
Ethereum Tests Documentation

Release 0.1

Ethereum Community

Oct 25, 2018

Contents:

1	Blockchain Tests	3
1.1	Test Implementation	3
1.2	Test Structure	3
2	General State Tests	7
2.1	Test Implementation	7
2.2	Test Structure	7
3	RLP Tests	11
3.1	Test Implementation	11
3.2	Test Structure	11
4	Difficulty Tests	13
4.1	Test Structure	13
5	Transaction Tests	15
5.1	Test Implementation	15
5.2	Test Structure	15
6	VM Tests	17
6.1	Test Implementation	17
6.2	Test Structure	18
7	Test Creation	21
8	Contribute to Docs	23
9	Indices and tables	25

Common tests for all clients to test against. The [git repo](#) updated regularly with new tests. This section describes basic test concepts and templates which are created by cpp-client.

Note: See *Contribute to Docs* if you want to help improve this documentation.

Blockchain Tests

The blockchain tests aim is to test the basic verification of a blockchain.

Location	/BlockchainTests
Supported Hardforks	Byzantium Constantinople EIP150 EIP158 Frontier Homestead
Status	Actively supported

A blockchain test is based around the notion of executing a list of single blocks, described by the `blocks` portion of the test. The first block is the modified genesis block as described by the `genesisBlockHeader` portion of the test. A set of pre-existing accounts are detailed in the `pre` portion and form the world state of the genesis block.

Of special notice is the [/BlockchainTests/GeneralStateTests](#) folder within the blockchain tests folder structure, which contains a copy of the *General State Tests* but executes them within the logic of the blockchain tests.

1.1 Test Implementation

It is generally expected that the test implementer will read `genesisBlockHeader` and `pre` and build the corresponding blockchain in the client. Then the new blocks, described by its RLP found in the `rlp` object of the `blocks` (RLP of a complete block, not the block header only), is read. If the client concludes that the block is valid, it should execute the block and verify the parameters given in `blockHeader` (block header of the new block), `transactions` (transaction list) and `uncleHeaders` (list of uncle headers). If the client concludes that the block is invalid, it should verify that no `blockHeader`, `transactions` or `uncleHeaders` object is present in the test. The client is expected to iterate through the list of blocks and ignore invalid blocks.

1.2 Test Structure

For a formal structure definition see also the related [JSON Schema](#) in the repo.

```
{
  "TESTNAME_Byzantium": {
    "blocks" : [
      {
        "blockHeader": { ... },
        "rlp": { ... },
        "transactions": { ... },
        "uncleHeaders": { ... }
      },
      {
        "blockHeader": { ... },
        "rlp": { ... },
        "transactions": { ... },
        "uncleHeaders": { ... }
      },
      { ... }
    ],
    "genesisBlockHeader": { ... },
    "genesisRLP": " ... ",
    "lastblockhash": " ... ",
    "network": "Byzantium",
    "postState": { ... },
    "pre": { ... },
    "sealEngine": [ "NoProof" | "Ethash" ]
  },
  "TESTNAME_EIP150": {
    ...
  }
  ...
}
```

1.2.1 The Blocks Section

The `blocks` section is a list of block objects, which have the following format:

- `rlp` section contains the complete rlp of the new block as described in the yellow paper in section 4.3.3.
- `blockHeader` section describes the block header of the new block in the same format as described in *genesisBlockHeader*.
- `transactions` section is a list of transactions which have the same format as in *Transaction Tests*.
- `uncleHeaders` section is a list of block headers which have the same format as described in *genesisBlockHeader*.

1.2.2 The genesisBlockHeader Section

coinbase: The 160-bit address to which all fees collected from the successful mining of this block be transferred, as returned by the **COINBASE** instruction.

difficulty: A scalar value corresponding to the difficulty level of this block. This can be calculated from the previous block's difficulty level and the timestamp, as returned by the **DIFFICULTY** instruction.

gasLimit: A scalar value equal to the current limit of gas expenditure per block, as returned by the **GASLIMIT** instruction.

number: A scalar value equal to the number of ancestor blocks. The genesis block has a number of zero.

timestamp: A scalar value equal to the reasonable output of Unix's `time()` at this block's inception, as returned by the `TIMESTAMP` instruction.

parentHash: The Keccak 256-bit hash of the parent block's header, in its entirety

bloom: The Bloom filter composed from indexable information (logger address and log topics) contained in each log entry from the receipt of each transaction in the transactions list.

extraData: An arbitrary byte array containing data relevant to this block. This must be 1024 bytes or fewer.

gasUsed: A scalar value equal to the total gas used in transactions in this block.

nonce: A 256-bit hash which proves that a sufficient amount of computation has been carried out on this block.

receiptTrie: The Keccak 256-bit hash of the root node of the trie structure populated with the receipts of each transaction in the transactions list portion of the block.

stateRoot: The Keccak 256-bit hash of the root node of the state trie, after all transactions are executed and finalisations applied.

transactionsTrie: The Keccak 256-bit hash of the root node of the trie structure populated with each transaction in the transactions list portion of the block.

uncleHash: The Keccak 256-bit hash of the uncles list portion of this block

1.2.3 Pre and postState Sections

- `pre` section: as described in *General State Tests*.
- `postState` section: as described in *General State Tests* (section - post).

1.2.4 Seal Engine

The `sealEngine` parameter (values: `NoProof` | `Ethash`) defines the seal engine the test is generated with. For tests with a value `NoProof` you can skip block validation which will speed up test execution. Note that this also means that you cannot rely on `PoW` specific block header values (`mixHash`, `nonce`) for tests labelled this way.

Currently this field is optional and there are still tests with no `sealEngine` parameter with the default here being the `NoProof` setting. So make sure to first check on parameter existence in your implementation.

1.2.5 Optional BlockHeader Information

`"blocknumber" = "int"` is section which defines what is the order of this block. It is used to define a situation when you have 3 blocks already imported but then it comes new version of the block 2 and 3 and thus you might have new best blockchain with blocks 1 2' 3' instead previous. If `blocknumber` is undefined then it is assumed that blocks are imported one by one. When running test, this field could be used for information purpose only.

`"chainname" = "string"` This is used for defining forks in the same test. You could mine blocks to chain "A": 1, 2, 3 then to chain "B": 1, 2, 3, 4 (chainB becomes primary). Then again to chain "A": 4, 5, 6 (chainA becomes primary) and so on. `chainname` could also be defined in uncle header section. If defined in uncle header it tells on which chain's block uncle header would be populated from. When running test, this field could be used for information purpose only.

`"chainnetwork" = "string"` Defines on which network rules this block was mined. (see the difference <https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2.mediawiki>). When running test, this field could be used for information purpose only.

General State Tests

The state tests aim is to test the basic workings of the state in isolation.

Location	<code>/GeneralStateTests</code>
Supported Hardforks	<code>Byzantium Constantinople EIP150 EIP158 Frontier Homestead</code>
Status	Actively supported

A state test is based around the notion of executing a single transaction, described by the `transaction` portion of the test. The overarching environment in which it is executed is described by the `env` portion of the test and includes attributes of the current and previous blocks. A set of pre-existing accounts are detailed in the `pre` portion and form the world state prior to execution. Similarly, a set of accounts are detailed in the `post` portion to specify the end world state. Since the data of the blockchain is not given, the opcode `BLOCKHASH` could not return the hashes of the corresponding blocks. Therefore we define the hash of block number `n` to be `SHA256("n")`.

The log entries (`logs`) as well as any output returned from the code (`output`) is also detailed.

2.1 Test Implementation

It is generally expected that the test implementer will read `env`, `transaction` and `pre` then check their results against `logs`, `out`, and `post`.

Note: The structure of state tests was reworked lately, see the associated discussion [here](#).

2.2 Test Structure

```
{
  "testname" : {
```

(continues on next page)

(continued from previous page)

```

"env" : {
  "currentCoinbase" : "address",
  "currentDifficulty" : "0x020000", //minimum difficulty for mining on blockchain
  "currentGasLimit" : "u64", //not larger then maxGasLimit = 0x7fffffffffffffff
  "currentNumber" : "0x01", //Irrelevant to hardfork parameters!
  "currentTimestamp" : "1000", //for blockchain version
  "previousHash" : "h256"
},
"post" : {
  "EIP150" : [
    {
      "hash" : "3e6dacc1575c6a8c76422255eca03529bbf4c0dda75dfc110b22d6dc4152396f",
      "indexes" : { "data" : 0, "gas" : 0, "value" : 0 }
    },
    {
      "hash" : "99a450d8ce5b987a71346d8a0a1203711f770745c7ef326912e46761f14cd764",
      "indexes" : { "data" : 0, "gas" : 0, "value" : 1 }
    },
    ...
  ],
  "EIP158" : [
    {
      "hash" : "3e6dacc1575c6a8c76422255eca03529bbf4c0dda75dfc110b22d6dc4152396f",
      "indexes" : { "data" : 0, "gas" : 0, "value" : 0 }
    },
    {
      "hash" : "99a450d8ce5b987a71346d8a0a1203711f770745c7ef326912e46761f14cd764",
      "indexes" : { "data" : 0, "gas" : 0, "value" : 1 }
    },
    ...
  ],
  "Frontier" : [
    ...
  ],
  "Homestead" : [
    ...
  ]
},
"pre" : {
  //same as for StateTests
},
"transaction" : {
  "data" : [ "" ],
  "gasLimit" : [ "285000", "100000", "6000" ],
  "gasPrice" : "0x01",
  "nonce" : "0x00",
  "secretKey" : "45a915e4d060149eb4365960e6a7a45f334393093061116b197e3240065ff2d8
↪",
  "to" : "095e7baea6a6c7c4c2dfeb977efac326af552d87",
  "value" : [ "10", "0" ]
}
}
}

```

2.2.1 The env Section

`currentCoinbase`

The current block's coinbase address, to be returned by the *COINBASE* instruction.

`currentDifficulty`

The current block's difficulty, to be returned by the *DIFFICULTY* instruction.

`currentGasLimit`

The current block's gas limit.

`currentNumber`

The current block's number. Also indicates network rules for the transaction. Since `blocknumber = 1000000` Homestead rules are applied to transaction. (see <https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2.mediawiki>)

`currentTimestamp`

The current block's timestamp.

`previousHash`

The previous block's hash.

2.2.2 The transaction Section

`data`

The input data passed to the execution, as used by the *CALLDATA...* instructions. Given as an array of byte values. See `$DATA_ARRAY`.

`gasLimit`

The total amount of gas available for the execution, as would be returned by the *GAS* instruction were it be executed first.

`gasPrice`

The price of gas for the transaction, as used by the *GASPRICE* instruction.

`nonce`

Scalar value equal to the number of transactions sent by the sender.

`address`

The address of the account under which the code is executing, to be returned by the *ADDRESS* instruction.

`secretKey`

The secret key as can be derived by the `v,r,s` values if the transaction.

`to`

The address of the transaction's recipient, to be returned by the *ORIGIN* instruction.

`value`

The value of the transaction (or the endowment of the create), to be returned by the *CALLVALUE'* instruction (if executed first, before any *CALL*).

2.2.3 The post Section

`Indexes` section describes which values from given array to set for transaction before it's execution on a pre state. Transaction now has `data`, `value`, and `gasLimit` as arrays. `post` section now has array of implemented forks. For each fork it has another array of execution results on that fork rules with post state root hash and transaction parameters.

2.2.4 The pre Section

The `pre` section have the format of a mapping between addresses and accounts. Each account has the format:

`balance`

The balance of the account.

`nonce`

The nonce of the account.

`code`

The body code of the account, given as an array of byte values. See `$DATA_ARRAY`.

`storage`

The account's storage, given as a mapping of keys to values. For key used notion of string as digital or hex number e.g: `"1200"` or `"0x04B0"` For values used `$DATA_ARRAY`.

Describes an **RLP** (<https://github.com/ethereum/wiki/wiki/RLP>) encoding using the .json file.

Location	/RLPTests
Supported Hardforks	Hardfork-independent
Status	Actively supported

3.1 Test Implementation

The client should read the rlp byte stream, **decode** and check whether the contents match its json representation. Then it should try do it reverse - **encode** json rlp representation into rlp byte stream and check whether it matches the given rlp byte stream.

If it is an invalid RLP byte stream in the test, then 'in' field would contain string `INVALID`.

Some RLP byte streams are expected to be generated by fuzz test suite. For those examples 'in' field would contain string `VALID` as it means that rlp should be easily decoded.

Note that RLP tests are testing a single RLP object encoding and not a stream of RLP objects in one array.

3.2 Test Structure

```
{
  "rlpTest": {
    "in": "dog",
    "out": "83646f67"
  },
  "multilist": {
    "in": [ "zw", [ 4 ], 1 ],
```

(continues on next page)

(continued from previous page)

```
    "out": "c6827a77c10401"
  },
  "validRLP": {
    "in": "VALID",
    "out": "c7c0c1c0c3c0c1c0"
  },
  "invalidRLP": {
    "in": "INVALID",
    "out": "bf0f000000000000021111"
  },
  ...
}
```

3.2.1 Sections

- in - json object (array, int, string) representation of the rlp byte stream (*except values VALID and INVALID)
- out - string of rlp bytes stream

Difficulty Tests

These tests are designed to just check the difficulty formula of a block.

Location	BasicTests (difficulty*.json)
Supported Hardforks	Test Networks Frontier Homestead
Status	Outdated

`difficulty = DIFFICULTY(currentBlockNumber, currentTimestamp, parentTimestamp, parentDifficulty)`

described at [EIP2](#) point 4 with homestead changes.

So basically this .json tests are just to check how this function is calculated on different function parameters (parent-Difficulty, currentNumber) in its extremum points.

There are several test files:

difficulty.json Normal Frontier/Homestead chain difficulty tests defined manually

difficultyFrontier.json Same as above, but auto-generated tests

difficultyMorden.json Tests for testnetwork difficulty. (it has different homestead transition block)

difficultyOlympic.json Olympic network. (no homestead)

difficultyHomestead.json Tests for homestead difficulty (regardless of the block number)

difficultyCustomHomestead.json Tests for homestead difficulty (regardless of the block number)

4.1 Test Structure

```
{
  "difficultyTest" : {
    "parentTimestamp" : "42",
    "parentDifficulty" : "1000000",
    "currentTimestamp" : "43",
```

(continues on next page)

(continued from previous page)

```
    "currentBlockNumber" : "42",  
    "currentDifficulty"  : "1000488"  
  }  
}
```

4.1.1 Sections

- `parentTimestamp` - indicates the timestamp of a previous block
- `parentDifficulty` - indicates the difficulty of a previous block
- `currentTimestamp` - indicates the timestamp of a current block
- `currentBlockNumber` - indicates the number of a current block (previous block number = `currentBlockNumber - 1`)
- `currentDifficulty` - indicates the difficulty of a current block

Transaction Tests

Describes a complete transaction and its RLP representation using the .json file.

Location	/TransactionTests
Supported Hardforks	Constantinople EIP158 Frontier Homestead
Status	Actively supported

5.1 Test Implementation

The client should read the rlp and check whether the transaction is valid, has the correct sender and corresponds to the transaction parameters. If it is an invalid transaction, the transaction and the sender object will be missing.

5.2 Test Structure

```
{
  "transactionTest1": {
    "rlp" : "bytearray",
    "sender" : "address",
    "blocknumber" : "1000000"
    "transaction" : {
      "nonce" : "int",
      "gasPrice" : "int",
      "gasLimit" : "int",
      "to" : "address",
      "value" : "int",
      "v" : "byte",
      "r" : "256 bit unsigned int",
      "s" : "256 bit unsigned int",
      "data" : "byte array"
    }
  }
}
```

(continues on next page)

(continued from previous page)

```
    },
    "invalidTransactionTest": {
      "rlp" : "bytearray",
    },
    ...
  }
}
```

5.2.1 Sections

- `rlp` - RLP encoded data of this transaction
- `transaction` - transaction described by fields
- `nonce` - A scalar value equal to the number of transactions sent by the sender.
- `gasPrice` - A scalar value equal to the number of wei to be paid per unit of gas.
- `gasLimit` - A scalar value equal to the maximum amount of gas that should be used in executing this transaction.
- `to` - The 160-bit address of the message call's recipient or empty for a contract creation transaction.
- `value` - A scalar value equal to the number of wei to be transferred to the message call's recipient or, in the case of contract creation, as an endowment to the newly created account.
- `v, r, s` - Values corresponding to the signature of the transaction and used to determine the sender of the transaction.
- `sender` - the address of the sender, derived from the `v,r,s` values.
- `blocknumber` - indicates network rules for the transaction. Since `blocknumber = 100000` Homestead rules are applied to transaction. (see <https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2.mediawiki>)

The VM tests aim is to test the basic workings of the VM in isolation.

Location	<code>/VMTests</code>
Supported Hardforks	Currently only Homestead
Status	Actively supported

This is specifically not meant to cover transaction, creation or call processing, or management of the state trie. Indeed at least one implementation tests the VM without calling into any Trie code at all.

A VM test is based around the notion of executing a single piece of code as part of a transaction, described by the `exec` portion of the test. The overarching environment in which it is executed is described by the `env` portion of the test and includes attributes of the current and previous blocks. A set of pre-existing accounts are detailed in the `pre` portion and form the world state prior to execution. Similarly, a set of accounts are detailed in the `post` portion to specify the end world state.

The gas remaining (`gas`), the log entries (`logs`) as well as any output returned from the code (`out`) is also detailed.

6.1 Test Implementation

It is generally expected that the test implementer will read `env`, `exec` and `pre` then check their results against `gas`, `logs`, `out`, `post` and `callcreates`. If an exception is expected, then latter sections are absent in the test. Since the reverting of the state is not part of the VM tests.

Because the data of the blockchain is not given, the opcode `BLOCKHASH` could not return the hashes of the corresponding blocks. Therefore we define the hash of block number `n` to be `SHA3-256("n")`.

Since these tests are meant only as a basic test of VM operation, the `CALL` and `CREATE` instructions are not actually executed. To provide the possibility of testing to guarantee they were actually run at all, a separate portion `callcreates` details each `CALL` or `CREATE` operation in the order they would have been executed. Furthermore, gas required is simply that of the VM execution: the gas cost for transaction processing is excluded.

6.2 Test Structure

```
{
  "test name 1": {
    "env": { ... },
    "pre": { ... },
    "exec": { ... },
    "gas": { ... },
    "logs": { ... },
    "out": { ... },
    "post": { ... },
    "callcreates": { ... }
  },
  "test name 2": {
    "env": { ... },
    "pre": { ... },
    "exec": { ... },
    "gas": { ... },
    "logs": { ... },
    "out": { ... },
    "post": { ... },
    "callcreates": { ... }
  },
  ...
}
```

6.2.1 The env Section

- `currentCoinbase`: The current block's coinbase address, to be returned by the `COINBASE` instruction.
- `currentDifficulty`: The current block's difficulty, to be returned by the `DIFFICULTY` instruction.
- `currentGasLimit`: The current block's gas limit.
- `currentNumber`: The current block's number.
- `currentTimestamp`: The current block's timestamp.
- `previousHash`: The previous block's hash.

6.2.2 The exec Section

- `address`: The address of the account under which the code is executing, to be returned by the `ADDRESS` instruction.
- `origin`: The address of the execution's origin, to be returned by the `ORIGIN` instruction.
- `caller`: The address of the execution's caller, to be returned by the `CALLER` instruction.
- `value`: The value of the call (or the endowment of the create), to be returned by the `CALLVALUE` instruction.
- `data`: The input data passed to the execution, as used by the `CALLDATA...` instructions. Given as an array of byte values. See `$DATA_ARRAY`.
- `code`: The actual code that should be executed on the VM (not the one stored in the `state(address)`) . See `$DATA_ARRAY`.
- `gasPrice`: The price of gas for the transaction, as used by the `GASPRICE` instruction.

- `gas`: The total amount of gas available for the execution, as would be returned by the `GAS` instruction were it be executed first.

6.2.3 The pre and post Section

The `pre` and `post` sections each have the same format of a mapping between addresses and accounts. Each account has the format:

- `balance`: The balance of the account.
- `nonce`: The nonce of the account.
- `code`: The body code of the account, given as an array of byte values. See `$DATA_ARRAY`.
- `storage`: The account's storage, given as a mapping of keys to values. For key used notion of string as digital or hex number e.g: "1200" or "0x04B0" For values used `$DATA_ARRAY`.

6.2.4 The callcreates Section

The `callcreates` section details each `CALL` or `CREATE` instruction that has been executed. It is an array of maps with keys:

- `data`: An array of bytes specifying the data with which the `CALL` or `CREATE` operation was made. In the case of `CREATE`, this would be the (initialisation) code. See `$DATA_ARRAY`.
- `destination`: The receipt address to which the `CALL` was made, or the null address ("0000...") if the corresponding operation was `CREATE`.
- `gasLimit`: The amount of gas with which the operation was made.
- `value`: The value or endowment with which the operation was made.

6.2.5 The logs Section

The `logs` sections contains the hex encoded hash of the rlp encoded log entries, reducing the overall size of the test files while still verifying that all of the data is accurate (at the cost of being able to read what the data should be). Each logentry has the format:

```
keccak(rlp.encode(log_entries))
```

(see https://github.com/ethereum/py-evm/blob/7a96fa3a2b00af9bea189444d88a3cce6a6be05f/eth/tools/_utils/hashing.py#L8-L16)

6.2.6 The gas and output Keys

Finally, there are two simple keys, `gas` and `out`:

- `gas`: The amount of gas remaining after execution.
- `out`: The data, given as an array of bytes, returned from the execution (using the `RETURN` instruction). See `$DATA_ARRAY`.
`$DATA_ARRAY` - type that intended to contain raw byte data and for convenient of the users is populated with three types of numbers, all of them should be converted and concatenated to a byte array for VM execution.
 1. number - (unsigned 64bit)
 2. "longnumber" - (any long number)
 3. "0xhex_num" - (hex format number)

CHAPTER 7

Test Creation

Instructions on how to create tests and how to use the `cpp-client testeth` tool can be found in the `c++` documentation [test generation chapter](#).

Since docker images pointed to in the test generation docs are often outdated, you will probably have to rely on your own [source build](#) of the Aleth C++ Ethereum client for generating tests (start this early on since this is taking some time).

CHAPTER 8

Contribute to Docs

This documentation has been build using the Python [Sphinx](#) documentation tool.

Since the [Ethereum tests](#) repository is very large to clone locally, a convenient way to contribute to the documentation is to make a fork of the test repo, add the changes online with the GitHub [reStructuredText](#) editor and then open a PR.

If you want to clone to your desk you might want to make use of `git clone --depth 1` for faster download.

You can build the documentation by running `make html` from the `docs` directory in the tests repository.

CHAPTER 9

Indices and tables

- `genindex`
- `modindex`
- `search`