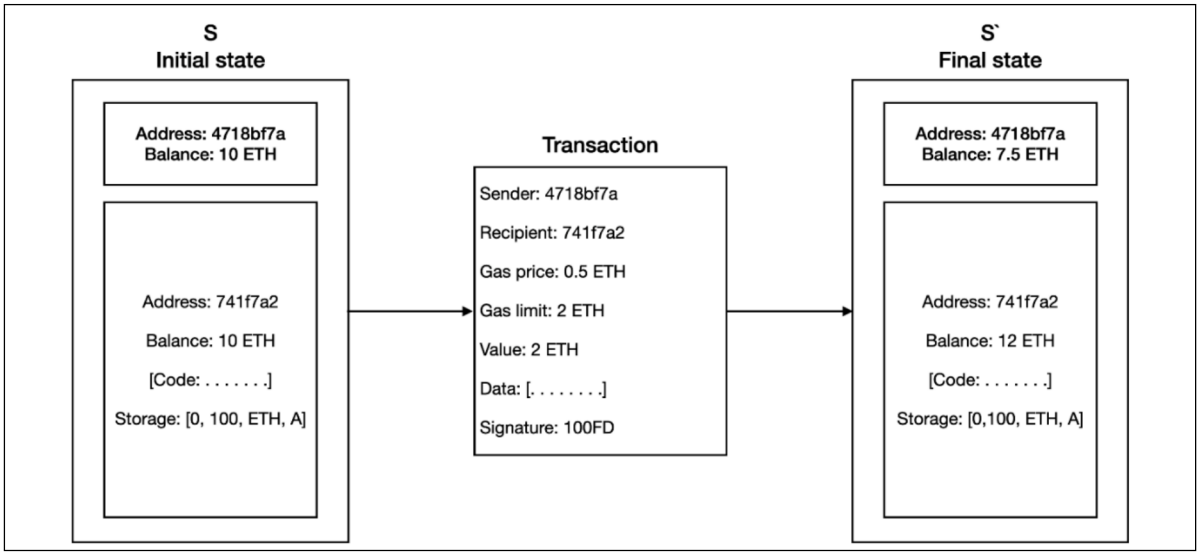


Figure 11.1: Ethereum state transition function

In the preceding example, a transfer of two 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. Mining plays a central role in state transition, and we will elaborate the mining process in detail in the later sections. The state is stored on the Ethereum network as the *world state*. This is the global state of the Ethereum blockchain.

More on this will be presented later in the *Nodes and miners* section in *Chapter 12, Further Ethereum*, in the context of state storage.

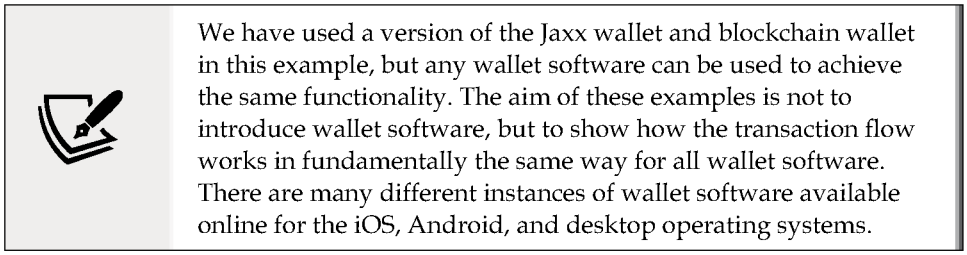
Ethereum – a user's perspective

In this section, we will see how Ethereum works from a user's point of view. For this purpose, we will present the most common use case of transferring funds – in our use case, from one user (Bashir) to another (Irshad). We will use two Ethereum clients, one for sending funds and the other for receiving. There are several steps involved in this process, as follows:

1. First, either a user requests money by sending the request to the sender, or the sender decides to send money to the receiver. We'll be using the Jaxx Ethereum wallet software on iOS – you can download the Jaxx wallet from `https://jaxx.io`. The request can be sent by sending the receiver's Ethereum address to the sender. For example, there are two users, Bashir and Irshad. If Irshad requests money from Bashir, then she can send a request to Bashir by using a QR code. Once Bashir receives this request he will either scan the QR code or manually type in Irshad's Ethereum address and send the ether to Irshad's address. This request is encoded as a QR code, shown in the following screenshot, that can be shared via email, text, or any other communication method:

Figure 11.2: QR code as shown in the blockchain wallet application

2. Once Bashir receives this request he will either scan this QR code or copy the Ethereum address in the Ethereum wallet software and initiate a transaction. This process is shown in the following screenshot, where Jaxx is used to send money to Irshad. The following screenshot shows that the sender has entered both the amount and the destination address to send the ether to receiver. Just before sending the ether, the final step is to confirm the transaction, which is also shown here:



We have used a version of the Jaxx wallet and blockchain wallet in this example, but any wallet software can be used to achieve the same functionality. The aim of these examples is not to introduce wallet software, but to show how the transaction flow works in fundamentally the same way for all wallet software. There are many different instances of wallet software available online for the iOS, Android, and desktop operating systems.

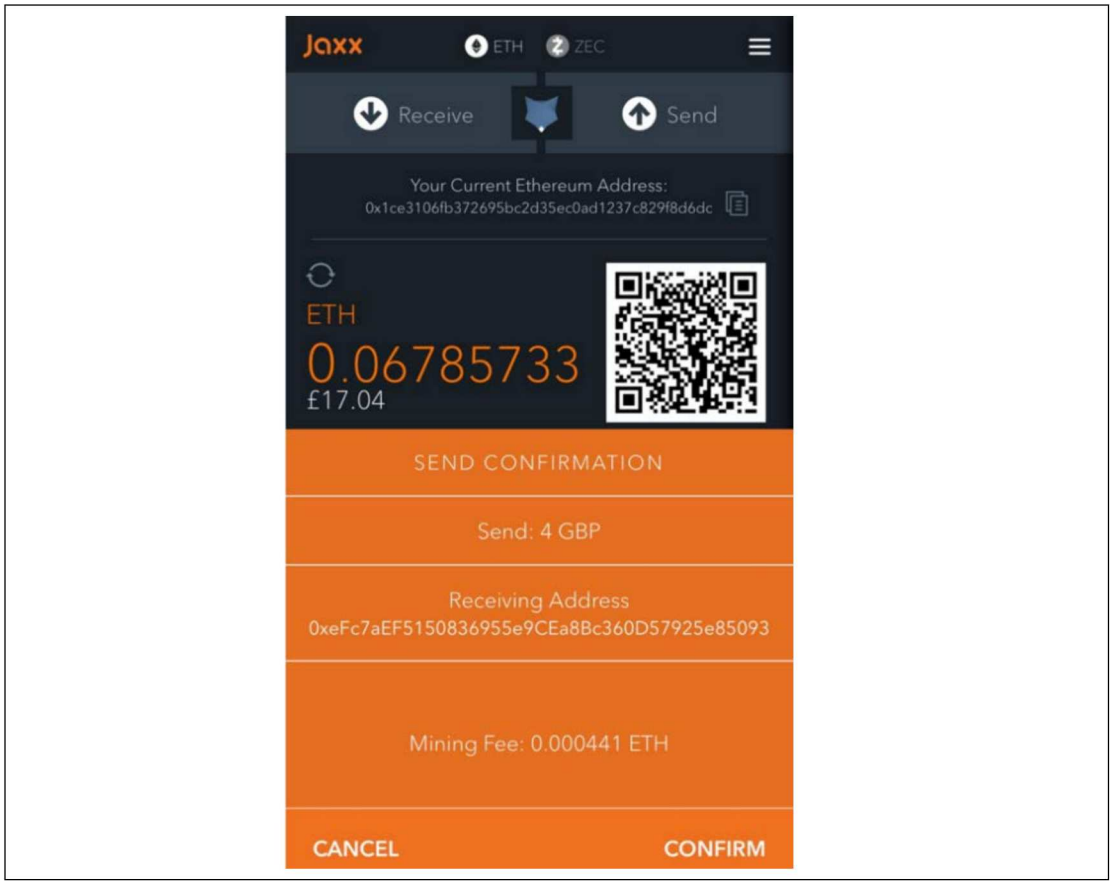


Figure 11.3: Confirmation of funds sent in the Jaxx wallet from Bashir

3. Once the request (transaction) for money to be sent is constructed in the wallet software, it is then broadcasted to the Ethereum network. The transaction is digitally signed by the sender as proof that he is the owner of the ether.
4. This transaction is then picked up by nodes called miners on the Ethereum network for verification and inclusion in the block. At this stage, the transaction is still unconfirmed.
5. Once it is verified and included in the block, the PoW process starts. We will explain this process in more detail later in *Chapter 12, Further Ethereum*.
6. Once a miner finds the answer to the PoW problem by repeatedly hashing the block with a new nonce, this block is immediately broadcasted to the rest of the nodes, which then verifies the block and PoW.
7. If all the checks pass then this block is added to the blockchain, and miners are paid rewards accordingly.
8. Finally, Irshad gets the ether, and it is shown in her wallet software. This is shown in the following screenshot:

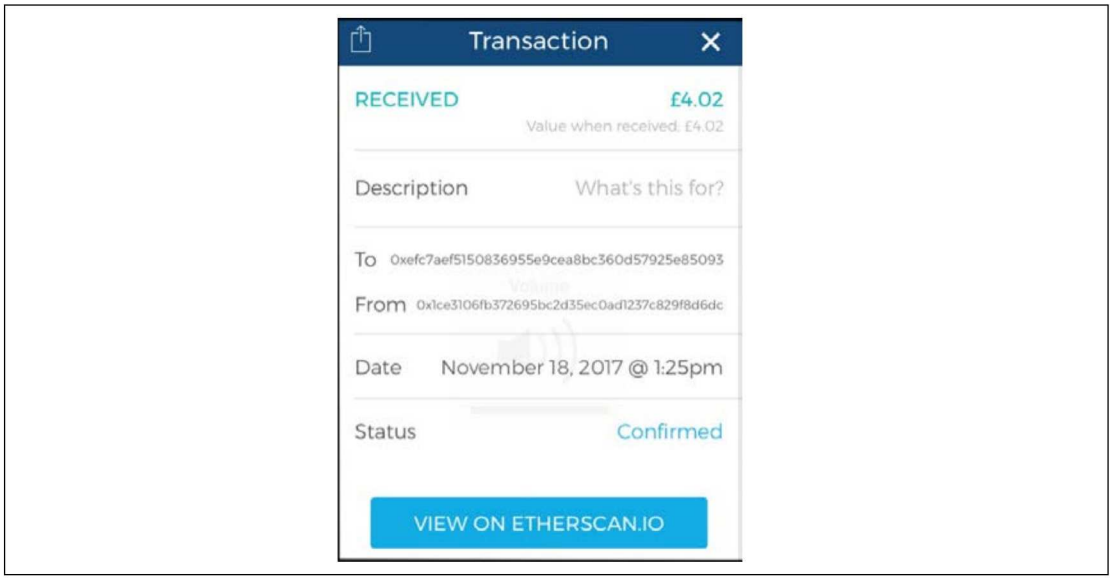


Figure 11.4: The transaction received in Irshad's blockchain wallet

On the blockchain, this transaction is identified by the following transaction hash: `0xc63dce6747e1640abd63ee63027c3352aed8cdb92b6a02ae25225666e171009e`

The details of this transaction can be visualized on the block explorer at `https://etherscan.io/`, as shown in the following screenshot:

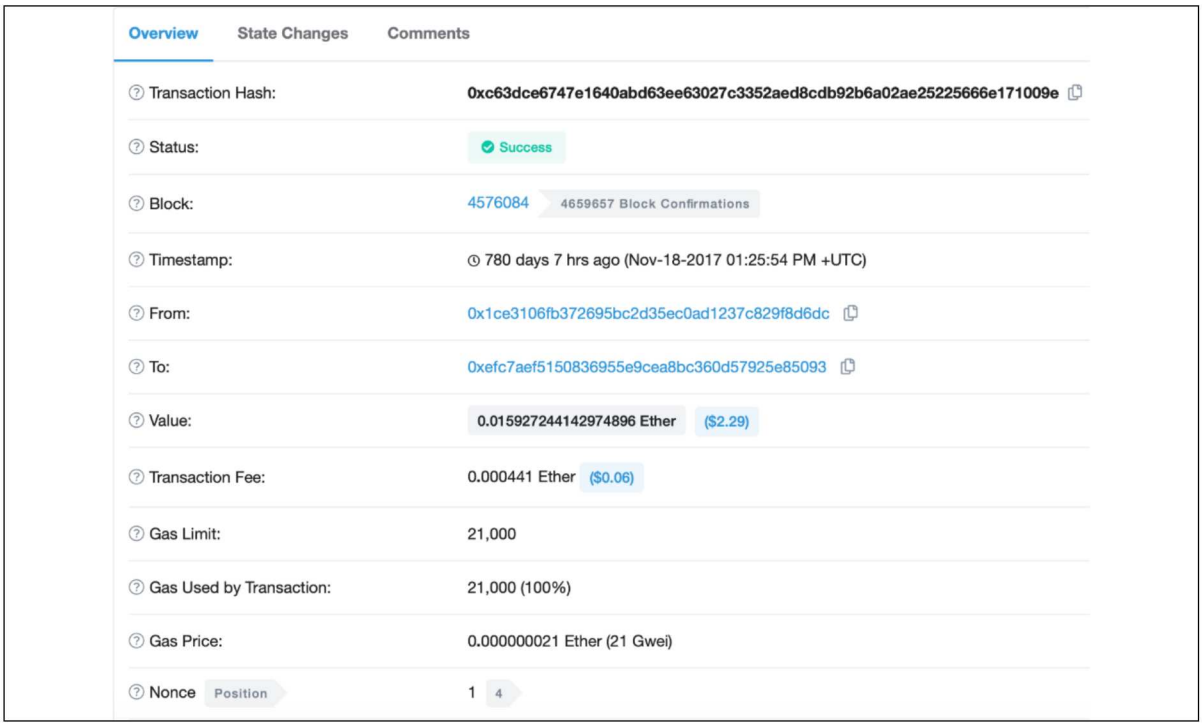


Figure 11.5: Etherscan Ethereum blockchain block explorer

Note the transaction hash (`TxHash`) at the top. Later in the next chapter, and we will use this hash to see how this transaction is constructed, processed, and stored in the blockchain. This hash is the unique ID of the transaction that can be used to trace this transaction throughout the blockchain network.

With this example, we complete our discussion on the most common usage of the Ethereum network: transferring ether from a user to another. This case was just a quick overview of the transaction process in order to introduce the concept. More in-depth technical details will be explained in the upcoming sections of this chapter where we discuss various components of the Ethereum ecosystem.

In the next section, we will see what components make up the Ethereum ecosystem. Once we have understood all the theoretical concepts, we will see in detail how the aforementioned transaction travels through the Ethereum blockchain to reach the beneficiary's address. At this point, we will be able to correlate the technical concepts with the preceding transfer transaction example.

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 the requirements and usage. These types are described in the following subsections.

The mainnet

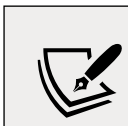
The **mainnet** is the current live network of Ethereum. Its network ID is 1 and its chain ID is also 1. The network and chain IDs are used to identify the network. A block explorer that shows detailed information about blocks and other relevant metrics is available at `https://etherscan.io`. This can be used to explore the Ethereum blockchain.

Testnets

There is a number of testnets available for Ethereum testing. The aim of these test blockchains is to provide a testing environment for smart contracts and DApps before being deployed to the production live blockchain. Moreover, being test networks, they also allow experimentation and research. The main testnet is called *Ropsten*, which contains all the features of other smaller and special-purpose testnets that were created for specific releases. For example, other testnets include *Kovan* and *Rinkeby*, which were developed for testing Byzantium releases. The changes that were implemented on these smaller testnets have also been implemented in Ropsten. Now the Ropsten test network contains all properties of Kovan and Rinkeby.

Private nets

As the name suggests, these are private networks that can be created by generating a new genesis block. This is usually the case in private blockchain networks, where a private group of entities start their blockchain network and use it as a permissioned or consortium blockchain.



A table detailing some major Ethereum networks, including their network IDs and other relevant details, can be found in this book's online resource page here: `https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf`

Network IDs and chain IDs are used by Ethereum clients to identify the network. Chain ID was introduced in EIP155 as part of replay protection mechanism. EIP155 is detailed at `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md`.

A comprehensive list of Ethereum networks is maintained and available at `https://chainid.network`.

More discussion on how to connect to a testnet and how to set up private nets will be had in *Chapter 13*, *Ethereum Development Environment*.

Components of the Ethereum ecosystem

The Ethereum blockchain stack consists of various components. At the core, there is the Ethereum blockchain running on the peer-to-peer 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 the `geth` client via the **Remote Procedure Call** (**RPC**) interface.

The overall Ethereum ecosystem architecture is visualized in the following diagram:

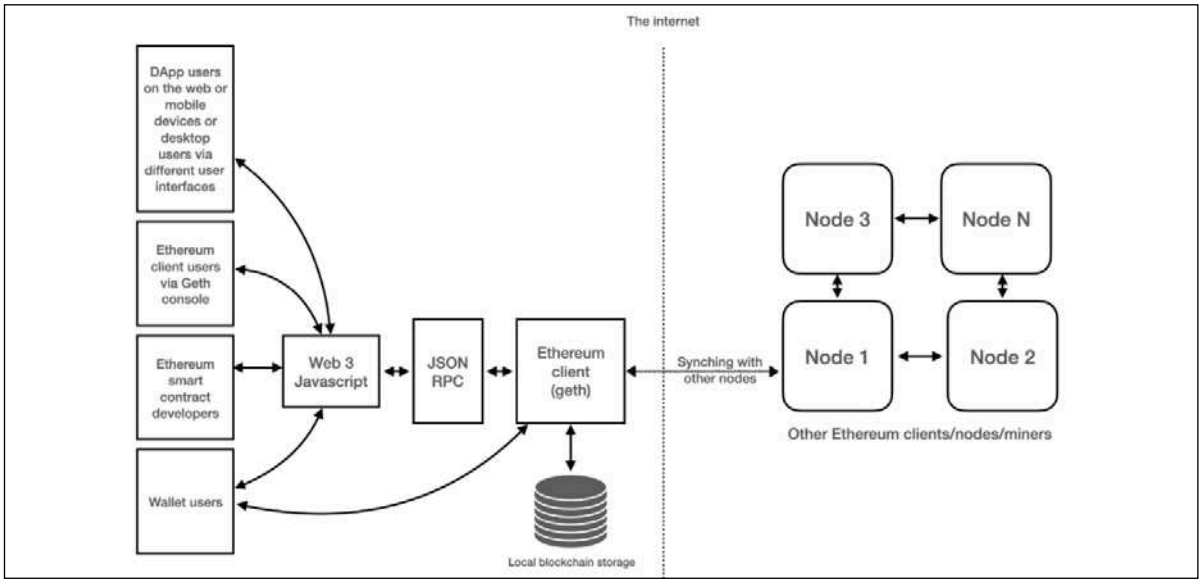


Figure 11.6: Ethereum high-level ecosystem

A list of elements present in the Ethereum blockchain is presented here:

- Keys and addresses
- Accounts
- Transactions and messages
- Ether cryptocurrency/tokens
- The EVM
- Smart contracts and native contracts

In the following sections, we will discuss each of these one by one. Other components such wallets, clients, development tools, and environments will be discussed in the upcoming chapters.

We will also discuss relevant technical concepts related to a high-level element within that section. First, let's have a look at keys and addresses.

Keys and addresses

Keys and addresses are used in the Ethereum blockchain to represent ownership and transfer ether. The keys used are made up of pairs of private and public parts. The private key is generated randomly and is kept secret, whereas a public key is derived from the private key. Addresses are derived from public keys and are 20-byte codes used to identify accounts.

The process of key generation and address derivation is as follows:

1. First, a private key is randomly chosen (a 256-bit positive integer) under the rules defined by the elliptic curve `secp256k1` specification (in the range [`1`, `secp256k1n - 1`]).
2. The public key is then derived from this private key using the **Elliptic Curve Digital Signature Algorithm** (**ECDSA**) recovery function. We will discuss this in the following *Transactions and messages* section, in the context of digital signatures.
3. An address is derived from the public key, specifically, from the rightmost 160 bits of the Keccak hash of the public key.

An example of how keys and addresses look in Ethereum is shown as follows:

- A private key:

  `b51928c22782e97cca95c490eb958b06fab7a70b9512c38c36974f47b954ffc4`

- A public key:

  ```
  3aa5b8eefd12bdc2d26f1ae348e5f383480877bda6f9e1a47f6a4afb35cf998ab847
  f1e3948b1173622dafc6b4ac198c97b18fe1d79f90c9093ab2ff9ad99260
  ```

- An address:

  `0x77b4b5699827c5c49f73bd16fd5ce3d828c36f32`

The preceding private key is shown here only as an example, and should not be reused.

Another key element in Ethereum is an account, which is required in order to interact with the blockchain. It either represents a user or a smart contract.

Accounts

Accounts are one of the main building blocks of the Ethereum blockchain. They are defined by pairs of private and public keys. Accounts are used by users to interact with the blockchain via transactions. A transaction is digitally signed by an account before submitting it to the network via a node. Ethereum, being a *transaction-driven state machine*, the state is created or updated as a result of the interaction between accounts and transaction executions. All accounts have a state that, when combined together, represents the state of the Ethereum network. With every new block, the state of the Ethereum network is updated. Operations performed between and on the accounts represent state transitions. The 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. The transaction fee is calculated, and the sending address is resolved using the signature. Furthermore, the sender's account balance is checked and subtracted accordingly, and the nonce is incremented. An error is returned if the account balance is insufficient.
3. Provide enough ETH (the gas price) to cover the cost of the transaction. We will cover gas and relevant concepts shortly in this chapter. This is charged per byte and is incrementally proportional to the size of the transaction. In this step, the actual transfer of value occurs. The flow is from the sender's account to the 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.
4. In cases of transaction failure due to insufficient account balance or gas, all state changes are rolled back except for the fee payment, which is paid to the miners.
5. Finally, the remainder (if any) of the fee is sent back to the sender as change and the fee is paid to the miners accordingly. At this point, the function returns the resulting state, which is also stored on the blockchain.

Now as we understand accounts in Ethereum generally, let's examine the different types of accounts in Ethereum.

Types of accounts

Two kinds of accounts exist in Ethereum:

- **Externally Owned Accounts** (**EOAs**)
- **Contract Accounts** (**CAs**)

The first type is EOAs, and the other is CAs. EOAs are similar to accounts that are controlled by a private key in Bitcoin. CAs are the accounts that have code associated with them along with the private key.

The various properties of each type of account are as follows:

EOAs

- They have a state.
- They are associated with a human user, hence are also called user accounts.
- EOAs have an ether balance.
- They are capable of sending transactions.
- They have no associated code.
- They are controlled by private keys.
- EOAs cannot initiate a call message.
- Accounts contain a key-value store.
- EOAs can initiate transaction messages.

CAs

- They have a state.
- They are not intrinsically associated with any user or actor on the blockchain.
- CAs have an ether balance.
- They have associated code that is kept in memory/storage on the blockchain. They have access to storage.
- They can get triggered and execute code in response to a transaction or a message from other contracts. It is worth noting that due to the Turing-completeness property of the Ethereum blockchain, the code within CAs can be of any level of complexity. The code is executed by the EVM by each mining node on the Ethereum network. The EVM is discussed later in the chapter in the *The Ethereum Virtual Machine (EVM)* section.
- Also, CAs can maintain their permanent states and can call other contracts. It is envisaged that in the Serenity (Ethereum 2.0) release, the distinction between EOAs and CAs may be eliminated.
- CAs cannot start transaction messages.
- CAs can initiate a call message.
- CAs contain a key-value store.
- CAs' addresses are generated when they are deployed. This address of the contract is used to identify its location on the blockchain.

Accounts allow interaction with the blockchain via transactions. We will now explain what an Ethereum transaction is and consider its different types.

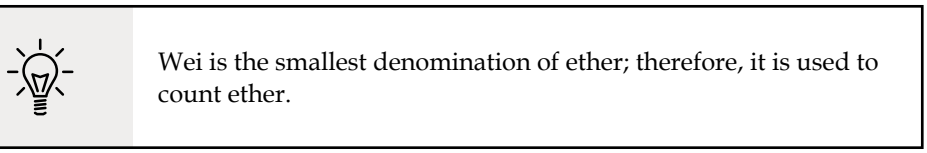
Transactions and messages

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:

- **Message call transactions**: This transaction simply produces a message call that is used to pass messages from one CA to another.
- **Contract creation transactions**: As the name suggests, these transactions result in the creation of a new CA. This means that when this transaction is executed successfully, it creates an account with the associated code.

Both of these transactions are composed of some standard fields, which are described as follows:

- **Nonce**: The 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. This is used for replay protection on the network.
- **Gas price**: The **gas price** field represents the amount of Wei required to execute the transaction. In other words, this is the amount of Wei you are willing to pay for this transaction. This is charged per unit of gas for all computation costs incurred as a result of the execution of this transaction.



Wei is the smallest denomination of ether; therefore, it is used to count ether.

- **Gas limit**: The **gas limit** field contains the value that represents the maximum amount of gas that can be consumed to execute the transaction. The concept of gas and gas limits will be covered later in the chapter in more detail. For now, it is sufficient to say that this is the fee amount, in ether, that a user (for example, the sender of the transaction) is willing to pay for computation.
- **To**: As the name suggests, the **To** field is a value that represents the address of the recipient of the transaction. This is a 20 byte value.
- **Value**: Value represents the total number of Wei to be transferred to the recipient; in the case of a CA, this represents the balance that the contract will hold.
- **Signature**: The 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, the sender of the transaction can also be determined from these values. The signature is based on the ECDSA scheme and makes use of the `secp256k1` curve. The theory of **Elliptic Curve Cryptography** (**ECC**) was discussed in *Chapter 4*, *Public Key Cryptography*. In this section, ECDSA will be presented in the context of its usage in Ethereum.

  V is a single byte value that depicts the size and sign of the elliptic curve point and can be either 27 or 28. *V* is used in the ECDSA recovery contract as a recovery ID. This value is used to recover (derive) the public key from the private key. In `secp256k1`, the recovery ID is expected to be either 0 or 1. In Ethereum, this is offset by 27. More details on the `ECDSARECOVER` function will be provided later in this chapter.

R is derived from a calculated point on the curve. First, a random number is picked, which is multiplied by the generator of the curve to calculate a point on the curve. The *x*-coordinate part of this point is *R*. *R* is encoded as a 32-byte sequence. *R* must be greater than 0 and less than the `secp256k1n` limit (`115792089237316195423570985008687907852837564279074904382605163141518161494337`).

S is calculated by multiplying *R* with the private key and adding it into the hash of the message to be signed, and then finally dividing it by the random number chosen to calculate *R*. *S* is also a 32-byte sequence. *R* and *S* together represent the signature.

To sign a transaction, the `ECDSASIGN` function is used, which takes the message to be signed and the private key as an input and produces *V*, a single-byte value; *R*, a 32-byte value; and *S*, another 32-byte value. The equation is as follows:

$$ECDSASIGN\ (Message,\ Private\ Key) = (V, R, S)$$

- **Init**: The **Init** field is used only in transactions that are intended to create contracts, that is, contract creation transactions. 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, it (init) gets destroyed immediately after that. Init also returns another code section called the *body*, which persists and runs in response to message calls that the CA 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**, and represents the input data of the message call. It is also unlimited in size and is organized as a byte array.

This structure is 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** (**MPT**)) composed of the transactions to be included. Finally, the root node of the 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:

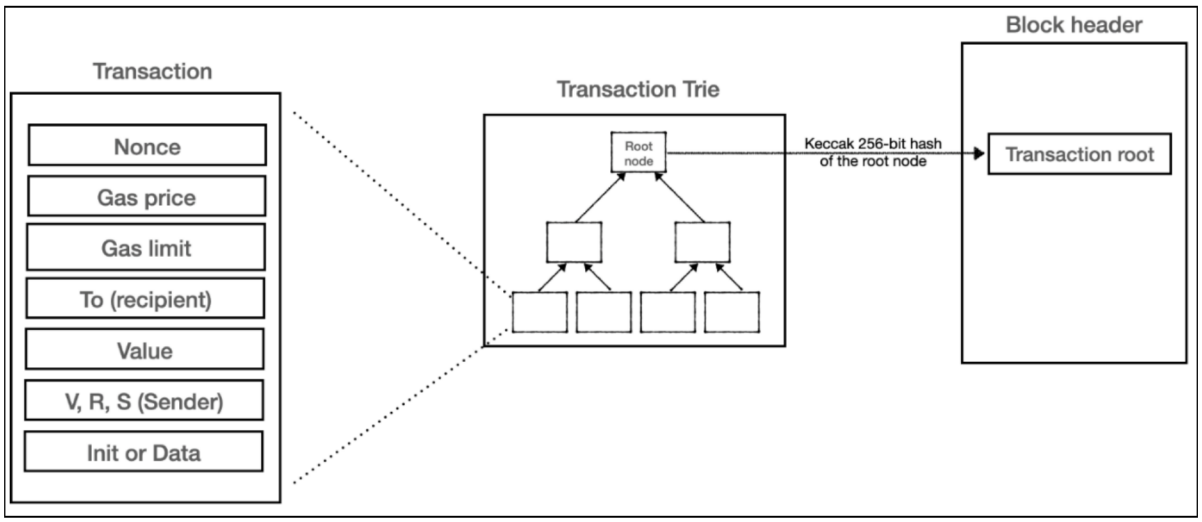


Figure 11.7: The relationship between the transaction, transaction trie, and block header

A block is a data structure that contains batches of transactions. Transactions can be found in either transaction pools or blocks. In transaction pools, they wait for verification by a node, and in blocks, they are added after successful verification. 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, such that once hashed with the block, it 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 chapters, *Chapter 6*, *Introducing Bitcoin* and *Chapter 7*, *Bitcoin Network and Payments*. Ethereum's PoW algorithm is called Ethash, and it was originally intended to be ASIC-resistant where finding a nonce requires large amounts of memory. However, now some ASICs are available for *Ethash* too. Consequently, there is also an agreement among Ethereum core developers to implement *ProgPow*, a new, more ASIC-resistant PoW consensus mechanism. We will explain ASICs and relevant concepts in *Chapter 12*, *Further Ethereum*, where we discuss mining in detail. We will also discuss block data structure in greater detail in this chapter.

The question that arises here is how all these accounts, transactions, and related messages flow through the Ethereum network and how are they stored. We saw earlier, in the *Ethereum – a user's perspective* section, a transaction flow example of how funds can be sent over the network from one user to another.

We need to know how this data looks on the network. So, before we move on to the different types of transactions and messages in Ethereum, let's explore how Ethereum data is encoded for storage and transmission. For this purpose, a new encoding scheme called **Recursive Length Prefix** (**RLP**) was developed, which we will cover here in detail.

RLP

To define RLP, we first need to understand the concept of serialization. Serialization is simply a mechanism commonly used in computing to encode data structures into a format (a sequence of bytes) that is suitable for storage and/or transmission over the communication links in a network. Once a receiver receives the serialized data, it de-serializes it to obtain the original data structure. Serialization and deserialization are also referred as marshaling and un-marshaling, respectively. Some commonly used serialization formats include XML, JSON, YAML, protocol buffers, and XDR. There are two types of serialization formats, namely *text* and *binary*. In a blockchain, there is a need to serialize and deserialize different types of data such as blocks, accounts, and transactions to support transmission over the network and storage on clients.

Why do we need a new encoding scheme when there are so many different serialization formats already available? The answer to this question is that RLP is a deterministic scheme, whereas other schemes may produce different results for the same input, which is absolutely unacceptable on a blockchain. Even a small change will lead to a totally different hash and will result in data integrity problems that will render the entire blockchain useless.

RLP is an encoding scheme developed by Ethereum developers. It 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 on storage media. It is a deterministic and consistent binary encoding scheme used to serialize objects on the Ethereum blockchain, such as account state, transactions, messages, and blocks. It operates on strings and lists to produce raw bytes that are suitable for storage and transmission. RLP is a minimalistic and simple-to-implement serialization format that does not define any data types and simply stores structures as nested arrays. In other words, RLP does not encode specific data types; instead, its primary purpose is to encode structures.

More information on RLP is available on the Ethereum wiki, at `https://eth.wiki/en/fundamentals/rlp`.

Now, having defined RLP, we can delve deeper into transactions and other relevant elements of the Ethereum blockchain. We will start by exploring different types of transactions on the Ethereum blockchain.

Contract creation transactions

A contract creation transaction is used to create smart contracts on the blockchain. There are a few essential parameters required for a contract creation transaction. These parameters are listed as follows:

- The sender
- The transaction originator
- Available gas
- Gas price
- Endowment, which is the amount of ether allocated
- A byte array of an arbitrary length
- Initialization EVM code
- The current depth of the message call/contract-creation stack (current depth means the number of items that are already present in the stack)

Addresses generated as a result of a contract creation transaction are 160 bits 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. The storage is also set to empty. The code hash is a Keccak 256-bit hash of the empty string.

The new account is initialized when the EVM code (the initialization EVM code, mentioned earlier) is executed. In the case of any exception during code execution, such as not having enough gas (running **Out of Gas**, or **OOG**), the state does not change. If the execution is successful, then the account is created after the payment of appropriate gas costs.

Since **Ethereum Homestead**, the result of a contract creation 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 versions prior to Homestead, where the contract would be created regardless of the contract code deployment being successful or not due to an OOG exception.

Message call transactions

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

- The sender
- The transaction originator
- The recipient
- The account whose code is to be executed (usually the same as the recipient)
- Available gas
- The value
- The gas price
- An arbitrary-length byte array
- The input data of the call
- The current depth of the message call/contract creation stack

Message calls result in a state transition. Message calls also produce output data, which is not used if transactions are executed. In cases where message calls are triggered by EVM code, the output produced by the transaction execution is used. As defined in the yellow paper, a message call is the act of passing a message from one account to another. If the destination account has an associated EVM code, then the EVM will start upon the receipt of the message to perform the required operations. If the message sender is an autonomous object (external actor), then the call passes back any data returned from the EVM operation.

The state is altered by transactions. These transactions are created by external factors (users) and are signed and then broadcasted to the Ethereum network.

Messages are passed between accounts using message calls. A description of messages and message calls is presented next.

Messages

Messages, as defined in the yellow paper, are the data and values that are passed between two accounts. A **message** is a data packet passed between two accounts. This data packet contains data and a value (the amount of ether). It can either be sent via a smart contract (autonomous object) or from an external actor (an **Externally Owned Account**, or **EOA**) in the form of a transaction that has been digitally signed by 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 to the Ethereum environment (EOAs).

A message consists of the following components:

- The sender of the message
- The recipient of the message
- Amount of Wei to transfer and the message to be sent to the contract address
- An optional data field (input data for the contract)
- The maximum amount of gas (`startgas`) that can be consumed

Messages are generated when the `CALL` or `DELEGATECALL` opcodes are executed by the contract running in the EVM.

In the following diagram, the segregation between the two types of transactions (**contract creation** and **message calls**) is shown:

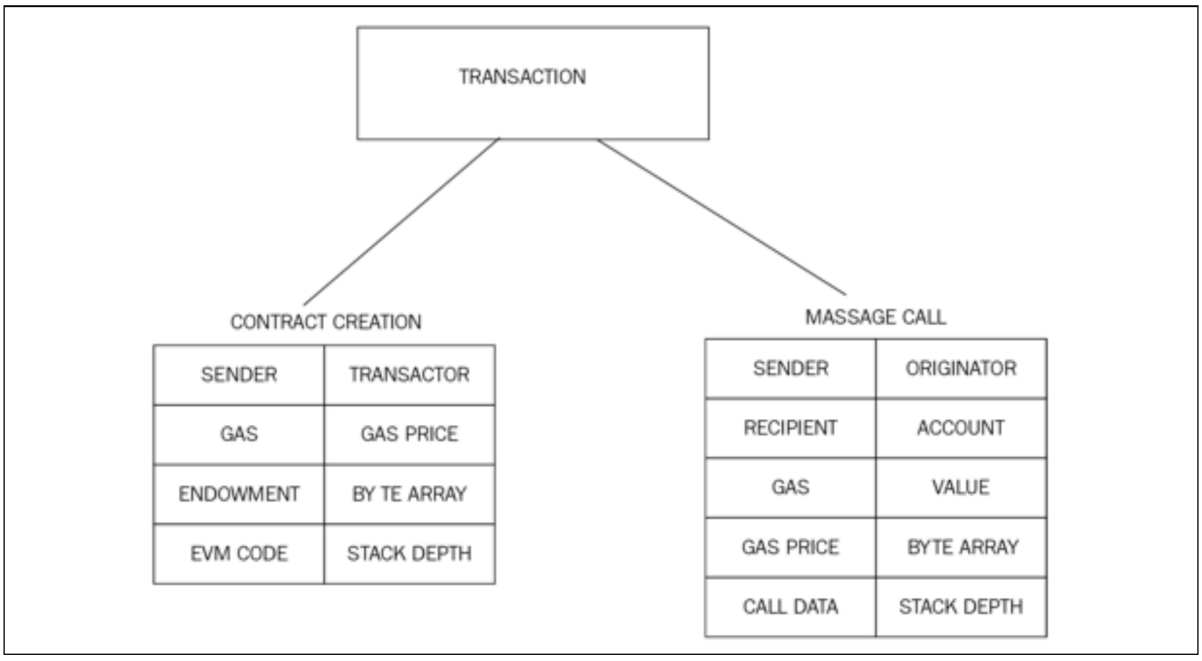


Figure 11.8: Types of transactions and the required parameters for execution

Each of these transactions has fields, which are shown with each type.

Calls

A call does not broadcast anything to the blockchain; instead, it is a local call and executes locally on the Ethereum 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 or a simulated run. Calls also do not allow ether transfer to CAs. Calls are executed locally on a node EVM and do not result in any state change because they are never mined. Calls are processed synchronously and they usually return the result immediately.

Do not confuse a *call* with a *message call transaction*, which in fact results in a state change. A call basically runs message call transactions locally on the client and never costs gas nor results in a state change. It is available in the `web3.js` JavaScript API and can be seen as almost a simulated mode of the message call transaction. On the other hand, a *message call transaction* is a `write` operation and is used for invoking functions in a CA (Contract Account, or smart contract), which does cost gas and results in a state change.

Transaction validation and execution

Transactions are executed after their validity has been verified. The initial checks 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 must be valid
- The transaction nonce must be equal to the sender's account's current nonce
- The gas limit must not be less than the gas used by the transaction
- The sender's account must contain sufficient balance to cover the execution cost

The transaction substate

A transaction substate is created during the execution of the transaction and is processed immediately after the execution completes. This transaction substate is a tuple that is composed of four items. These items are as follows:

- **Suicide set or self-destruct set**: This element contains the list of accounts (if any) that are disposed of after the transaction executes.
- **Log series**: This is an indexed series of checkpoints that allows 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. Events will be covered with practical examples in *Chapter 15*, *Introducing Web3*.
- **Refund balance**: This is the total price of gas for the transaction that initiated the execution. Refunds are not immediately executed; instead, they are used to offset the total execution cost partially.
- **Touched accounts**: Touched accounts can be defined as those accounts which are involved any potential state changing operation. Empty accounts from this set are deleted at the end of the transaction. The final state is reached after deleting accounts in the self-destruct set and emptying accounts from the touched accounts set.

State storage in the Ethereum blockchain

At a fundamental level, the Ethereum blockchain is a transaction- and consensus-driven state machine. The state needs to be stored permanently in the blockchain. For this purpose, the world state, transactions, and transaction receipts are stored on the blockchain in blocks. We discuss these components next.

The world state

This is a *mapping* between Ethereum addresses and account states. The addresses are 20 bytes (160 bits) long. This mapping is a data structure that is serialized using RLP.

The account state

The account state consists of four fields: nonce, balance, storage root, and code hash, and is described in detail here:

- **Nonce**: This is a value that is incremented every time a transaction is sent from the address. In the case of CAs, it represents the number of contracts created by the account.
- **Balance**: This value represents the number of weis, which is the smallest unit of the currency (ether) in Ethereum, held by the given address.
- **Storage root**: This field represents the root node of an MPT that encodes the storage contents of the account.
- **Code hash**: 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 the accounts trie, accounts, and block header are visualized in the following diagram. It shows the **account state**, or data structure, 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.

The accounts trie is an MPT 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.

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 the **state root** hash:

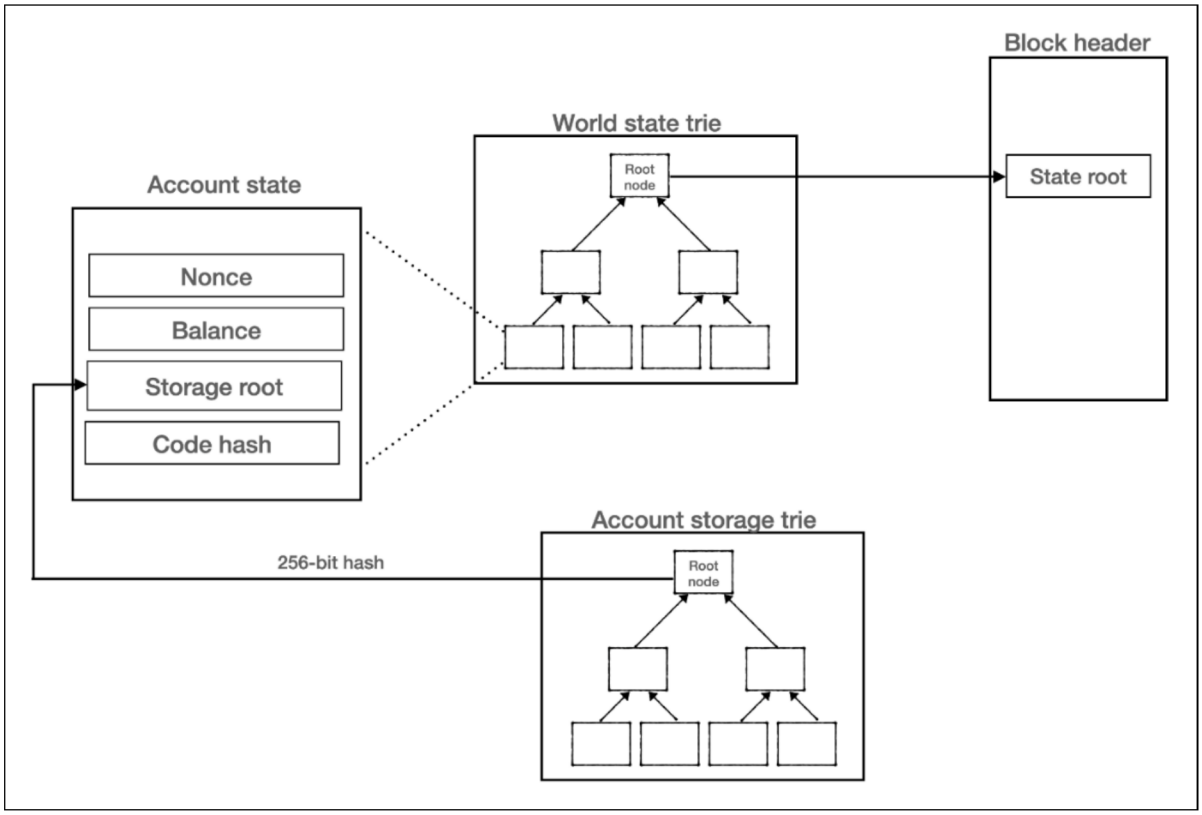


Figure 11.9: The accounts trie (the storage contents of account), account tuple, world state trie, and state root hash and their relationship

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. The hash (a 256-bit Keccak hash) of the root of this trie is placed in the block header as the receipt's root. It is composed of four elements as follows:

- **The post-transaction state**: This item is a trie structure that holds the state after the transaction has been executed. It is encoded as a byte array.
- **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.
- **Set of logs**: This field shows the set of log entries created as a result of the transaction execution. Log entries contain the logger's address, a series of log topics, and the log data.

- **The bloom filter**: A bloom filter is created from the information contained in the set of logs discussed earlier. 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. A log entry is composed of the logger's address, log topics, and log data. Log topics are encoded as a series of 32-byte data structures. The log data is made up of a few bytes of data.

Since the Byzantium release, an additional field returning the success (`1`) or failure (`0`) of the transaction is also available.

More information about this change is available at `https://github.com/ethereum/EIPs/pull/658`.

This process of transaction receipt generation is visualized in the following diagram:

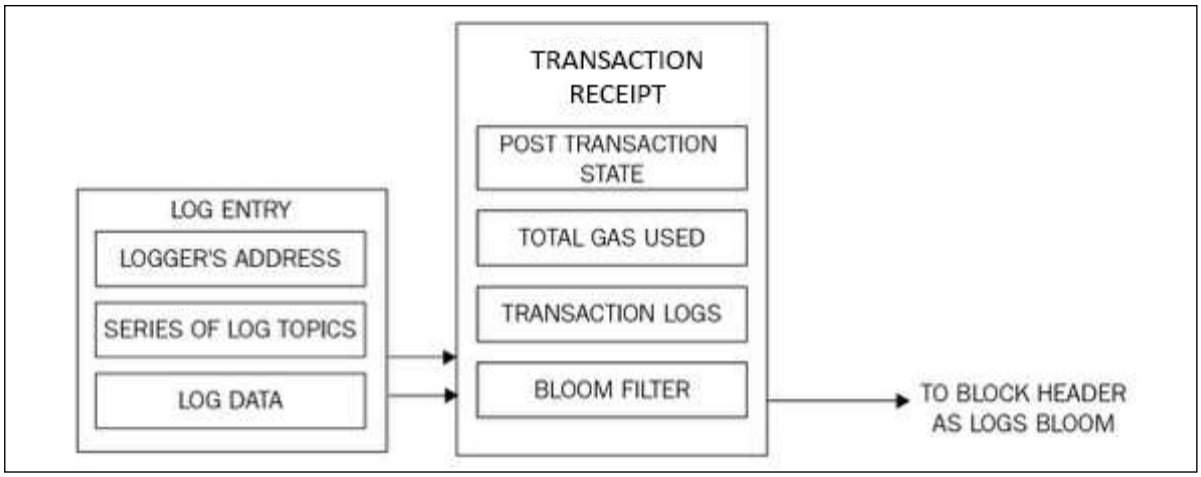


Figure 11.10: Transaction receipts and logs bloom

As a result of the transaction execution process, the state morphs from an initial state to a target state. This concept was discussed briefly at the beginning of the chapter. This state needs to be stored and made available globally in the blockchain. We will see how this process works in the next section.

Ether cryptocurrency/tokens (ETC and ETH)

As an incentive to the miners, Ethereum rewards its own native currency called **ether** (abbreviated as **ETH**). After the **Decentralized Autonomous Organization** (**DAO**) hack described in *Chapter 10*, *Smart Contracts*, a hard fork was proposed in order to mitigate the issue; therefore, there are now two Ethereum blockchains: one is called Ethereum Classic, and its currency is represented by ETC, whereas the hard-forked version is ETH, which continues to grow and on which active development is being carried out. ETC, however, has its own following with a dedicated community that is further developing ETC, which is the unforked original version of Ethereum.

This chapter is focused on ETH, which is currently the most active and official Ethereum blockchain.

Ether is minted by miners as a currency reward for the computational effort they spend to secure the network by verifying 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 to perform computation on the Ethereum blockchain.

The denomination table is as follows:

Unit	Alternative name	Wei value	Number of weis
Wei	Wei	1 Wei	1
KWei	Babbage	1e3 Wei	1,000
MWei	Lovelace	1e6 Wei	1,000,000
GWei	Shannon	1e9 Wei	1,000,000,000
microether	Szabo	1e12 Wei	1,000,000,000,000
milliether	Finney	1e15 Wei	1,000,000,000,000,000
ether	ether	1e18 Wei	1,000,000,000,000,000,000

Fees are charged for each computation performed by the EVM on the blockchain. A detailed fee schedule is described in *Chapter 12*, *Further Ethereum*.

The Ethereum Virtual Machine (EVM)

The EVM is a simple stack-based execution machine that runs bytecode instructions to transform the system state from one state to another. The word size of the EVM is set to 256-bit. The stack size is limited to 1,024 elements and is based on the **Last In, First Out** (**LIFO**) queue. The 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. The EVM also supports exception handling should exceptions occur, such as not having enough gas or providing invalid instructions, in which case the machine would immediately halt and return the error to the executing agent.

The EVM is an entirely 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. This results in increased security, deterministic execution, and allows untrusted code (code that can be run by anyone) to be executed on Ethereum blockchain.

As discussed earlier, the EVM is a stack-based architecture. The EVM is big-endian by design, and it uses 256-bit-wide words. This word size allows for Keccak 256-bit hash and ECC computations.

There are three main types of storage available for contracts and the EVM:

- **Memory**: The first type is called memory or volatile memory, which is a word-addressed byte array. When a contract finishes its code execution, the memory is cleared. It is akin to the concept of RAM. `write` operations to the memory can be of 8 or 256 bits, whereas `read` operations are limited to 256-bit words. Memory is unlimited but constrained by gas fee requirements.
- **Storage**: The other type is called storage, which is a key-value store and is permanently persisted on the blockchain. Keys and values are each 256 bits wide. It is allocated to all accounts on the blockchain. As a security measure, storage is only accessible by its own respective CAs. It can be thought of as hard disk storage.
- **Stack**: EVM is a stack-based machine, and thus performs all computations in a data area called the stack. All in-memory values are also stored in the stack. It has a maximum depth of 1024 elements and supports the word size of 256 bits.

The storage associated with the EVM is a word-addressable word array that is non-volatile and is maintained as part of the system state. Keys and values are 32 bytes in size and storage. The program code is stored in **virtual read-only memory** (**virtual ROM**) that is accessible using the `CODECOPY` instruction. The `CODECOPY` instruction copies the program code into the main memory. Initially, all storage and memory are 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 the main memory using the `CODECOPY` instruction. 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:

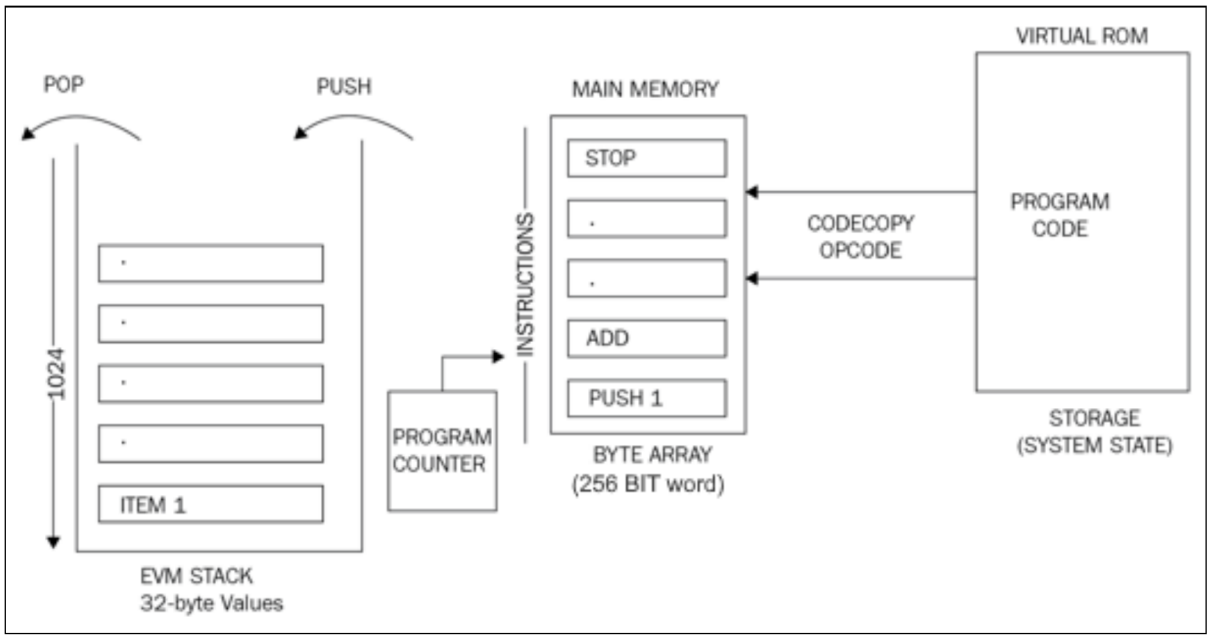


Figure 11.11: EVM operation

The preceding diagram shows an EVM stack on the left side showing that elements are pushed and popped from the stack. It also shows that a program counter is maintained, and is incremented with instructions being read from the main memory. The main memory gets the program code from the virtual ROM/storage via the `CODECOPY` instruction.

EVM optimization is an active area of research, and recent research has suggested that the EVM can be optimized and tuned to a very fine degree to achieve high performance. Research and development on **Ethereum WebAssembly** (**ewasm**)—an Ethereum-flavored iteration of WebAssembly—is already underway. **WebAssembly** (**Wasm**) was developed by Google, Mozilla, and Microsoft, and is now being designed as an open standard by the W3C community group. Wasm aims to be able to run machine code in the browser that will result in execution at native speed. More information and GitHub repository of Ethereum-flavored Wasm is available at `https://github.com/ewasm`.

Another intermediate language called YUL, which can compile to various backends such as the EVM and ewasm, is under development. More information on this language can be found at `https://solidity.readthedocs.io/en/latest/yul.html`.

Execution environment

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

- The system state.
- The remaining gas for execution.
- The address of the account that owns the executing code.
- The address of the sender of the transaction. This is the originating address of this execution (it can be different from the sender).
- The gas price of the transaction that initiated the execution.
- 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.
- 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 is the address of the account.
- The value or transaction value. This is the amount in Wei. If the execution agent is a transaction, then it is the transaction value.
- The code to be executed presented as a byte array that the iterator function picks up in each execution cycle.
- The block header of the current block.
- The number of message calls or contract creation transactions (CALLs, CREATEs or CREATE2s) currently in execution.
- Permission to make modifications to the state.

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

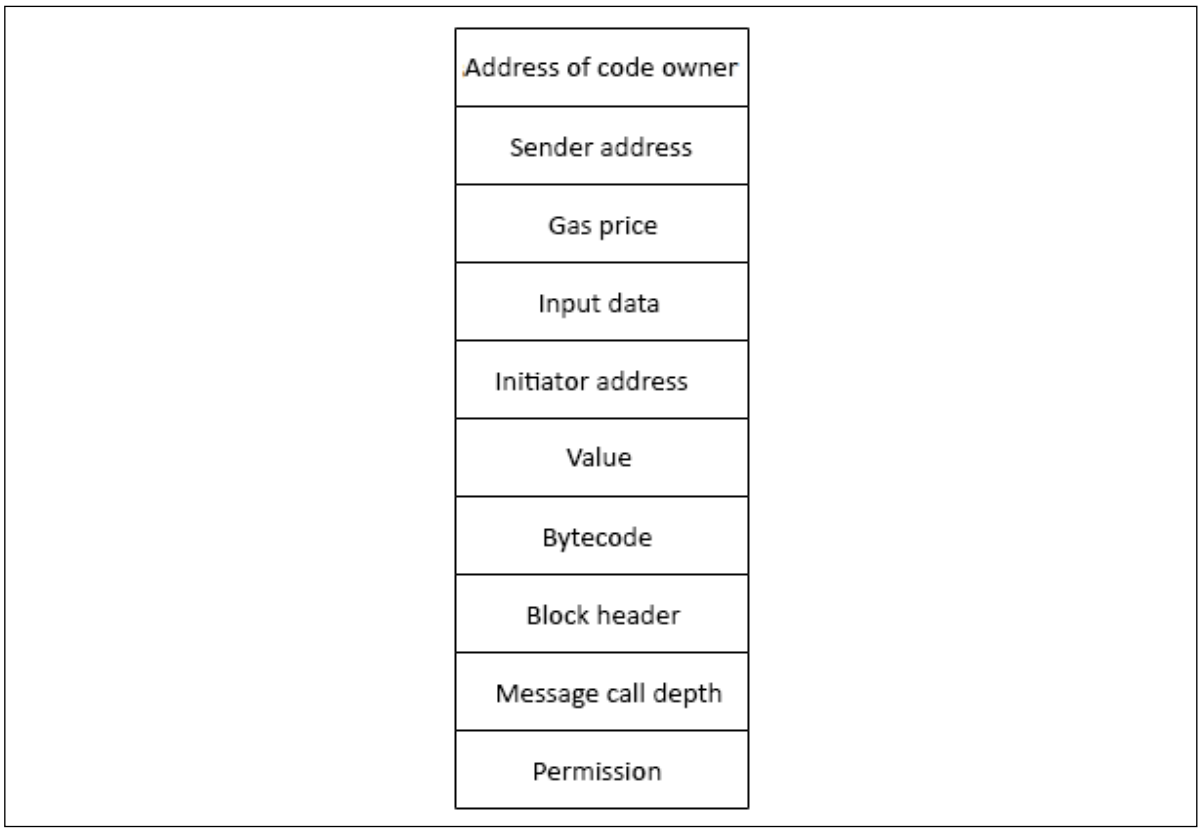


Figure 11.12: Execution environment tuple

The execution results in producing the resulting state, the gas remaining after the execution, the self-destruct or suicide set, log series, and any gas refunds.

The machine state

The machine state is also maintained internally, and updated after each execution cycle of the EVM. An iterator function (detailed in the next section) runs in the EVM, which outputs the results of a single cycle of the state machine.

The machine state is a tuple that consists of the following elements:

- Available gas
- The program counter, which is a positive integer of up to 256
- The contents of the memory (a series of zeroes of size 2^{256})
- The active number of words in memory (counting continuously from position 0)
- The contents of the stack

The EVM is designed to handle exceptions and will halt (stop execution) if any of the following exceptions should occur:

- Not having enough gas required for execution
- Invalid instructions
- Insufficient stack items
- Invalid destination of jump opcodes
- Invalid stack size (greater than 1,024)

Machine state can be viewed as a tuple, as shown in the following diagram:

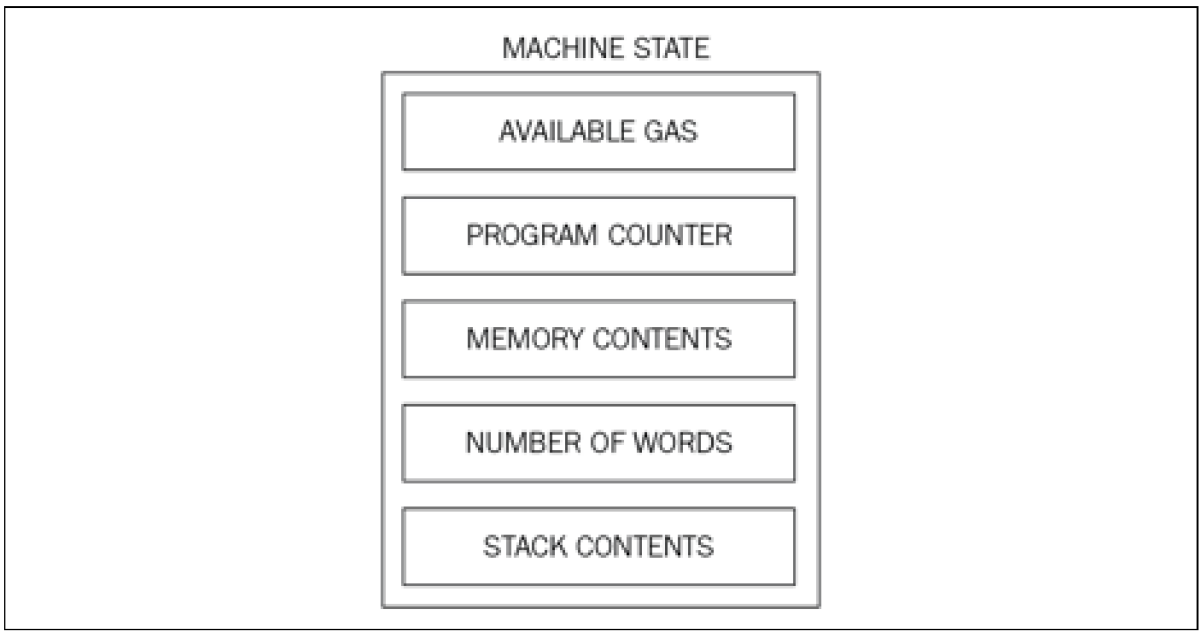


Figure 11.13: Machine state tuple

The iterator function

The iterator function mentioned earlier performs various vital 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**).

The EVM is also able to halt in normal conditions if `STOP`, `SUICIDE`, or `RETURN` opcodes are encountered during the execution cycle.

Smart contracts

We discussed smart contracts at length in the previous *Chapter 10*, *Smart Contracts*. It is sufficient to say here that Ethereum supports the development of smart contracts that run on the EVM. Different languages can be used to build smart contracts, and we will discuss this in the programming section and at a deeper level in *Chapter 14*, *Development Tools and Frameworks* and *Chapter 15*, *Introducing Web3*.

There are also various contracts that are available in precompiled format in the Ethereum blockchain to support different functions. These contracts, known as **precompiled contracts** or **native contracts**, are described in the following subsection.

These are not strictly smart contracts in the sense of user-programmed Solidity smart contracts, but are in fact functions that are available natively to support various computationally intensive tasks. They run on the local node and are coded within the Ethereum client; for example, `parity` or `geth`.

Native contracts

There are nine **precompiled contracts** or **native contracts** in the Ethereum Istanbul release. The following subsections outline these contracts and their details.

The elliptic curve public key recovery function

`ECDSARECOVER` (the ECDSA recovery function) is available at address `0x1`. It is denoted as `ECREC` and requires 3,000 gas in fees 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 ECC.

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* have been discussed in detail previously in this chapter.

The SHA-256-bit hash function

The SHA-256-bit hash function is a precompiled contract that is available at address `0x2` and produces a `SHA256` hash of the input. The gas requirement for SHA-256 (`SHA256`) depends on the input data size. The output is a 32-byte value.

The RIPEMD-160-bit hash function

The RIPEMD-160-bit hash function is used to provide a RIPEMD 160-bit hash and is available at address `0x3`. The output of this function is a 20-byte value. The gas requirement, similar to SHA-256, is dependent on the amount of input data.

The identity/datacopy function

The identity function is available at address `0x4` 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. The 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.

Big mod exponentiation function

This function implements a native big integer exponential modular operation. This functionality allows for RSA signature verification and other cryptographic operations. This is available at address `0x05`.

Elliptic curve point addition function

We discussed elliptic curve addition in detail at a theoretical level in *Chapter 4*, *Public Key Cryptography*. This is the implementation of the same elliptic curve point addition function. This contract is available at address `0x06`.

Elliptic curve scalar multiplication

We discussed elliptic curve multiplication (point doubling) in detail at a theoretical level in *Chapter 4*, *Public Key Cryptography*. This is the implementation of the same elliptic curve point multiplication function. Both elliptic curve addition and doubling functions allow for ZK-SNARKS and the implementation of other cryptographic constructs. This contract is available at `0x07`.

Elliptic curve pairing

The elliptic curve pairing functionality allows for performing elliptic curve pairing (bilinear maps) operations, which enables zk-SNARKS verification. This contract is available at address `0x08`.

Blake2 compression function 'F'

This precompiled contract allows the `BLAKE2b` hash function and other related variants to run on the EVM. This improves interoperability with Zcash and other Equihash-based PoW chains. This precompiled contract is implemented at address `0x09`. The EIP is available at `https://eips.ethereum.org/EIPS/eip-152`.

More information on the Blake hash function is available at `https://blake2.net`.

All the aforementioned precompiled contracts can potentially become native extensions and might be included in the EVM opcodes in the future.

All pre-compiled contracts are present in the `contracts.go` file in the Ethereum source code. It is available at the following link:

`https://github.com/ethereum/go-ethereum/blob/8bd37a1d919194d9a488d1cee823eaecfeb5ed9a/core/vm/contracts.go#L66`

Next hard fork release of Ethereum called Berlin will introduce new pre-compiled contracts for BLS12-381 curve operations along with other updates. You can follow the EIP-2070 for further updates on this release. EIP is available here at `https://eips.ethereum.org/EIPS/eip-2070`

We will continue our discussion on Ethereum in the next chapter. Readers are also encouraged to explore some extra content on Ethereum networks, and trading and investing, on this book's online resource page here: `https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf`.

Summary

This chapter started with a discussion on the history of Ethereum, the motivation behind Ethereum development, and Ethereum clients. Then, we introduced the core concepts of the Ethereum blockchain, such as the state machine model, the world and machine states, accounts, and types of accounts. Moreover, a detailed introduction to the core components of the EVM was also presented, along with some of the fundamentals of Ethereum trading and investing.

In the next chapter, we will continue to explore more Ethereum concepts. We will look at more ideas such as programming languages, blockchain data structures, blocks, mining, and various Ethereum clients.