Ethereum Tests Documentation

Release 0.2

Ethereum Community

Oct 19, 2021

Tutorial:

Common tests for all clients to test against. The git repo updated regulary 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.

CHAPTER 1

Retesteth

Ori Pomerantz

Note: This document is a tutorial. For reference on the **retesteth** options, look here.

1.1 Retesteth through the Web

The easiest way to run the tests is through the web interface.

1.1.1 Request Helper

To run an existing state test, you can use the **request helper**. You set these parameters:

Parameter	Meaning	Sample Value
GeneralStateTests/	test suite	stExample
–singletest	name of test	add11
–clients	client to use	t8ntool (the value for geth)
–singlenet	fork to use	Berlin
–vmtrace	trace to produce	raw
–verbosity	log verbosity	none

When a test file contains multiple tests you can restrict which ones you'll run with the **-d**, **-g**, and **-v** parameters.

1.1.2 Request Single File

You can run a single test, either state test or blockchain test, using the **request single file** option. You specify the test type and then upload a test file. Here are the parameters:

Parameter	Meaning	Sample Value
-t	test suite	test type (state or blockchain)
–testfile	The file to test	you upload this file
–clients	client to use	t8ntool (the value for geth)
–vmtrace	trace to produce	raw
–filltests	test file type	see below

If –filltests is set to **none**, you need to upload a generated test file. You can find those here (for state tests), and here (for blockchain tests).

If –filltests is set to **filltests** then you can upload a filler test file, which you can write yourself. This is documented in this tutorial (for state tests) and this one (for blockchain tests).

1.1.3 Custom Command

The command line parameters for **retesteth** are documented here. You can use this option to run whatever parameters you want.

1.2 Retesteth in a Docker Container

If you want to run the tests locally you can run **retesteth** inside a Docker container.

These directions are written using Debian Linux 10 on Google Cloud Platform, but should work with minor changes on any other version of Linux running anywhere else with an Internet connection.

1. Install docker. You may need to reboot afterwards to get the latest kernel version.

```
sudo apt install -y wget docker docker.io
```

2. Download the **retesteth** docker image. It is a tar file.
3. Load the docker image:

```
sudo docker load -i dretest*.tar
```

4. Download the **dretesteth.sh** script.

```
wget https://raw.githubusercontent.com/ethereum/retesteth/master/dretesteth.sh
chmod +x dretesteth.sh
```

5. Download the tests:

```
git clone --branch develop https://github.com/ethereum/tests.git
```

6. Run a test. This has two purposes:
   - Create the **retesteth** configuration directories in **~/tests/config**, where you can modify them.
   - A sanity check (that you can run tests successfully).

```
sudo ./dretesteth.sh -t GeneralStateTests/stExample -- \
 --testpath ~/tests --datadir /tests/config
```

   The output should be similar to:

```
Running 1 test case...
Running tests using path: /tests
Active client configurations: 't8ntool '
Running tests for config 'Ethereum GO on StateTool'
Test Case "stExample":
100%
*** No errors detected
*** Total Tests Run: 1
```

Note: The **/tests** directory is referenced inside the docker container. It is the same as the **~/tests** directory outside it.

7. To avoid having to run with **sudo** all the time, add yourself to the **docker** group.

```
sudo usermod -a -G docker `whoami`
```

1.2.1 How Does This Work?

A docker container is similar to a virtual machine, except that it doesn't run a separate instance of the operating system inside itself so it takes far less resources. One of the features of docker is that it can mount a directory of the host computer inside its own file system. The **–testpath** parameter to **dretesteth.sh** tells it what directory to mount, in this case **~/tests** which you just cloned from github. It mounts it as **/tests** inside the container.

By default the **retesteth** configuration files are in **~/.retesteth**. However, that directory is not accessible to us outside the docker. Instead, we use **–datadir /tests/config** to tell it to use (or create) the configuration in what appears to us to be **~/tests/config**, which is easily accessible.

1.3 Test Against Your Client

There is an instance of **geth** inside the docker container that you can run tests against. However, unless you are specifically developing tests what you want is to test your client. There are several ways to do this:

- Keep the client on the outside and keep the configuration files intact
- Put your client, and any prerequisites, inside the docker and change the configuration files
- Keep your client on the outside and connect to it through the network and change the configuration files

When we ran the test in the previous section we also created those configuration files in **~/tests/config**, but they were created as being owned by root. If you need to edit them, change the permissions of the config files. To change the configuration files to your own user, run this command:

```
sudo find ~/tests/config -exec chown $USER {} \; -print
```

If you look inside **~/tests/config**, you'll see a directory for each configured client. Typically this directory has these files:

- **config**, which contains the configuration for the client:
  - The communication protocol to use with the client (typically TCP)
  - The address(es) to use with that protocol
  - The forks the client supports

  - The exceptions the client can throw, and how **retesteth** should interpret them. This is particularly important when testing the client's behavior when given invalid blocks.

- **start.sh**, which starts the client inside the docker image
- **stop.sh**, which stops the client instance(s)
- **genesis**, a directory which includes the genesis blocks for various forks the client supports. If this directory does not exist for a client, it uses the genesis blocks for the default client.

Click here for additional documentation. Warning: This documentation may not be up to date

1.3.1 Client Outside the Docker, Keep Configuration Files Intact

If you want to run your client outside the docker without changing the configuration, these are the steps to follow.

1. Make sure that the routing works in both directions (from the docker to the client and from the client back to the docker). You may need to configure network address translation.
2. Run your client. Make sure that the client accepts requests that don't come from **localhost**. For example, to run **geth** use:

```
geth --http --http.addr 0.0.0.0 retesteth
```

   To run **besu** use:

```
docker run -p 8545:8545 -p 13001:30303 \
     hyperledger/besu:latest retesteth --rpc-http-port 8545 \
     --host-allowlist '*'
```

3. Run the test the same way you would for a client that runs inside docker, but with the addition of the **–nodes** parameter. Also, make sure the **–clients** parameter is set to the client you're testing.

```
./dretesteth.sh -t BlockchainTests/ValidBlocks/VMTests -- \
   --testpath ~/tests --datadir /tests/config --clients geth \
   --nodes \<ip\>:\<port, usually defaults to 8545\>
```

1.3.2 Client Inside the Docker, Modify Configuration Files

If you want to run your client inside the docker, follow these steps:

1. Move the client into **~/tests**, along with any required infrastructure (virtual machine software, etc). If you just want to test the directions right now, you can download geth here.
2. Modify the appropriate **start.sh** to run your version of the client instead. For example, you might edit **~/tests/config/geth/start.sh** to replace **geth** with **/tests/geth** in line ten if you put your version of **geth** in **~/tests**.
3. Run the tests, adding the **–clients <name of client>** parameter to ensure you're using the correct configuration. For example, run this command to run the virtual machine tests on **geth**:

```
./dretesteth.sh -t BlockchainTests/ValidBlocks/VMTests -- --testpath \
~/tests --datadir /tests/config --clients geth
```

1.3.3 Client Outside the Docker, Modify Configuration Files

If you want to run your client outside the docker and specify the connectivity in the configuration files, these are the steps to follow:

1. Create a client in **~/tests/config** that doesn't have **start.sh** and **stop.sh**. Typically you would do this by copying an existing client, for example:

```
mkdir ~/tests/config/gethOutside
cp ~/tests/config/geth/config ~/tests/config/gethOutside
```

2. If you want to specify the IP address and port in the **config** file, modify the host in the **socketAddress** to the appropriate remote address. This address needs to work with the JSON over RPC test protocol.

   For example,

```
{
   "name" : "Ethereum GO on TCP",
   "socketType" : "tcp",
   "socketAddress" : [ "10.128.0.14:8545" ],
   ...
}
```

3. Make sure that the routing works in both directions (from the docker to the client and from the client back to the docker). You may need to configure network address translation.

4. Run your client. Make sure that the client accepts requests that don't come from **localhost**. For example, to run **geth** use:

```
geth --http --http.addr 0.0.0.0 retesteth
```

5. Run the test the same way you would for a client that runs inside docker:

```
./dretesteth.sh -t BlockchainTests/ValidBlocks/VMTests -- \
    --testpath ~/tests --datadir /tests/config --clients gethOutside
```

1.4 Running Multiple Threads

To improve performance you can run tests across multiple threats. To do this:

1. If you are using **start.sh** start multiple nodes with different ports
2. Provide the IP addresses and ports of the nodes, either in the **config** file or the **–nodes** parameter
3. Run with the parameters **-j <number of threads>**.

1.5 Using the Latest Version

The version of retesteth published as a docker file may not have the latest updates. If you want the latest features, you need to build an image from the **develop** branch yourself:

1. Install docker.

```
sudo apt install -y wget docker docker.io
```

2. Download the **dretesteth.sh** script and the **Dockerfile**. Make sure to do this in an otherwise empty directory, because the docker builder copies everything in or below the directory where **Dockerfile** is located.

```
mkdir ~/retestethBuild
cd ~/retestethBuild
wget https://raw.githubusercontent.com/ethereum/retesteth/develop/dretesteth.sh
chmod +x dretesteth.sh
wget https://raw.githubusercontent.com/ethereum/retesteth/develop/Dockerfile
```

3. Modify the **RUN git clone** line in the **Dockerfile** to change the **-b** parameter from **master** to **develop**.
4. Build the docker image yourself:

```
sudo ./dretesteth.sh build
```

Note: This is a slow process. It took me about an hour on a GCP **e2-medium** instance.

1.6 Conclusion

In most cases people don't start their own client from scratch, but modify an existing client. If the existing client is already configured to support **retesteth**, you should now be able to run tests on a modified version to ensure it still conforms to Ethereum specifications. If you are writing a completely new client, you still need to implement the RPC calls that **retesteth** uses and to write the appropriate configuration (**config**, **start.sh**, and **stop.sh**) for it.

There are several actions you might want to do with **retesteth** beyond testing a new version of an existing client. Here are links to documentation. Note that it hasn't been updated in a while, so it may not be accurate.

- **Add configuration for a new client**. To do this you need to add retesteth support to the client itself and create a new config for it
- **Test with a new fork of Ethererum**. New forks usually mean new opcodes. Therefore, you will need a docker with a new version of lllc.

If you want to write your own tests, read the next tutorial.

CHAPTER 2

State Transition Tests

Ori Pomerantz

In this tutorial you learn how to write and execute Ethereum state transition tests. These tests can be very simple, for example testing a single evm assembler opcode, so this is a good place to get started. This tutorial is not intended as a comprehensive reference, look in the table of content on the left.

2.1 The Environment

Before you start, make sure you read and understand the Retesteth Tutorial, and create the docker environment explained there.

2.2 Compiling Your First Test

Before we get into how tests are built, lets compile and run a simple one.

1. The source code of the tests is in **tests/src**. It is complicated to add another tests directory, so we will use **GeneralStateTestsFiller/stExample**.

```
cd ~/tests/src/GeneralStateTestsFiller/stExample
cp ~/tests/docs/tutorial_samples/01* .
cd ~
```

2. The source code of tests doesn't include all the information required for the test. Instead, you run **retesteth.sh**, and it runs a client with the Ethereum Virtual Machine (evm) to fill in the values. This creates a compiled version in **tests/GeneralStateTests/stExample**.

```
./dretesteth.sh -t GeneralStateTests/stExample -- \
    --singletest 01_add22 --testpath ~/tests \
    --datadir /tests/config --filltests
sudo chown $USER tests/GeneralStateTests/stExample/*
```

3. Run the test normally, with verbose output:

```
./dretesteth.sh -t GeneralStateTests/stExample -- \
   --singletest 01_add22 --testpath ~/tests \
   --datadir /tests/config --clients geth --verbosity 5
```

2.2.1 The Source Code

Now that we've seen that the test works, let's go through it line by line. This test specification is written in YAML, if you are not familiar with this format click here.

All the fields are defined under the name of the test. Note that YAML comments start with a hash (**#**) and continue to the end of the line.

If you want to follow along with the full source code You can see the complete code, here

```
# The name of the test
01_add22:
```

This is the general Ethereum environment before the transaction:

```
env:
    currentCoinbase: 2adc25665018aa1fe0e6bc666dac8fc2697ff9ba
    currentDifficulty: '0x20000'
    currentGasLimit: 10_000_000
```

You can put underscores (_) in numbers to make them more readable.

```
currentNumber: 1
currentTimestamp: "1000"
previousHash: 5e20a0453cecd065ea59c37ac63e079ee08998b6045136a8ce6635c7912ec0b6
```

This is where you put human readable information. In contrast to # comments, these comment fields get copied to the compiled JSON file for the test.

```
_info:
  comment: "You can put a comment here"
```

These are the relevant addresses and their initial states before the test starts:

```
pre:
```

This is a contract address. As such it has code, which can be in one of three languages:

1. Ethereum virtual machine (EVM) machine language
2. Lisp Like Language (lll). One advantage of lll is that it lets us use Ethereum Assembler almost directly.
3. Solidity, which is the standard language for Ethereum contracts. Solidity is well known, but it is not ideal for VM tests because it adds its own code to compiled contracts. Click here for a test written in Solidity.
4. The Yul language, which is a low level language for the EVM. Click here for a test written in Yul.

```
095e7baea6a6c7c4c2dfeb977efac326af552d87:
  balance: '0x0ba1a9ce0ba1a9ce'
```

LLL code can be very low level. In this case, **(ADD 2 2)** is translated into three opcodes:

- PUSH 2

- PUSH 2
- ADD (which pops the last two values in the stack, adds them, and pushes the sum into the stack).

This expression **[[0]]** is short hand for **(SSTORE 0 <the value at the top of the stack>)**. It stores the value (in this case, four) in location 0.

```
code: |
  {
    ; Add 2+2
    [[0]] (ADD 2 2)
  }
  nonce: '0'
```

Every address in Ethereum has associated storage, which is essentially a lookup table. You can read more about it here. In this case the storage is initially empty.

```
storage: {}
```

This is a "user" address. As such, it does not have code. Note that you still have to specify the storage.

```
a94f5374fce5edbc8e2a8697c15331677e6ebf0b:
  balance: '0x0ba1a9ce0ba1a9ce'
  code: '0x'
  nonce: '0'
  storage: {}
```

This is the transaction that will be executed to check the code. There are several scalar fields here:

- **gasPrice** is the price of gas in Wei. Note that starting with the London fork the block base fee is ten by default, and a lower gasPrice will get rejected.
- **nonce** has to be the same value as the user address
- **to** is the contract we are testing. If you want to create a contract, keep the **to** definition, but leave it empty.

Additionally, these are several fields that are lists of values. The reason to have lists instead of a single value is to be able to run multiple similar tests from the same file (see the **Multitest Files** section below).

- **data** is the data we send
- **gasLimit** is the gas limit
- **value** is the amount of Wei we send with the transaction

```
transaction:
  data:
  - '0x10'
  gasLimit:
  - '80000000'
  gasPrice: 1000
  nonce: '0'
  to: 095e7baea6a6c7c4c2dfeb977efac326af552d87
  value:
  - '1'
```

This is the state we expect after running the transaction on the **pre** state. The **indexes:** subsection is used for multitest files, for now just copy and paste it into your tests.

```
expect:
   - indexes:
       data: !!int -1
       gas:  !!int -1
       value: !!int -1
     network:
       - '>=London'
```

We expect the contract's storage to have the result, in this case 4.

```
result:
  095e7baea6a6c7c4c2dfeb977efac326af552d87:
    storage:
      0x00: 0x04
```

2.2.2 Failing a Test

To verify that **retesteth** really does run tests, lets fail one. The **02_fail** test is almost identical to **01_add22**, except that it expects to see that 2+2=5. Here are the steps to use it.

1. Copy the test to the *stExample* directory:

   ```
   cp ~/tests/docs/tutorial_samples/02* ~/tests/src/GeneralStateTestsFiller/stExample
   ```

2. Fill the information and run the test:

   ```
   ./dretesteth.sh -t GeneralStateTests/stExample -- \
      --singletest 02_fail --testpath ~/tests \
      --datadir /tests/config --filltests
   ```

3. Delete the test so we won't see the failure when we run future tests (you can run all the tests in a directory by omitting the **–singletest** parameter:

   ```
   rm ~/tests/src/GeneralStateTestsFiller/stExample/02_*
   ```

2.2.3 Tests that are Supposed to Fail

When a test transaction is supposed to fail, you add an **expectException:** section to the **result**. You can see a complete example in 10_expectExceptionFiller

```
expect:
- indexes:
    data: !!int -1
    gas:  !!int -1
    value: !!int -1
  network:
    - '>=London'
  expectException:
    '>=London': TR_FeeCapLessThanBlocks
  result: {}  # No point checking the result when no transaction happened
```

You can see the complete list of supported exceptions either in the config file for the client, or in the retesteth source code.

Note that running out of gas is not an exception. Technically speaking a transaction that runs out of gas is successful, it is just reverted.

2.3 Yul Tests

Yul is a language that is very close to EVM assembler. As such it is a good language for writing tests. You can see a Yul test at tests/docs/tutorial_samples/09_yulFiller.yml.

This is a sample contract:

```
ccccccccccccccccccccccccccccccccccccccc:
  balance: '0x0ba1a9ce0ba1a9ce'
  code: |
   :yul {
     let cellAddr := sub(10,10)

     sstore(cellAddr,add(60,9))
   }
  nonce: 1
  storage: {}
```

It is very similar to an LLL test, except for having the **:yul** keyword before the opening curly bracket ({).

2.4 Solidity Tests

You can see a solidity test at tests/docs/tutorial_samples/03_solidityFiller.yml. Here are the sections that are new.

Note: The Solidity compiler adds a lot of extra code that handles ABI encoding, ABI decoding, contract constructors, etc. This makes tracing and debugging a lot harder, which makes Solidity a bad choice for most Ethereum client tests.

This feature is available for tests where it is useful, but LLL or Yul is usually a better choice.

You can have a separate **solidity:** section for your code. This is useful because Solidity code tends to be longer than LLL (or Yul) code.

```
solidity: |
    // SPDX-License-Identifier: GPL-3.0
```

The **retesteth** docker only includes one version of the Solidity compiler, so it is best not to have a **pragma solidity** line.

```
contract Test {
```

Solidity keeps state variables in the storage, starting with location 0. We can use state variables for the results of operations, and check them in the **expect:** section

```
uint256 storageVar = 0xff00ff00ff00ff00;
function val2Storage(uint256 addr, uint256 val) public
{
  storageVar = val;
```

Another possibility is to use the SSTORE opcode directly to write to storage. This is the format to embed assembly into Solidity.

```
    assembly { sstore(addr, val) }
  }   // function val2Storage
}     // contract Test
```

To specify a contract's code you can use **:solidity <name of contract>**. Alternatively, you can put the solidity code directly in the account's **code:** section if it has **pragma solidity** (otherwise it is compiled as LLL).

```
pre:
  ccccccccccccccccccccccccccccccccccccccc:
    balance: '0x0ba1a9ce0ba1a9ce'
    code: ':solidity Test'
    nonce: '0'
    storage: {}
```

In contrast to LLL, Solidity handles function signatures and parameters for you. Therefore, the transaction data has to conform to the Application Binary Interface (ABI). You do not have to calculate the data on your own, just start it with **:abi** followed by the function signature and then the parameters. These parameters can be bool, uint, single dimension arrays, and strings.

Note: ABI support is a new feature, and may be buggy. Please report any bugs you encounter in this feature.

```
transaction:
  data:
  - :abi val2Storage(uint256,uint256) 5 69
  gasLimit:
  - '80000000'
```

The other sections of the test are exactly the same as they are in an LLL test.

2.4.1 ABI values

These are examples of the values that **:abi** can have.

- **:abi baz(uint32,bool) 69 1**: Call **baz** with a 32 bit value (69) and a true boolean value
- **:abi bar(bytes3[2]) ["abc", "def"]**: Call **bar** with a two value array, each value three bytes
- **:abi sam(bytes,bool,uint256[]) "dave" 0 [1,2,3]**: Call **sam** with a string ("dave"), a false boolean value, and an array of three 256 bit numbers.
- **:abi f(uint256,uint32[],bytes10,bytes) 0x123 [0x456, 0x789] "1234567890" "Hello, world"**: Call **f** with these parameters
  - An unsigned 256 bit integer
  - An array of 32 bit values (it can be any size)
  - A string of ten bytes
  - A string which could be any size
- **:abi g(uint256[][],string[]) [[1,2],[3],[4,5] ["one","two","three"]**: Call **g** with two parameters, a two dimensional array of uint256 values and an array of strings.
- **:abi h(uint256,uint32[],bytes10,bytes) 291 [1110,1929] "1234567890" "Hello, world!"**: Call **h** with a uint256, an array of uint32 values of unspecified size, ten bytes, and a parameter with an unspecified number of bytes.

- **:abi ff(uint256,address) 324124 "0xcd2a3d9f938e13cd947ec05abc7fe734df8dd826"**: Call **ff** with a uint256 and an address (Ethererum addresses are twenty bytes).

2.5 Multitest Files

It is possible to combine multiple similar tests in one file. Here is an example.

There are two steps to doing that:

- Modify the **transaction:** section. This section has three subsections that are lists. You can add multiple values to the **data:**, **gasLimit:**, and **value:**.

  For example:

```
transaction:
   data:
   - :abi val2Storage(uint256,uint256) 0x10 0x10
   - :abi val2Storage(uint256,uint256) 0x11 0x11
   - :abi val2Storage(uint256,uint256) 0x11 0x12
   - :abi val2Storage(uint256,uint256) 0x11 0x11
   gasLimit:
   - '80000000'
   gasPrice: '1'
   nonce: '0'
   to: cccccccccccccccccccccccccccccccccccccccc
   secretKey: "45a915e4d060149eb4365960e6a7a45f334393093061116b197e3240065ff2d8"
   value:
   - 0
```

- The **expect:** section is also a list, and can have multiple values. Just put the indexes to the correct **data**, **gas**, and **value** values, and have the correct response in the **result:** section.

  For example:

```
expect:
```

  The indexes are integer values. By default YAML values are strings. The **!!int** overrides this. These are all the first values in their lists, so the data is equivalent to the call **val2Storage(0x10, 0x10)**.

```
- indexes:
    data: !!int 0
    gas:  !!int 0
    value: !!int 0
  network:
    - '>=Istanbul'
  result:
    cccccccccccccccccccccccccccccccccccccccc:
      storage:
        0:    0x10
        0x10: 0x10
```

  This is for the second and fourth items in the **data:** subsection above. When you have multiple values that produce the same test results, you can specify **data**, **gas**, or **value** as a list instead of a single index.

```
- indexes:
    data:
    - !!int 1
```

(continues on next page)

(continued from previous page)

```
    - !!int 3
    gas:  !!int 0
    value: !!int 0
  network:
    - '>=Istanbul'
  result:
    cccccccccccccccccccccccccccccccccccccccc:
      storage:
        0:    0x11
        0x11: 0x11
```

2.5.1 Multiple Tests, Same Result

When you have multiple tests that produce the same results, you do not have to list them individually in the **expect:** section.

Range. You can specify a range, such as **4-6**, inside the **expect.data:** list. Remember *not* to specify !!int, the range is a string, not an integer.

Label. You can preface the value with **:label <word> <value>**:

```
transaction:
  data:
  - :label odd  :abi f(uint) 1
  - :label even :abi f(uint) 2
  - :label odd  :abi f(uint) 3
  - :label even :abi f(uint) 4
  - :label odd  :abi f(uint) 5
  - :label even :abi f(uint) 6
  - :label odd  :abi f(uint) 7
  - :label even :abi f(uint) 8
```

In the **expect.data:** list, you specify **:label <word>** and it applies to every value that has that label.

```
expect:
  - indexes:
      data:
      - :label odd
      - :label even
      gas: !!int -1
      value: !!int -1
```

2.6 Conclusion

At this point you should be able to run simple tests that verify the EVM opcodes work as well as more complex algorithms work as expected. You are, however, limited to a single transaction in a single block. In a next tutorial, *Blockchain Tests*, you will learn how to write blockchain tests that can involve multiple blocks, each of which can have multiple transactions.

CHAPTER 3

Blockchain Tests

Ori Pomerantz

In this tutorial you learn how to use the skills you learned writing state tests to write blockchain tests. These tests can include multiple blocks and each of those blocks can include multiple transactions.

3.1 The Environment

Before you start, make sure you create the retesteth tutorial and create the environment explained there. Also make sure you read and understand the state transition tests tutorial.

3.2 Types of Blockchain Tests

If you go to **tests/src/BlockchainTestsFiller** you will see three different directories.

- **ValidBlocks** are tests that only have valid blocks, which the client should accept.
- **InvalidBlocks** are tests that should raise an exception because they include invalid blocks.
- **TransitionTests** are tests that verify the transitions between different versions of the Ethereum protocol (called forks) are handled correctly. These tests are very important, but the people who write them are typically the people who write the tests software so I am not going to explain them here.

3.3 Valid Block Tests

There is a valid block test in tests/docs/tutorial_samples/05_simpleTxFiller.yml. We copy it to **bcExample**.

```
mkdir ~/tests/src/Blo*/Val*/bcExample*
cp ~/tests/docs/tu*/05_* ~/tests/src/Blo*/Val*/bcExample*
cd ~
```

(continues on next page)

(continued from previous page)

```
./dretesteth.sh -t BlockchainTests/ValidBlocks/bcExample  -- \
    --testpath ~/tests --datadir /tests/config --filltests \
    --singletest 05_simpleTx
```

3.3.1 Test Source Code

This section explains **05_simpleTxFiller.yml**. I am only going to document the things in which it is different from state transition tests.

State transition tests take their genesis block from the client configuration (or, failing that, from the default client configuration) in **retesteth**. In blockchain tests the values may be relevant to the test, so you specify them directly.

```
genesisBlockHeader:
  bloom:
↪'0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪'
  coinbase: '0x8888f1f195afa192cfee860698584c030f4c9db1'
  difficulty: '131072'
  extraData: '0x42'
  gasLimit: '3141592'
  gasUsed: '0'
  mixHash: '0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421'
  nonce: '0x0102030405060708'
  number: '0'
  parentHash: '0x0000000000000000000000000000000000000000000000000000000000000000'
  receiptTrie: '0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421'
  stateRoot: '0xf99eb1626cfa6db435c0836235942d7ccaa935f1ae247d3f1c21e495685f903a'
  timestamp: '0x54c98c81'
  transactionsTrie:
↪'0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421'
  uncleHash: '0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347'
```

In a lot of existing tests you will see a definition for **sealEngine**. This is related to getting a proof of work as part of the test. However, this is no longer part of **retesteth**, so you can omit it or set it to **NoProof**.

```
#  sealEngine: NoProof
```

Instead of a single transaction, we have a list of blocks. In a YAML list you tell different items apart by the dash character (**-**). The block list has two items in it.

```
blocks:
```

The first block has one field, **transactions**, a list of transactions. Every individual transaction is specified with the same fields used in state transition tests. This block only has one transaction, which transfers 10 Wei.

```
- transactions:
  - data: ''
    gasLimit: '50000'
    gasPrice: 20
```

This is the **nonce** value for the transaction. The first value is the **nonce** associated with the address in the **pre:** section. Each subsequent transaction from the same address increments the **nonce**.

Alternatively, if you use **auto** for every transaction of an account, the retesteth tool will provide the nonce values automatically.

```
nonce: '0'
secretKey: 45a915e4d060149eb4365960e6a7a45f334393093061116b197e3240065ff2d8
to: 0xde570000de570000de570000de570000de570000
value: '10'
```

This is the second block. In contrast to the first block, in this one we specify a **blockHeader** and override some of the default values.

```
- blockHeader:
    gasLimit: '3141592'
```

A block can also contain references to uncle blocks (blocks mined at the same time). Note that writing tests with uncle headers is complicated, because you need to run the test once to get the correct hash value. Only then can you put the correct value in the test and run it again so it'll be successful.

```
uncleHeaders: []
```

This block has two transactions.

```
transactions:
- data: ''
  gasLimit: '50000'
  gasPrice: '20'
```

This is another transaction from the same address, so the **nonce** is one more than it was in the previous one.

```
  nonce: '1'
  secretKey: 45a915e4d060149eb4365960e6a7a45f334393093061116b197e3240065ff2d8
  to: 0xde570000de570000de570000de570000de570000
  value: '20'
- data: ''
  gasLimit: '50000'
  gasPrice: '30'
```

This transaction comes from a different address (addresses are uniquely derived from the private key, and this one is different from the one in the previous transcation). This transaction's **nonce** value is the initial value for that address, zero.

```
nonce: '0'
secretKey: 41f6e321b31e72173f8ff2e292359e1862f24fba42fe6f97efaf641980eff298
to: 0xde570000de570000de570000de570000de570000
value: '30'
```

3.4 Invalid Block Tests

The invalid block test is in tests/docs/tutorial_samples/06_invalidBlockFiller.yml We copy it to **bcExample**.

```
mkdir ~/tests/src/BlockchainTestsFiller/InvalidBlocks/bcExample
cp ~/tests/docs/tutorial_samples/06_* ~/tests/src/Bl*/In*/bcExample*
cd ~
./dretesteth.sh -t BlockchainTests/InvalidBlocks/bcExample  -- \
    --testpath ~/tests --datadir /tests/config --filltests \
    --singletest 06_invalidBlock
```

Invalid block tests contain invalid blocks, blocks that cause a client to raise an exception. To tell **retesteth** which exception should be raised by a block, we add an **expectException** field to the **blockHeader**. In that field we put the different forks the test supports, and the exception we expect to be raised in them. It is a good idea to have a field that includes future forks.

```
- blockHeader:
    gasLimit: '30'
  expectException:
    Berlin: TooMuchGasUsed
    '>=London': TooMuchGasUsed
```

Warning: The **expectException** field is only used when **–filltests** is specified. When it is not, **retesteth** just expects the processing of the block to fail, without ensuring the exception is the correct one. The reason for this feature is that not all clients tells us the exact exception type when they reject a block as invalid.

3.4.1 Getting Exception Names

If you don't know what exception to expect, run the test without an **expectException**. The output will include an error message similar to this one:

```
Error: Postmine block tweak expected no exception! Client errors with:
'Error importing raw rlp block: Invalid gasUsed: header.gasUsed > header.gasLimit'
(bcBlockGasLimitTest/06_invalidBlock_Berlin, fork: Berlin, chain: default, block: 2)
```

Then took in **tests/conf/<name of client>/config** and look for the first few words of the error message. For example, in **tests/conf/tn8tool/config** we find this line:

```
"TooMuchGasUsed" : "Invalid gasUsed:",
```

This tells us that the exception to expect is **TooMuchGasUsed**.

3.5 Conclusion

You should now be able to write most types of Ethereum tests. If you still have questions, you can look in the reference section or e-mail for help.

CHAPTER 4

Testing EIPs

Ori Pomerantz

In this tutorial you learn how to write tests for a new EIP. We'll use EIP 2315 as an example. It was chosen because this EIP is already implemented by several clients, so we'll be able to run the tests we write.

4.1 Environment

These directions are written using Debian Linux 10 on Google Cloud Platform, but should work with minor changes on any other version of Linux running anywhere else with an Internet connection.

1. Install docker. You may need to reboot afterwards to get the latest kernel version.

   ```
   sudo apt install -y wget docker docker.io
   ```

2. Download the **retesteth** docker image. It is a tar file.

3. Load the docker image:

   ```
   sudo docker load -i dretest*.tar
   ```

4. Download the **dretesteth.sh** script.

   ```
   wget https://raw.githubusercontent.com/ethereum/retesteth/master/dretesteth.sh
   chmod +x dretesteth.sh
   ```

5. Download the tests:

   ```
   git clone --branch develop https://github.com/ethereum/tests.git
   ```

6. Run a test. This has two purposes:

   - Create the **retesteth** configuration directories in **~/tests/config**, where you can modify them.
   - A sanity check (that you can run tests successfully).

```
sudo ./dretesteth.sh -t GeneralStateTests/stExample -- \
 --testpath ~/tests --datadir /tests/config
```

The output should be similar to:

```
Running 1 test case...
Running tests using path: /tests
Active client configurations: 't8ntool '
Running tests for config 'Ethereum GO on StateTool'
Test Case "stExample":
100%
*** No errors detected
*** Total Tests Run: 1
```

Note: The **/tests** directory is referenced inside the docker container. It is the same as the **~/tests** directory outside it.

7. To avoid having to run with **sudo** all the time, add yourself to the **docker** group.

```
sudo usermod -a -G docker `whoami`
```

4.2 Test Cases

The EIP includes multiple tests cases, some valid, some not. You can see the first test case in docs/tutorial_samples/07_eip2315_simple_subFiller.yml. The simplest way to run these test cases is to have a separate state transition test for each one.

To run this test case, execute these commands:

```
cd ~
cp tests/docs/tutorial_samples/07* tests/src/Gen*/stExample
./dretesteth.sh -t GeneralStateTests/stExample -- \
     --singletest 07_eip2315_simple_sub --vmtrace --testpath ~/tests \
     --datadir /tests/config --filltests
```

4.2.1 The Test Source

Normally we don't want raw machine language code in tests if that can be avoided, so **retesteth** emits a warning. However, in this case we are taking the code directly from the EIP and provide the assembler equivalent in comments. This is the best we can do, at least until we have compilers that support the new opcodes, so having the raw code is justified. The **:raw** label disables the warning.

```
pre:

  # The first valid test in EIP 2315:
  # 0 PUSH1 0x04
  # 2 JUMPSUB
  # 3 STOP
  # 4 BEGINSUB
  # 5 RETURNSUB
  0x22222222222222222222222222222222222222222222:
```

(continues on next page)

(continued from previous page)

```
    balance: '0x0'
    code: :raw 0x60045e005c5d
    nonce: '0'
    storage: {}
```

To be sure that the test runs successfully, rather than appears to run successfully because of a bug in the code that runs the test, we need to be sure of what happens when a test fails. To do this, I created a second contract that always fails. It tries to run the opcode **0xFE**, which is invalid (see the EVM opcode table).

```
# An invalid test. 0xFE is not a valid opcode
0x2222222222222222222222222222222222222222FE:
  balance: '0x0'
  code: :raw 0xfe
  nonce: '0'
  storage: {}
```

This is code that runs the tests. It uses the Lisp Like Language, which is lower level than Solidity.

The curly brackets (**{**, **}**) mean to evaluate all the expressions inside them in sequence. The syntax **[[n]] <expr>** means to evaluate the expression and set storage location **n** to that value. The opcode **call** calls another contract. Taken together, this means we call the first contract (0x22...22) and store the result in location **1**. Then we call the second contract (0x22...22FE) and store the result in location **2**.

```
# Run the two tests and store the results in the account storage
0x1111111111111111111111111111111111111111:
  balance: 0
  code: |
    {
      [[1]] (call allgas 0x2222222222222222222222222222222222222222 0 0 0 0 0)
      [[2]] (call allgas 0x22222222222222222222222222222222222222FE 0 0 0 0 0)
    }
  nonce: 0
  storage: {}
```

As you can see from running the test, a successful contract call returns **1**. A failed one either does not return a value or returns **0** (in Ethereum storage zero and empty are the same thing).

```
expect:
  - indexes:
      data: !!int -1
      gas:  !!int -1
      value: !!int -1
    network:
      - '>=Berlin'
    result:
      0x1111111111111111111111111111111111111111:
        storage:
          0x01: 1
          0x02: 0  # If other words, no value
```

You can implement the second valid test case in exactly the same way.

4.2.2 Invalid Test Cases

The only difference with invalid test cases is that we store zero (or no value) instead of one. You can see one of those test cases in docs/tutorial_samples/08_eip2315_invalid_jumpFiller.yml.

```
cd ~
cp tests/docs/tutorial_samples/08* tests/src/Gen*/stExample
./dretesteth.sh -t GeneralStateTests/stExample -- \
     --singletest 08_eip2315_invalid_jump --vmtrace --testpath ~/tests \
     --datadir /tests/config --filltests
```

4.3 How Does It Work?

The explanation here is for this command:

```
./dretesteth.sh -t GeneralStateTests/stExample -- \
     --singletest 08_eip2315_invalid_jump --vmtrace --testpath ~/tests \
     --datadir /tests/config --filltests
```

However, any similar command would work the same way.

1. The **dretesteth.sh** script runs **retesteth** inside a docker container.
2. This **retesteth** sees the **–filltests** command line flag, so it knows the test is a source file that needs to be filled with additional information.

   Based on the command line, the complete file name is **tests/src/GeneralStateTestsFiller/stExample/08_eip2315_invalid_jumpF**

   - Source files are under **tests/src**.
   - The **-t GeneralStateTests/stExample** parameter means that the directory inside it is **GeneralStateTests-Filler/stExample**.
   - The **–singletest 08_eip2315_invalid_jump** parameter means that the file inside that directory is either **08_eip2315_invalid_jumpFiller.json** or **08_eip2315_invalid_jumpFiller.yml**, depending on the format.
3. The **retesteth** in the docker reads this file. You can see the format of this file here.
4. The **retesteth** in the docker container runs a client (**geth**, which is also in the docker) and receives from it additional information.
5. The filled test with the complete information is written to **tests/GeneralStateTests/stExample/08_eip2315_invalid_jump.json**. Click here to read about the format of filled tests files.
6. In the future you can run this test without **–filltests** and it would use **tests/GeneralStateTests/stExample/08_eip2315_invalid_jump.json**.

   ```
   ./dretesteth.sh -t GeneralStateTests/stExample -- \
       --singletest 08_eip2315_invalid_jump --vmtrace --testpath ~/tests \
       --datadir /tests/config
   ```

4.4 Conclusion

At this point you should know enough to test whether a client implements an EIP correctly or not, at least for EIPs that modify or add opcodes.

CHAPTER 5

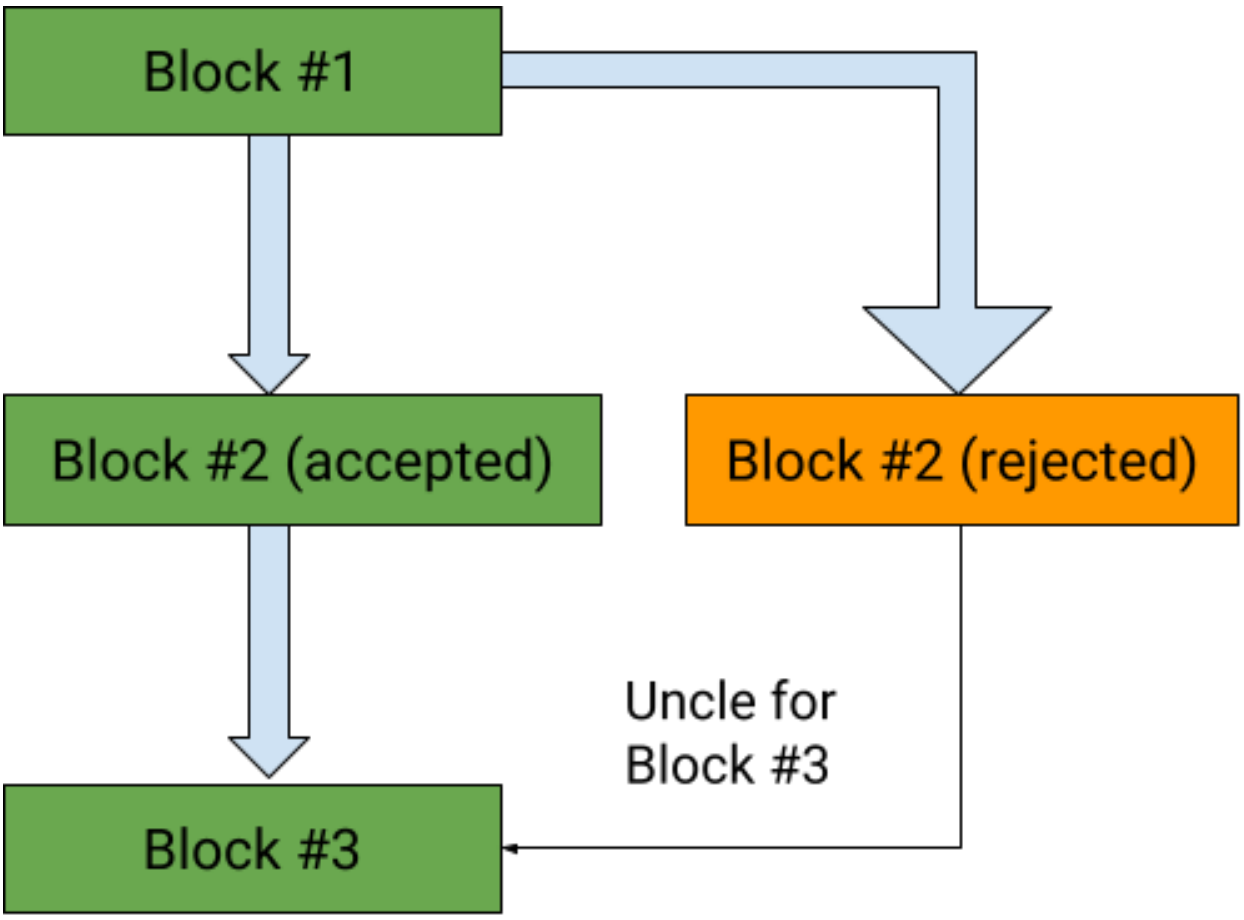
Blocktests with Ommer / Uncle Blocks

Ori Pomerantz

In this tutorial you learn hwo to write blockchain tests where the chain includes ommer/uncle blocks, blocks that are not part of the main chain but still deserve a reward.

5.1 Uncle Blocks

Uncle blocks are created because there are multiple miners working at any point in time. If two miners propose a follow-up block for the chain, only one of them becomes the real block, but the other miner who did the same work also deserves a reward (otherwise the chances of getting a mining reward will be too low, and there will be a lot less miners to keep the blockchain going).



For example, in the illustration above block #1 was proposed by a miner and accepted. Then two separate miners created followup blocks labeled as 2. One of them is the block that was eventually accepted, the green 2. The other one is the orange block that was eventually rejected. Then a miner (one of the ones above or a separate one) created block 3 and specified that the green 2 is the parent block and the orange 2 is an uncle / ommer block (ommer is a gender neutral term for an uncle or aunt). This way the miner that created the green 2 block gets the full reward (2 ETH), but the one who created the orange 2 still gets something (1.75 ETH in this case).

5.2 Writing Tests

The test writing process for ommer tests is a bit complicated. First you write a test file such as **11_ommerFiller.yml**, which has the uncle information, and run the test. However, this test always fails. The state root that is calculated by retesteth in the uncle block is different from the actual state root, because it does not include the payment to the miner of the uncle block.

5.2.1 Writing the Filler File

The only fields of the uncle block that matter are in the header, so you don't specify them in the filler file as blocks, but as a list of uncle block headers. For example, here is block 4 from 11_ommerFiller.yml.

```
- blocknumber: 4
  transactions: []
```

(continues on next page)

(continued from previous page)

```
  uncleHeaders:
    - populateFromBlock: 1
      extraData: 0x43
      coinbase: 0xCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
    - populateFromBlock: 2
      extraData: 0x43
      coinbase: 0xBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
```

As you can see, the **uncleHeaders** list contains two uncles. The first is an attempt to continue block 1, so it is an alternative block 2. The second is an attempt to continue block 2, so it is an alternative block 3.

Note: Most of the header fields are copied from the parent, except for the ones we explicitly specify (in this case, **extraData** and **coinbase**).

5.2.2 Correcting the State Root

When you run this test it is guaranteed to fail. The reason is that the state root that **retesteth** calculates is wrong, because it ignores the payments to the miners that created the uncles (in this case, 0xCC...CC and 0xBB...BB) So you can expect an error message similar to this:

```
difficulty 0x020000 vs 0x020000
extraData 0x42 vs 0x42
gasLimit 0x9c4000 vs 0x9c4000
gasUsed 0x00 vs 0x00
hash 0x9a44c40042de19b1ac61e45f9167a8b861588e63a141b2f1209fb24207f1c82a vs 0xe9c919b3737f0d5
da8846fdd1b8265285fbcb51718c2a9d280f79443bca07400
mixHash 0x0000000000000000000000000000000000000000000000000000000000000000 vs 0x000000000000
0000000000000000000000000000000000000000000000000000
nonce 0x0000000000000000 vs 0x0000000000000000
number 0x04 vs 0x04
parentHash 0xa94a42cda1934d299c86629188bd4dba3c7cecfc121319e576156e9c714ddc43 vs 0xa94a42cda
1934d299c86629188bd4dba3c7cecfc121319e576156e9c714ddc43
receiptTrie 0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421 vs 0x56e81f17
1bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421
stateRoot 0xf5f50cf42b9811e377cd17117a19961c4505fec01866aa36acfd047d7b409469 vs 0x73c5e06314
44838bfc2d8368cc4c11a15596de9b586606ebdfe8ba03c45f1039
timestamp 0x1396 vs 0x1396
transactionsTrie 0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421 vs 0x56e
81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421
uncleHash 0xabab2a7bc59ae0a16274efa337455fa1c177b60bc6d794fa188315d559894f5b vs 0xabab2a7bc5
9ae0a16274efa337455fa1c177b60bc6d794fa188315d559894f5b
baseFeePerGas 0x024c vs 0x024c
' (bcExample/11_ommer_London, fork: London, chain: default, block: 4)

--------
TestOutputHelper detected 1 errors during test execution!

*** 1 failure is detected in the test module "Master Test Suite"
```

As you can see, the field that is different is **stateRoot** (and **hash**, which is the hash of everything so if there are any differences it ends up different too). When you put that field in the block header with the correct value the test works, as you can see by running 12_ommerGoodFiller.yml.

5.3 Conclusion

The main case in which new ommer tests are needed is when the block header is modified and we need to make sure that only valid block headers are accepted, not just in the main chain but also as ommers. Here is an example of such a test. Hopefully you now know enough to write this kind of test.

CHAPTER 6

Test Internals

Ori Pomerantz

In this tutorial you learn more about the internal representation of Ethereum tests and how to run them with additional details. In theory you could write any test you want without understanding these details, but they are useful for debugging.

6.1 Compiled Tests

By default the compiled version of **tests/src/<test type>Filler/<directory>/<test>Filler** goes in **tests/<test type>/<directory><test>.json**. For example, after we copy **tests/doc/tutorial_samples/01_add22.yml** to **tests/src/GeneralStateTests/stExample/01_add22.yml** and compile it, it is available at **tests/GeneralStateTests/stExample/01_add22.json**. Here it is with explanations:

```
{
  "01_add22" : {
```

The **_info:** section includes any comments you put in the source code of the test, as well as information about the files used to generate the test (the test source code, the evm compiler if any, the client software used to fill in the data, and the tool that actually compiled the test).

```
"_info" : {
    "comment" : "You can put a comment here",
    "filling-rpc-server" : "Geth-1.9.20-unstable-54add425-20200814",
    "filling-tool-version" : "retesteth-0.0.8-docker+commit.96775cc7.Linux.g++",
    "lllcversion" : "Version: 0.5.14-develop.2020.8.15+commit.9189ad7a.Linux.g++",
    "source" : "src/GeneralStateTestsFiller/stExample/01_add22Filler.yml",
    "sourceHash" : "6b5a88627d0b69c7f61fb05f35ac3f14066d2f4bbe248aa08c3091d7534744d8"
},
```

The **env:** and **transaction:** sections contain the information provided in the source code.

```
"env" : {
    ...
    },
"transaction" : {
    ...
    },
```

The **pre:** section contains mostly information from the source file, but any code provided source (either LLL or Solidity) is compiled.

```
"pre" : {
    "0x095e7baea6a6c7c4c2dfeb977efac326af552d87" : {
        "balance" : "0x0ba1a9ce0ba1a9ce",
        "code" : "0x600260020160005500",
        "nonce" : "0x00",
        "storage" : {
        }
    },
    "0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b" : {
       ...
    }
},
```

The **post:** section is the situation after the test is run. This could be different for different versions of the Ethereum protocol, so there is a value for every version that was checked. In this case, the only one is Istanbul.

```
"post" : {
    "Istanbul" : [
        {
            "indexes" : {
                "data" : 0,
                "gas" : 0,
                "value" : 0
            },
```

Instead of keeping the entire content of the storage and logs that are expected, it is enough to just store hashes of them.

```
                "hash" :
↪"0x884b8640efb63506c2f8c2d9514335b678815e1ed362107628cf1cd6edd658c2",
                "logs" :
↪"0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347"
            }
        ]
    }
}
```

6.2 Virtual Machine Trace

If you are using the geth t8ntool, can use the **–vmtrace** command line option to get a trace of the virtual machine. For example, this is the command to get a trace of **01_add22**:

```
./dretesteth.sh -t GeneralStateTests/stExample -- --singletest 01_add22 \
   --testpath ~/tests --datadir /tests/config --filltests --vmtrace
```

6.2.1 Normal Virtual Machine Trace

This is the trace produced by the command above:

```
VMTrace: (stExample/01_add22, fork: Berlin, TrInfo: d: 0, g: 0, v: 0)
Transaction number: 0, hash:␣
↪0x4e6549e2276d1bc256b2a56ead2d9705a51a8bf54e3775fbd2e98c91fb0e4494

N    OPNAME   GASCOST  TOTALGAS REMAINGAS               ERROR
0     PUSH1         3         0  79978984
1     PUSH1         3         3  79978981
2       ADD         3         6  79978978
3     PUSH1         3         9  79978975
4    SSTORE     20000        12  79978972
      SSTORE [0x0] = 0x4
5      STOP         0     20012  79958972

{"stateRoot":"0x54d60243629f67e60925f5a9d6daf5f5ee3d774a728aa10c4ef05b8b20b1e192"}
```

6.2.2 Raw Virtual Machine Trace

The virtual machine trace above does not include the value of the program counter (PC), the content of the stack, or the full content of the storage and memory for the account. To get this information you need the raw trace:

```
./dretesteth.sh -t GeneralStateTests/stExample -- --singletest 01_add22 \
   --testpath ~/tests --datadir /tests/config --filltests --vmtraceraw | more
```

The program creates this trace:

```
VMTrace: (stExample/01_add22, fork: Istanbul, TrInfo: d: 0, g: 0, v: 0)
Transaction number: 0, hash:␣
↪0x4e6549e2276d1bc256b2a56ead2d9705a51a8bf54e3775fbd2e98c91fb0e4494
```

This is the status before the first operation. For the sake of clarity I passed it through a JSON formatter.

```
{
```

The program counter starts at zero. The opcode at that point is 96, or in hexadecimal **0x60**. Looking at the opcode table, this operation pushes a one byte value on the stack.

```
"pc":0,
"op":96,
```

The amount of gas that is currently available, and the cost of this opcode

```
"gas":"0x4c461e8",
"gasCost":"0x3",
```

Current short term (not to be stored as part of the blockchain) values: RAM, the computation stack, and the return locations stack.

```
"memory":"0x",
"memSize":0,
"stack":[

],
```

(continues on next page)

(continued from previous page)

```
"returnStack":[

],
"returnData":null,
```

The depth of the contract call. The contract called directly by the transaction is depth one. If that contract calls code in a different contract, that code will run with depth two, etc.

```
"depth":1,
```

Contracts get a refund for releasing storage they no longer need by setting it to zero). This is the amount of the refund.

```
"refund":0,
```

The name of the opcode (corresponding to the **op** value above).

```
"opName":"PUSH1",
```

The error, if any.

```
   "error":""
}
```

The second operation is almost identical to the first. The differences are:

- The program counter is two, after running an opcode with two bytes (the opcode itself and the value being pushed)
- The gas counter is lower by three (the cost of the previous operation)
- The stack, rather than empty, has a single value: **0x2**.

```
{"pc":2,"op":96,"gas":"0x4c461e5","gasCost":"0x3","memory":"0x","memSize":0,"stack":[
↪"0x2"],"returnStack":[],"returnData":null,"depth":1,"refund":0,"opName":"PUSH1",
↪"error":""}
```

Now the evm adds the two top values (turning a stack of **["0x2", "0x2"]** into **["0x4"]**) and then pushes the value zero.

```
{"pc":4,"op":1,"gas":"0x4c461e2","gasCost":"0x3","memory":"0x","memSize":0,"stack":[
↪"0x2","0x2"],"returnStack":[],"returnData":null,"depth":1,"refund":0,"opName":"ADD",
↪"error":""}
{"pc":5,"op":96,"gas":"0x4c461df","gasCost":"0x3","memory":"0x","memSize":0,"stack":[
↪"0x4"],"returnStack":[],"returnData":null,"depth":1,"refund":0,"opName":"PUSH1",
↪"error":""}
```

Now we store the value at the second place in the stack at the location in the first place. This is writing to the state, so it is an expensive operation, costing twenty thousand gas.

```
{"pc":7,"op":85,"gas":"0x4c461dc","gasCost":"0x4e20","memory":"0x","memSize":0,"stack
↪":["0x4","0x0"],"returnStack":[],"returnData":null,"depth":1,"refund":0,"opName":
↪"SSTORE","error":""}
```

Finally, stop the evm. The final line gives the output return value, the amount of gas used, and how long it took to run the program.

```
{"pc":8,"op":0,"gas":"0x4c413bc","gasCost":"0x0","memory":"0x","memSize":0,"stack":[],
↪"returnStack":[],"returnData":null,"depth":1,"refund":0,"opName":"STOP","error":""}
{"output":"","gasUsed":"0x4e2c","time":527368}
```

6.3 Conclusion

At this point you should be able to write and debug Ethereum tests.

CHAPTER 7

How to Contribute Tests

Ori Pomerantz

You've written a useful test and now you want to contribute it to the repository. This tutorial teaches you how to do it.

7.1 Generalizing Tests

Many of our tests started from the need to test a single scenario, but we figured out related scenarios that are also worth testing. For example, this test started from a request to make a CREATE2 opcode fail at a specific stage.

However, there were two ways to generalize this:

- There are multiple opcodes that are probably implemented with a number of steps, each of which may have a gas cost.
- Why fail only at one step? If we run the operation with different amounts of gas, we can probably trigger failures at each step.

While the state test transaction can only have one direct destination, we can provide whatever data we want, and that data can be used to calculate an address to call. Different addresses can contain contracts with different operations.

The easiest way to run multiple tests in a state test is to use different data values. You can use the **:abi** encoding to send multiple values. If you use **uint**, the first value will be available at **$4** (LLL) or **calldata(4)** (Yul), the second value at **$36 / calldata(0x24)**, etc.

7.2 Files

The source/filler test file is written in either YML or JSON and located under the **src** directory. This is the type of file explained in the tutorials.

In addition to the source file, your pull request needs to include the generated/filled version(s) of the test file. This version includes additional information, such as merkle tree roots of the current state, the compiled bytecode, etc.

Note: The directions below assume you are running **retesteth** through docker. See here if you are not familiar with using **retesteth** that way.

7.2.1 State Tests

State tests have their source at **src/GeneralStateTestsFiller/<directory>/<test>Filler.<yml or json>**. Every state test is supposed to have two filled versions:

1. Filled state test, which is located in **GeneralStateTests/<directory>/<test>.json**. You use a command similar to this one to create this file:

```
./dretesteth.sh -t $suite -- --testpath $dir --singletest $test --filltests
```

2. Blockchain state test, which is located in **BlockchainTests/GeneralStateTests/<directory>/<test>.json**. You use a command similar to this one to create this file:

```
./dretesteth.sh -t $suite -- --testpath $dir --singletest $test --fillchain
```

7.2.2 Blockchain Tests

State tests have their source at **src/BlockchainTestsFiller/<directory>/<test>Filler.<yml or json>**. These tests only have a single filled version, located in **BlockchainTests/<directory>/<test>.json**. You use a command similar to this one to create this file:

```
./dretesteth.sh -t $suite -- --testpath $dir --singletest $test --filltests
```

7.3 Conclusion

At this point you should know enough to submit PRs with useful tests. Go write some and amaze us.

CHAPTER 8

Using Retesteth

8.1 Command Line Options

Note: There has to be a double dash (**–**) between the **-t** option that sets the suite and all the other options.

8.1.1 Set The Suite

Option	Meaning
-t <TestSuite>	Run all the tests in that suite
-t <TestSuite>/<Test Case Folder>	Run a specific test case folder (for **GeneralStateTests**)
-t <TestSuite>/<Test Type>/<Test Case Folder>	Run a specific test case folder (for **BlockchainTests**)

Note: In **BlockchainTests** there are three possible values for the test type:

- **ValidBlocks**
- **InvalidBlocks**
- **TransitionTests**

8.1.2 Retesteth Options

Option	Meaning
-j <ThreadNumber>	Run test execution using threads
–clients *client1, client2*	Use following configurations from datadir path (default: ~/.retesteth)
–datadir	Path to configs (default: ~/.retesteth)
–nodes	List of client tcp ports (“addr:ip, addr:ip”)
–help -h	Display list of command arguments
–version -v	Display build information
–list	Display available test suites

Note: Setting **–nodes** overrides the **socketAddress** section of the **config** file, documented here.

8.1.3 Setting the Test Suite and Test

Option	Meaning
–testpath <PathToTheTestRepo>	Set path to the test repo
–testfile <TestFile>	Run tests from a file. Requires -t <TestSuite>
–singletest <TestName>	Run on a single test. *Testname* is the filename without Filler.<type> (either **json** or **yml**)
–singletest <TestName>/<Subtest>	*Subtest* is a test name inside the file.

Note: **<Subtest>** is only relevant in **BlockchainTests**. Other test suites do not support files with multiple test names.

8.1.4 Debugging

Option	Meaning
-d <index>	Set the transaction data array index when running GeneralStateTests
-g <index>	Set the transaction gas array index when running GeneralStateTests
-v <index>	Set the transaction value array index when running GeneralStateTests
–vmtraceraw	Trace transaction execution
–vmtrace	Trace transaction execution, simplified version
–limitblocks <num>	Limit the block exectuion in blockchain tests for debugging to the first <num> blocks
–limitrpc	Limit the rpc exectuion in tests for debug
–verbosity <level>	Set logs verbosity. 0 - silent, 1 - only errors, 2 - informative, >2 - detailed
–exectimelog	Output execution time for each test suite
–statediff	Trace state difference for state tests
–stderr	Redirect ipc client stderr to stdout
–travisout	Output ‘.‘ to stdout

8.1.5 Additional Tests

Option	Meaning
–all	Enable all tests
–lowcpu	Disable cpu intensive tests

8.1.6 Test Generation

Option	Meaning
–filltests	Run test fillers
–fillchain	When filling the state tests, fill tests as blockchain instead
–showhash	Show filler hash debug information
–poststate	Show post state hash or fullstate
–fullstate	Do not compress large states to hash

8.2 Examples

These examples assume you configured your enviroment the way it was shown in the tutorial and that you are in your home directory. If you used different directories, or did not use docker, the commands will be slightly different.

1. Run most state tests:

```
./dretesteth.sh -t GeneralStateTests -- --testpath ~/tests
```

Run multiple tests simultaneously:

```
./dretesteth.sh -t GeneralStateTests -- --testpath ~/tests -j 8
```

Run all the tests including the time consuming ones:

```
./dretesteth.sh -t GeneralStateTests -- --testpath ~/tests -all
```

2. Run most blockchain tests:

```
./dretesteth.sh -t BlockchainTests -- --testpath ~/tests
```

Run only the valid blocks tests:

```
./dretesteth.sh -t BlockchainTests/ValidBlocks -- --testpath ~/tests
```

Run only the invalid blocks tests:

```
./dretesteth.sh -t BlockchainTests/InvalidBlocks -- --testpath ~/tests
```

Run only a specific suite of tests:

```
./dretesteth.sh -t BlockchainTests/ValidBlocks/bcGasPricerTest \
    -- --testpath ~/tests
```

Run only the tests in a specific file (typically there would only be one):

```
./dretesteth.sh -t BlockchainTests/ValidBlocks/bcGasPricerTest \
    -- --testpath ~/tests --singletest highGasUsage
```

Run a specific test from a specific file:

```
./dretesteth.sh -t BlockchainTests/InvalidBlocks/bcForgedTest \
    -- --testpath ~/tests \
    --singletest bcBlockRLPAsList/BLOCK_difficulty_GivenAsList_Byzantium
```

3. Run transition tests (tests that verify the transition from one fork to the next is implemented correctly):

```
./dretesteth.sh -t BlockchainTests/TransitionTests -- --testpath ~/tests
```

Run the tests for a specific transition (in this case **Byzantium** to **ConstantinopleFix**):

```
./dretesteth.sh -t \
  BlockchainTests/TransitionTests/bcByzantiumToConstantinopleFix -- \
  --testpath ~/tests
```

Note: Not all transitions have associated test cases. To see which test cases are avaliable, run:

```
ls tests/BlockchainTests/TransitionTests
```

---

4. Run a test from your own file:

```
./dretesteth.sh -t GeneralStateTests -- --testpath ~/tests \
    --testfile tests/GeneralStateTests/stExample/add11.json
```

Note: In this case the test is part of the test suite and there are easier ways to run it. However, you can use **–testfile** for files that are located elsewhere. You can mount any directory inside the docker (using **–testpath**), and it will appear in the docker as **/tests**.

---

5. Fill tests. So far all of the examples have been using the generated, filled test files. However, you can also use the test source code (a.k.a. the filler version).

Fill (and run) a test that is part of the test suite (in this case, **tests/GeneralStateTests/stExample/add11**, whose source code is **tests/src/GeneralStateTestsFiller/stExample/add11Filler.json**):

```
./dretesteth.sh -t GeneralStateTests/stExample -- \
  --testpath ~/tests --singletest add11 --filltests
```

Combine this option with **–testfile** to fill and run your own tests:

```
./dretesteth.sh -t GeneralStateTests -- --testpath ~ --filltests \
    --testfile tests/tests/docs/tutorial_samples/01_add22Filler.yml
```

6. Run a test on a specific network (fork, such as **Istanbul** or **Berlin**):

```
./dretesteth.sh -t BlockchainTests/ValidBlocks/bcStateTests -- \
    --testpath ~/tests --singletest simpleSuicide --filltests  \
    --singlenet Berlin
```

Note: The generated files usually contain tests for the current fork. If you want to test a different fork, as we do here, it may be necessary to use **–filltests**.

7. Run a single test from a multitest file. The actual values come from the test file, the parameters you specify (**-d**, **-g**, and **-v**) are indexes into their respective lists (data, gas, and transaction value):

```
./dretesteth.sh -t GeneralStateTests -- --testpath ~/tests --filltests \
   --testfile /tests/docs/tutorial_samples/04_multitestFiller.yml -d 1
```

8. Run a test and produce a trace of the Ethererum Virtual Machine::

```
./dretesteth.sh -t GeneralStateTests/stExample -- \
     --testpath ~/tests --vmtrace
```

   Proudce a more detailed, but less readable, trace:

```
./dretesteth.sh -t GeneralStateTests/stExample -- \
     --testpath ~/tests --vmtraceraw
```

9. Run a test and dump the state (accounts balances, storage, etc.) at the end of it:

```
./dretesteth.sh -t GeneralStateTests/stExample -- --testpath ~/tests --poststate
```

CHAPTER 9

The Retesteth Config Directory

The retesteth **config** directory contains the **retesteth** configuration. If it is empty **retesteth** creates one with the default values. Every directory under it contains either the default configuration, or configuration for a specific client (to override the default for that client).

These directories can contain this information:

- **config** this file contains the client configuration, a JSON file with these parameters:
  - **name**, the name of the client
  - **socketType**, the type of socket used to communicate with the client. There are four supported types: **tcp**, **ipc**, **ipc-debug**, and **transition-tool**. The first three are self explanatory. The **transition-tool** "socket" is used by **t8ntool**, which runs a separate instance of **evm t8n** for each test.

    You can find more information about the communication between **retesteth** and clients in the retesteth wiki.
- **socketAddress**, the address of the socket, either a list of TCP ports (in the format **<ip>:<port>**), a file for IPC, or an executable to run (for **transition-tool**).
- **initializeTime**, the time to wait for the client to initialize before sending it tests.
- **forks**, the main supported forks.
- **additionalForks**, additional forks, which are supported but only if they are specified explicitly. For example, if a client's **config** file specifies:

```
"forks" : [
  "EIP158",
  "Byzantium",
  "Constantinople",
  "ConstantinopleFix",
  "Istanbul",
  "Berlin"
],
"additionalForks" : [
  "EIP158ToByzantiumAt5",
```

(continues on next page)

(continued from previous page)

```
  "HomesteadToDaoAt5",
  "ByzantiumToConstantinopleFixAt5"
],
```

And the test specifies **>=Byzantium**, it will test these forks:

- Byzantium
- Constantinople
- ConstantinopleFix
- Istanbul
- Berlin

But not additional forks such as **ByzantiumToConstantinopleFixAt5**.

- **exceptions**, the exception messages that the client emits for blocks that are invalid in various ways. The key is the string used to identify the exception in the **expectException** field of invalid block tests. The value is the message the client emits.

Note: The exception is only checked if:

1. **–filltests** is specified.
2. The test is in **BlockchainTests/InvalidBlocks**.

Otherwise, either **retesteth** only checks that an exception occured, not which exception it was (without **–filltests**), or treats any exception as an abort (if the test is not for invalid blocks).

- **start.sh** the meaning of this script varies depending on the method used to communicate with the client.
  - With **tcp** and **ipc** clients the script starts the client and possibly provides it with the port or pipe on which it should listen. In both cases it is possible to start multiple clients to run tests in parallel.

    Note: If there is no **start.sh** script at all **retesteth** assumes that it needs to connect to an existing client rather than run its own.

  - With **ipc-debug** clients the script is ignored, because it is assumed that the that the client is already running in debug mode.
  - With **transition-tool** clients the script is executed for every test, and the program it runs (**evm t8n** in the case of **geth**) communicates with the client to execute the test.
- **stop.sh** stop the client.
- In the case of **transition-tool** clients, this directory also contains the script that runs the client for each test. This script's name is specified in the **socketAddress** field.

  In the case of **t8ntool**, at writing the only client that uses the **transition-tool** socket type, this script is **start.sh**.
- **genesis/<forkname>.json**, this is the genesis config for the client, primarily the way to specify for the client what fork it is running. The forkname value is matched with the value for the **network:** field in the test file. This file is necessary because different clients refer to the forks by different names.

  This file may also contain an **accounts** field. This is legacy and can be ignored.
- **genesis/correctMiningReward.json**, a file that includes the mining reward for each fork.

CHAPTER 10

Transition Tool

10.1 Command Line Parameters

The command line parameters are used to specify the parameters, input files, and output files.

10.1.1 Test Parameters

- **–state.fork** *fork name*
- **–state.reward** *block mining reward* (appears only in Block tests)
- **–trace** produce a trace

10.1.2 Input Files

- **–input.alloc** *full path to pretest allocation file*
- **–input.txs** *full path to transaction file*
- **–input.env** *full path to environment file*

Note: If you want to specify any of this information in *stdin*, either omit the parameter or use the value **stdin**.

10.1.3 Output Files

- **–output.basedir** *directory to write the output files*
- **–output.result** *file name for test output*
- **–output.alloc** *file name for post test allocation file*
- **–output.body** *file name for a list of rlp transactions* (a binary file)

Note: If you want to receive this information into *stdout*, either omit the parameter or use the value **stdout**.

10.2 File Structures

All of the transition tool files are in JSON format. Any values that are not provided are assumed to be zero or empty, as applicable.

10.2.1 Transaction File

The transaction file is a list that contains maps, one for each transaction. This is an input to the tool, which *retesteth* calls *txs.json*.

Every transaction can include these fields:

- *gas*
- *gasPrice*
- *input*, the transaction data
- *nonce*
- *value*, the value in WEI sent by the transaction
- *to*, the destination. If it is not specified the transaction creates a contract

In addition, there are a few special fields that may or may not appear, as explained below.

Transaction Signatures

Transactions can be previously signed by the caller that runs the tool. In that case the transaction includes the *v*, *r*, and *s* values of the signature.

Alternatively, the transaction can include *secretKey*, in which the tool is responsible for the signature.

EIP 2930

This EIP defines a new transaction type which includes a list of addresses and storage locations. If a transaction uses EIP 2930 it would have two additional fields:

- *type* equal to one. If the transaction is normal it either has a value of zero or does not appear at all.
- *accessList*, an EIP 2930 access list.

Example

The first transaction in this list is a normal transaction, already signed. The second is an EIP 2930 transaction which needs to be signed.

```
[
   {
      "gas": "0x5208",
      "gasPrice": "0x2",
      "hash": "0x0557bacce3375c98d806609b8d5043072f0b6a8bae45ae5a67a00d3a1a18d673",
      "input": "0x",
      "nonce": "0x0",
      "r": "0x9500e8ba27d3c33ca7764e107410f44cbd8c19794bde214d694683a7aa998cdb",
      "s": "0x7235ae07e4bd6e0206d102b1f8979d6adab280466b6a82d2208ee08951f1f600",
      "to": "0x8a8eafb1cf62bfbeb1741769dae1a9dd47996192",
      "v": "0x1b",
      "value": "0x1"
   },
   {
      "gas": "0x4ef00",
      "gasPrice": "0x1",
      "chainId": "0x1",
      "input": "0x",
      "nonce": "0x0",
      "to": "0x000000000000000000000000000000000000aaaa",
      "value": "0x1",
      "type" : "0x1",
      "accessList": [
         {
            "address": "0x000000000000000000000000000000000000aaaa",
            "storageKeys": [
               "0x0000000000000000000000000000000000000000000000000000000000000000",
               "0x0000000000000000000000000000000000000000000000000000000000000012"
            ]
         },
         {
            "address": "0x000000000000000000000000000000000000aaab",
            "storageKeys": [
               "0x00000000000000000000000000000000000000000000000000000000000060A7",
               "0x0000000000000000000000000000000000000000000000000000000000000012"
            ]
         }
      ]
      "v": "0x0",
      "r": "0x0",
      "s": "0x0",
      "secretKey": "0x45a915e4d060149eb4365960e6a7a45f334393093061116b197e3240065ff2d8
↪"
   }
]
```


10.2.2 Environment File

This file is a map with the execution environment. This is an input to the tool, which *retesteth* calls *env.json*. It has these fields:

- *currentCoinbase*
- *currentDifficulty*
- *currentGasLimit*
- *currentNumber*

- *currentTimestamp*
- *previousHash*, the hash of the previous (*currentNumber*-1) block
- *blockHashes*, a map of historical block numbers and their hashes

Note: Some tests include multiple blocks. In that case, the test software runs *t8ntool* multiple times, one per block.

Example

```
{
   "currentCoinbase" : "0x2adc25665018aa1fe0e6bc666dac8fc2697ff9ba",
   "currentDifficulty" : "0x020000",
   "currentGasLimit" : "0x05f5e100",
   "currentNumber" : "0x01",
   "currentTimestamp" : "0x03e8",
   "previousHash" :
↪"0xe729de3fec21e30bea3d56adb01ed14bc107273c2775f9355afb10f594a10d9e",
   "blockHashes" : {
       "0" : "0xe729de3fec21e30bea3d56adb01ed14bc107273c2775f9355afb10f594a10d9e"
   }
}
```

10.2.3 Allocation Files

These files show the state of various accounts and contracts on the blockchain. In *retesteth* there are two of these files: *alloc.json* which is the input state and *outAlloc.json* which is the output state.

The file is a map of *address* values to account information. The account information that can be provided is:

- *balance*
- *code* (in machine language format)
- *nonce*
- *storage*

Example

```
{
    "a94f5374fce5edbc8e2a8697c15331677e6ebf0b": {
        "balance": "0x5ffd4878be161d74",
        "code": "0x5854505854",
        "nonce": "0xac",
        "storage": {
           "0x0000000000000000000000000000000000000000000000000000000000000000":
           "0x0000000000000000000000000000000000000000000000000000000000000004"
        }
     },
     "0x8a8eafb1cf62bfbeb1741769dae1a9dd47996192":{
        "balance": "0xfeedbead",
        "nonce" : "0x00"
     }
}
```

10.2.4 Result File

In *retesteth* this file is called *out.json*. It is the post state after processing the block. It should include the following fields:

- *stateRoot*
- *txRoot*
- *receiptRoot*
- *logsHash*
- *logsBloom*, the bloom filter for the logs.
- *receipts*, a list of maps, one for each transaction, with the transaction receipt. Each of those receipts includes these fields:
  - *root*
  - *status*
  - *cumulativeGasUsed*
  - *logsBloom*
  - *logs*
  - *transactionHash*
  - *contractAddress*, the address of the created contract, if any
  - *gasUsed*
  - *blockHash*, all zeros because this is created before the block is finished
  - *transactionIndex*

Example

```
{
  "stateRoot": "0x1c99b01120e7a2fa1301b3505f20100e72362e5ac3f96854420e56ba8984d716",
  "txRoot": "0xb5eee60b45801179cbde3781b9a5dee9b3111554618c9cda3d6f7e351fd41e0b",
  "receiptRoot": "0x86ceb80cb6bef8fe4ac0f1c99409f67cb2554c4432f374e399b94884eb3e6562",
  "logsHash": "0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347",
  "logsBloom":
↪"0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
  "receipts": [
     {
         "root": "0x",
         "status": "0x1",
         "cumulativeGasUsed": "0xa878",
         "logsBloom":
↪"0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
         "logs": null,
         "transactionHash":
↪"0x4e6549e2276d1bc256b2a56ead2d9705a51a8bf54e3775fbd2e98c91fb0e4494",
         "contractAddress": "0x0000000000000000000000000000000000000000",
         "gasUsed": "0xa878",
         "blockHash":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
```

(continues on next page)

(continued from previous page)

```
            "transactionIndex": "0x0"
        }
    ]
}
```

10.2.5 Trace Files

If **–trace** is specified, the t8ntool creates a file (or files) called *trace-<transaction number>-<transaction hash>.jsonl*. The format of this file is specified in EIP 3155.

If the transaction fails and does not produce a hash, the name of the file is still *trace-<transaction number>-<value that is a legitimate hash>.jsonl*.

10.3 Using Standard Input and Output

It should also be possible to run a *t8ntool* with input coming from *stdin* and output going to *stdout*. In this case, the input is all one object and the output is all one object.

10.3.1 Input

When the input is provided using *stdin*, it can have any combination of these three fields (whichever ones aren't provided in file form)

- *txs*, a list of transactions
- *alloc*, a map of the pretest accounts
- *env*, a map of the execution environment

10.3.2 Output

When the output goes to *stdout*, it can have any combination of these fields (whichever ones don't have a specified output file):

- *result*, the poststage
- *body*, the transactions and their results

CHAPTER 11

The RPC Interface

Some clients, such as besu, run tests using this interface. This allows the client to run anywhere there is connectivity to the system running **retesteth**.

In addition to requiring some of the standard Ethereum RPC function, **retesteth** requires some specific functions to setup and execute tests.

11.1 Retesteth-Specific RPCs

11.1.1 debug_accountRange

Get a list of accounts at a certain point in time.

Parameters

1. **string _blockHashOrNumber**: The hash or number of the block
2. **int _txIndex**: Transaction index for the point in which we want the list of accounts
3. **string _addressHash**: The hash at which to start the list If **_maxResults** is equal to the the number of accounts or more than that then we receive all the addresses and there is no problem. But if there are too many accounts to report all them, we receive the next hash at which we can find an address. We then call this method again, with that value in **_addressHash**, to get the next batch of addresses.
4. **int _maxResults**: Maximum number of results

Result

- **addressMap**: An object with hash values and the addresses they represent. We use the hashes (both here and in the **_addressHash** parameter) because that is the order in which addresses are stored in the client, so the easiest order to for paged retrieval.

- **nextKey**: The next hash (in case there are more addresses to return than **_maxResults**.

Sample Request

```
{
  "jsonrpc": "2.0",
  "method": "debug_accountRange",
  "params": [
    "1",
    1,
    "0x0000000000000000000000000000000000000000000000000000000000000001",
    10
  ],
  "id": 9
}
```

This request came from a state transition test, which means that there is only one block and within it only one transaction.

1. **_blockHashOrNumber**: The block number, which is one (the only block there is)
2. **_txIndex**: We want the state after one transaction (the only transaction in the block)
3. **_addressHash**: This is the first request, so we want to start at the beginning.
4. **_maxResults**: We want up to ten results.

Sample Result

```
{
  "jsonrpc": "2.0",
  "id": 9,
  "result": {
    "addressMap": {
      "0x03601462093b5945d1676df093446790fd31b20e7b12a2e8e5e09d068109616b":
↪"0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b",
      "0x0fbc62ba90dec43ec1d6016f9dd39dc324e967f2a3459a78281d1f4b2ba962a6":
↪"0x095e7baea6a6c7c4c2dfeb977efac326af552d87",
      "0x9d860e7bb7e6b09b87ab7406933ef2980c19d7d0192d8939cf6dc6908a03305f":
↪"0x2adc25665018aa1fe0e6bc666dac8fc2697ff9ba"
    },
    "nextKey": "0x0000000000000000000000000000000000000000000000000000000000000000"
  }
}
```

- **addressMap**: Three entries, there are three addresses with meaningful information.
- **nextKey**: Zero, because there are no more results to return.

11.1.2 debug_storageRangeAt

Get a list of storage values.

Parameters

1. **string _blockHashOrNumber**: The hash or number of the block
2. **int _txIndex**: Transaction index for the point in which we want the list of accounts
3. **string _address**: Read storage values for this address.
4. **string _begin**: Start from this hash
5. **int _maxResults**: Maximum number of results

Result

- **storage**: An object with hash values, and for each of them the key and value it represents.
- **complete**: Boolean value, true if this completes the storage entries.

Sample Request

```
{
  "jsonrpc": "2.0",
  "method": "debug_storageRangeAt",
  "params": [
    "1",
    1,
    "0x095e7baea6a6c7c4c2dfeb977efac326af552d87",
    "0x0000000000000000000000000000000000000000000000000000000000000000",
    20
  ],
  "id": 17
}
```

1. **string _blockHashOrNumber**: One, the only valid value for a state test
2. **int _txIndex**: One, the only valid value for a state test
3. **string _address**: An address
4. **string _begin**: Start from the beginning, zero
5. **int _maxResults**: Read up to twenty results

Sample Result

```
{
  "jsonrpc": "2.0",
  "id": 17,
  "result": {
    "storage": {
      "0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563": {
        "key": "0x00",
        "value": "0x02"
      },
      "0x8a8c65155279fdd366bbe4502fff15c2162ef3f469afd7533efe047403a26923" : {
        "key" : "0x60a7",
        "value" : "0x60a7"
```

(continues on next page)

(continued from previous page)

```
        }
      },
      "complete": true
    }
}
```

- **storage**: An object with two hash values, each of which has the key and value that it represents.
- **complete**: True, this is the entire storage.

11.1.3 debug_traceTransaction

Get the virtual machine trace of a transaction. Not currently implemented.

11.1.4 test_mineBlocks

Put the existing valid transactions into the current block and finish it, and create a number of blocks after it.

Parameters

1. **int _number**: The number of blocks to create after the current block.

Result

Boolean value, **true** if successful

Sample Request

```
{
  "jsonrpc": "2.0",
  "method": "test_mineBlocks",
  "params": [
    1
  ],
  "id": 28
}
```

Create one additional block

Sample Result

```
{
  "jsonrpc": "2.0",
  "id": 28,
  "result": true
}
```

Success

11.1.5 test_modifyTimestamp

Parameters

1. **int _timestamp**: The new timestamp

Result

Boolean value, **true** if successful

Sample Request

```
{
  "jsonrpc": "2.0",
  "method": "test_modifyTimestamp",
  "params": [
    1000
  ],
  "id": 2
}
```

Change the timestamp to 1000. This value is a Unix timetamp, 1000 second after midnight on January 1st, 1970, GMT.

Sample Result

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": true
}
```

Success

11.1.6 test_rewindToBlock

Revert the state of the blockchain to a specific block number. Cancel the blocks after it, which lets us run multiple tests without having to setup a new genesis block for each one.

Parameters

1. **int _block**: The number of the last block that is not cancelled. If it is the genesis block, this value is zero.

Result

Boolean value, **true** if successful

Sample Request

```
{
  "jsonrpc": "2.0",
  "method": "test_rewindToBlock",
  "params": [
    0
  ],
  "id": 22
}
```

Rewind all the way to the genesis block.

Sample Result

```
{
  "jsonrpc": "2.0",
  "id": 22,
  "result": true
}
```

Success

11.1.7 test_setChainParams

This method tells a client to initialize a test chain to a given state.

Parameters

An object that contains the chain parameters for the test:

- **params**: Chain parameters: - **chainID**: The chain identifier. - **<fork>ForkBlock**: The block in which that fork starts on this chain.
- **accounts**: The accounts at the test's start. This is an object whose keys are the addresses of the accounts. For each account there are these parameters (all the scalar values are strings with a hexadecimal number in them): - **balance**: Balance in wei - **code**: The EVM code (**0x** if there is none). - **nonce**: The nonce for the next transaction from this address. - **storage**: An object with keys and their values.
- **sealEngine**: Currently always **NoReward**.
- **genesis**: The parameters of the genesis block.

Result

Boolean value, **true** if successful

Sample Request

```
{
  "jsonrpc": "2.0",
  "method": "test_setChainParams",
  "params": [
    {
      "params": {
        "homesteadForkBlock": "0x00",
        "EIP150ForkBlock": "0x00",
        "EIP158ForkBlock": "0x00",
        "byzantiumForkBlock": "0x00",
        "constantinopleForkBlock": "0x00",
        "constantinopleFixForkBlock": "0x00",
        "istanbulForkBlock": "0x00",
        "berlinForkBlock": "0x00",
        "chainID": "0x01"
      },
      "accounts": {
        "0x095e7baea6a6c7c4c2dfeb977efac326af552d87": {
          "balance": "0x0de0b6b3a7640000",
          "code": "0x600160010160005500",
          "nonce": "0x00",
          "storage": {}
        },
        "0x2adc25665018aa1fe0e6bc666dac8fc2697ff9ba": {
          "balance": "0x00",
          "code": "0x",
          "nonce": "0x01",
          "storage": {}
        },
        "0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b": {
          "balance": "0x0de0b6b3a7640000",
          "code": "0x",
          "nonce": "0x00",
          "storage": {}
        }
      },
      "sealEngine": "NoReward",
      "genesis": {
        "author": "0x2adc25665018aa1fe0e6bc666dac8fc2697ff9ba",
        "difficulty": "0x020000",
        "gasLimit": "0xff112233445566",
        "extraData": "0x00",
        "timestamp": "0x00",
        "nonce": "0x0000000000000000",
        "mixHash": "0x0000000000000000000000000000000000000000000000000000000000000000
↪"
      }
    }
  ],
  "id": 1
}
```

Sample Result

```
{
  "jsonrpc": "2.0",
```

(continues on next page)

(continued from previous page)

```
  "id": 1,
  "result": true
}
```

Success

11.2 Standard RPCs Retesteth Uses

- eth_blockNumber
- eth_getBalance
- eth_getBlockByNumber
- eth_getCode
- eth_getTransactionCount
- eth_sendRawTransaction
- web3_clientVersion

CHAPTER 12

Blockchain Tests

12.1 Blockchain Tests Source Code

Location: /src/BlockchainTestsFiller

Blockchain tests can include multiple blocks and each of those blocks can include multiple transactions. These blocks can be either valid or invalid.

12.1.1 Subfolders

InvalidBlocks	Tests containing blocks that are expected to fail on import
ValidBlocks	Normal blockchain tests
TransitionTests	Blockchain tests with exotic network rules switching forks at block #5

Note: Except for the values in the **indexes** section, all the field values in the tests' source code are strings. When a value is numeric, such as a value in storage or an address's balance, it can be specified either in hexademical (starting with **0x**), or in decimal (starting with a digit). For the sake of legibility, numeric values can also have underscores. For example, you can use **1_000_000_000** for 10^9, which is a lot more readable than **1000000000**.

In YAML string is the default field type. In JSON string values are enclosed by quotes.

12.1.2 Test Structure

You can write tests either in JSON format or in YAML format. All tests are inside a structure with their name

Format

Format	JSON	YAML
Filename	name-of-testFiller.json	name-of-testFiller.yml

Format	<pre>{
	  "name-of-test": {
	  Sections go here
	  }
	}</pre>
		<pre>name-of-test:
		  Sections go here</pre>

12.1.3 Genesis Block

This section contains the genesis block that starts the chain being tested.

Format

JSON

```
{
   "name-of-test": {
      <other sections>,
      genesisBlockHeader: {
         "bloom" : "0x0 <<lots more 0s>>"
         "coinbase" :
↪"0x8888f1f195afa192cfee860698584c030f4c9d
↪",
         "difficulty" : "0x020000",
         "extraData" : "0x42",
         "gasLimit" : "0x2fefd8",
         "gasUsed" : "0x00",
         "mixHash" :
↪"0x0000000000000000000000000000000000000000
↪",
         "nonce" : "0x0000000000000000",
         "number" : "0x00",
         "parentHash" :
↪"0x0000000000000000000000000000000000000000
↪",
         "receiptTrie" :
↪"0x56e81f171bcc55a6ff8345e692c0f86e5b48e0
↪",
         "stateRoot" :
↪"0x14f0692d8daa55f0eb56a1cf1e2b07746d66dd
↪",
         "timestamp" : "0x54c98c81",
         "transactionsTrie" :
↪"0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421
↪",
         "uncleHash" :
↪"0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
↪"
         "baseFee" : "1000"
      }
   }
}
```

YAML

```
name-of-test:
   <other sections>
   genesisBlockHeader:
      bloom: 0x0 <<lots more 0s>>
      coinbase:␣
↪0x8888f1f195afa192cfee860698584c030f4c9db1
      difficulty: 131072
      extraData: 0x42
      gasLimit: 3141592
      gasUsed: 0
      mixHash:␣
↪0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc00162
      nonce: 0x0102030405060708
      number: 0
      parentHash:␣
↪0x0000000000000000000000000000000000000000000000000000000000000000
      receiptTrie:␣
↪0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc00162
      stateRoot:␣
↪0xf99eb1626cfa6db435c0836235942d7ccaa935f1ae247d3f1c21
      timestamp: 0x54c98c81
      transactionsTrie:␣
↪0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc00162
      uncleHash:␣
↪0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a1
      baseFee: 1000
```

Fields

Name in Block Header Sections	Meaning
bloom	bloom filter to speed searches
coinbase	beneficiary of mining fee
extraData	data added to the block, ignored by **retesteth**
difficulty	difficulty of previous block
gasLimit	limit of gas usage per block
gasUsed	gas used by this block
mixHash and nonce	used by the proof of work algorithm, ignored by **retesteth**
number	number of ancestor blocks
parentHash	hash of previous block
receiptTrie	The root of the receipt trie after this block
stateRoot	The root of the state trie after this block
timestamp	Unix time
transactionTrie	The root of the transaction trie after this block
uncleHash	hash of uncle block or blocks
baseFee	The base fee per gas required of transactions (London and later, because of EIP 1559)

You can read more about the block header fields here.

12.1.4 Pre

This section contains the initial information of the blockchain.

Format

JSON

```
{
   "name-of-test": {
      <other sections>,
      "pre": {
         "address 1": {
             "balance": "0xba1a9ce000",
             "nonce": "0",
             "code": ":raw␣
↪0x600160010160005500"
             "storage: {
                     "0":       "12345",
                     "0x12": "0x121212"
         },
         "address 2": {
             <address fields go here>
         }
      }
   }
}
```

YAML

```
name-of-test:
   <other sections>
   pre:
     address 1:
       balance: 0xba1a9ce000,
       nonce: 0,
       code: :raw 0x600160010160005500
       storage:
         0:       12345
         0x12: 0x121212
     address 2:
       <address fields go here>
```

Address Fields

- **balance**:

  Wei balance at the start of the test

- **code**:

  The code of the contract. In the **expect:** section this has to be raw virtual machine code.

- **nonce**:

  The nonce counter for the address. This value is used to make sure each transaction is processed only once. The first transaction the address sends has a nonce equal to this value, the second one is the nonce plus one, etc.

- **storage**:

  Values in the storage of the address

JSON	YAML
`storage: { "1": 5, "0x20": 0x10 }`	`storage: 1: 5 0x20: 0x10`

- **code**:

  The code of the contract. There are several possibilities:

  - If the account is not a contract, this value is **0x**
  - Raw virtual machine code. This is for cases where it is impossible to provide source code, or the source code is in a language retesteth does not recognize, such as Vyper.

    ```
    :raw 0x600160010160005500
    ```

  - Lisp Like Language (lll), for example:

    ```
    {
       ; Add 2+2 and store the value in storage location 0
       [[0]] (ADD 2 2)
    }
    ```

  - Yul, which is documented here, for example:

    ```
    :yul {
       // Add 2+2 and store the value in storage location 0
       sstore(0, add(2,2))
    }
    ```

  - Solidity, which you can learn here. Solidity code can be provided to a test in two ways:
    - Put the solidity code itself in the contract definition (same place as the LLL or Yul code).
    - Put a **:solidity** section with the contract source code. In that case, the value in **code:** is **:solidity <name of contract>**.

12.1.5 Blocks

This section contains the blocks of the blockchain that are supposed to modify the state from the one in the **pre** section to the one in the **expect** section.

Format

JSON

```
{
   "name-of-test": {
      <other sections>,
      blocks: [
        { transactions: [
            { <transaction> },
            { <transaction> }
          ]
        },
        { transactions: [
            { <transaction> },
            { <transaction> }
          ]
          blockHeader: {
             "extraData" : "0x42",
             "gasLimit" : "0x2fefd8",
             "gasUsed" : "0x5208",
          },
          uncleHeaders: [ <values here> ]
        }
      ]
   }
}
```

YAML

```
name-of-test:
   <other sections>
   blocks:
   - transactions:
     - <transaction>
     - <transaction>
   - blockHeader:
        extraData: 42
        gasLimit: 100_000
        gasUsed: 2_000
     uncleHeaders:
       <values here>
     transactions:
     - <transaction>
     - <transaction>
```

Fields

The fields in each block are optional. Only include those fields you need.

- **blockHeader**:

  This field contains the block header parameters. Parameters that are missing are copied from the genesis block.

Name in Block Header Sections	Meaning
bloom	bloom filter to speed searches
coinbase	beneficiary of mining fee
extraData	data added to the block, ignored by **retesteth**
difficulty	difficulty of previous block
gasLimit	limit of gas usage per block
gasUsed	gas used by this block
mixHash and nonce	used by the proof of work algorithm, ignored by **retesteth**
number	number of ancestor blocks
parentHash	hash of previous block
receiptTrie	The root of the receipt trie after this block
stateRoot	The root of the state trie after this block
timestamp	Unix time
transactionTrie	The root of the transaction trie after this block
uncleHash	hash of uncle block or blocks
baseFee	The base fee per gas required of transactions (London and later, because of EIP 1559)

You can read more about the block header fields here.

One field inside the block header which is not standard in Ethereum is **expectException**. That field, which is only used in invalid block tests, identifies the exception we expect to receive for the block on different forks of Ethereum. You can read more about it in the Invalid Block Tests section of the Blockchain Tests tutorial.

Note that starting with London **gasLimit** cannot be changed by more than 1/1024 from the previous value because of EIP 1559. You can specify **baseFee**, but the block is only valid if it is the same value that was calculated from the previous block.

- **blocknumber** and **chainname**:

  If you are testing behavior in the presence of multiple competing chains, these fields let you specify the chain and the block's location within it.

- **uncleHeaders**:

  A list of the uncle blocks (blocks mined at the same time). Each item in the list has two fields:

  - **chainname**: The name of the chain from which the uncle block comes
  - **populateFromBlock**: The block number within that chain for the block that is an uncle of the block you are specifying.

  However, if you write a test with uncles, you need to run it twice, once to get the state hash values to write them in the test filler file, and again to actually run the test.

- **transactions**:

  A list of transaction objects in the block.

12.1.6 Transaction

This is the data of a transaction. Every block contains a list of transactions

Format

JSON

```
{
    "name-of-test":
    {
       <other sections>
       "blocks": [
          {
            transactions: [
              {
                data: "0xDA7A",
                gasLimit: "0x6a506a50",
                gasPrice: 1,
                value: 1,
                to:
↪"add13ess01233210add13ess01233210",
                secretKey: "5ec13e7 ...␣
↪5ec13e7"
                nonce: '0x909ce'
              },
              {
                data: "0xDA7A",
                accessList: [
                  {
                     "address":
↪"0xccccccccccccccccccccccccccccccccccccccc
↪",
                     "storageKeys": [
↪"0x1000", "0x60A7"]
                  },
                  {
                     "address":
↪"0xccccccccccccccccccccccccccccccccccccccc
↪",
                     "storageKeys": []
                  }
                ],
                gasLimit: "0x6a506a50",
                maxFeePerGas: 1000,
                maxPriorityFeePerGas: 10,
                value: 1,
                to:
↪"add13ess01233210add13ess01233210",
                secretKey: "5ec13e7 ...␣
↪5ec13e7"
                nonce: '0x909ce'
              },
              <other transactions>
            ]
            <other block fields>
          },
          <other blocks>
       ]
   }
```

YAML

```
<test-name>:
  <other sections>
  blocks:
  - transactions:
    - data: 0xDA7A
      gasLimit: '0x6a506a50'
      maxFeePerGas: 1000
      maxPriorityFeePerGas: 10
      value: 1
      to:
↪"add13ess01233210add13ess01233210"
      secretKey: "5ec13e7 ... 5ec13e7"
      nonce: '0x909ce'
    - data: 0xDA7A
      accessList:
      - address:␣
↪0xccccccccccccccccccccccccccccccccccccccccd
        storageKeys:
        - 0x1000
      - address:␣
↪0xccccccccccccccccccccccccccccccccccccccccc
        storageKeys: []
      gasLimit: '0x6a506a50'
      gasPrice: "1"
      value: 1
      to:
↪"add13ess01233210add13ess01233210"
      secretKey: "5ec13e7 ... 5ec13e7"
      nonce: '0x909ce'
    - <another transaction>
    <other block fields>
  - <another block>
```

Fields

- **data**:

  The data, either in hexadecimal or an ABI call with this format: **:abi <function signature> <function parameters separated by spaces>**.

- **accessList**:

  An optional EIP2930 access list. The **accessList** is a list of structures, each of which has to have an **address** and a list of **storageKeys** (which may be empty).

- **gasLimit**:

  Gas limit for the transaction

- **gasPrice**:

  Gas price in Wei, prior to London (changed by EIP 1559).

- **maxFeePerGas**:

  Maximum acceptable gas price in Wei. Available in London and later.

- **maxPriorityFeePerGas**:

  Tip to give the miner (per gas, in Wei). The real tip is either this value or **maxFeePerGas-baseFee** (the lower of the two). Available in London and later.

- **value**:

  The value the transaction transmits in Wei

- **to**:

  The destination address, typically a contract

- **secretKey**:

  The secret key for the sending address. That address is derived from the secret key and therefore does not need to be specified explicitely (see here).

- **nonce**:

  The nonce value for the transaction. The first transaction for an address has the nonce value of the address itself, the second transaction has the nonce plus one, etc. Alternatively, if you replace all the **nonce** values with **auto**, the tool does this for you.

- **invalid**:

  If the transaction is invalid, meaning clients should reject it, set this value to "1"

- **expectException**:

  This field specifies the exception we expect to see raised by for this transaction. It is optional - you only add it if an exception is expected.

JSON	YAML

`"expectException": {`
`  ">=London": TR_TipGtFeeCap`
`}`
	`expectException:`
	`  ">=London": TR_TipGtFeeCap`

The fields are fork specifications:

Type of specification	Example
Specific fork	Berlin
A specific fork and all forks after it	>=London
Anything prior to a specific fork (not including that fork)	<Berlin

The value is an exception name. You can see the list in the retesteth code.

12.1.7 Expect

This section contains the information we expect to see after the test is concluded.

Format

JSON

```
{
   "name-of-test": {
      <other sections>,
      "expect": [
        {
           "network": ["Istanbul",
↪<other forks, see below>],
           "result": {
                "address 1": {
                    "balance":
↪"0xba1a9ce000",
                    "nonce": "0",
                    "storage: {
                        "0x0":     12345,
                        "10" : "0x121212"
                    },
                    "code": "0x00"
                },
                "address 2": {
                    <address fields go␣
↪here>
                }
        },
        { <forks & results> }
      ]
   }
}
```

YAML

```
name-of-test:
   <other sections>
   expect:
   - network:
     - Istanbul
     - <another fork>
     result:
       address 1:
         balance: 0xba1a9ce000,
         nonce: 0,
         storage:
           0x0:  12345
           10: 0x121212
         code: 0x00
       address 2:
          <address fields go here>
   - <forks & results>
```

The Network Specification

The string that identifies a fork (version) within a **network:** list is one of three option:

- The specific version: **Istanbul**
- The version or anything later: **>=Frontier**

- Anything up to (but not including) the version **<Constantinople**

Address Fields

It is not necessary to include all fields for every address. Only include those fields you wish to test.

- **balance**:

  Wei balance at the start of the test

- **code**:

  The code of the contract. In the **expect:** section this has to be raw virtual machine code.

- **nonce**:

  The nonce counter for the address. This value is used to make sure each transaction is processed only once. The first transaction the address sends has a nonce equal to this value, the second one is the nonce plus one, etc.

- **storage**:

  Values in the storage of the address

JSON	YAML

`storage: {`
`  "1": 5,`
`  "0x20": 0x10`
`}`
	`storage:`
	`  1: 5`
	`  0x20: 0x10`

12.2 Generated Blockchain Tests

Location /BlockchainTests

Subfolders

GeneralStateTests	Tests generated in blockchain form from GeneralStateTests
InvalidBlocks	Tests containing blocks that are expected to fail on import
ValidBlocks	Normal blockchain tests
TransitionTests	BC tests with exotic network rules switching forks at block#5

12.2.1 Test Structure

Contains **blocks** that are to be imported on top of **genesisRLP** of network fork rules **network** using sealEngine **NoProof** (Ethash no longer supported) and having genesis state as **pre**.

The result of block import must be state **postState** or **postStateHash** if result state is too big. And the last block of chain with maxTotalDifficulty must be block with hash **lastblockhash**

Single blockchain test file might contain many tests as there are many test generations for each individual **network** fork rules from single test source.

```
{
   "testname": {
     "_info" : { ... },
     "sealEngine": [ "NoProof" | "Ethash" ]
     "network": "Byzantium",
     "pre": { ... },
     "genesisBlockHeader": { ... },
     "genesisRLP": " ... ",
     "blocks" : [ ... ],
     "postState": { ... },
     "lastblockhash": " ... "
   },
   "testname": {
     "_info" : { ... },
     "sealEngine": [ "NoProof" | "Ethash" ]
     "network": "Byzantium",
     "pre": { ... },
     "genesisBlockHeader": { ... },
     "genesisRLP": " ... ",
     "blocks" : [ ... ],
     "postStateHash": " ... ",
     "lastblockhash": " ... "
   }
   ...
}
```

12.2.2 Info Section

```
"_info" : {
         "comment" : "A test for (add 1 1) opcode result",
         "filling-rpc-server" : "Geth-1.9.14-unstable-8cf83419-20200512",
         "filling-tool-version" : "retesteth-0.0.3+commit.672a84dd.Linux.g++",
         "lllcversion" : "Version: 0.5.14-develop.2019.11.27+commit.8f259595.Linux.g++
↪",
         "source" : "src/GeneralStateTestsFiller/stExample/add11Filler.json",
         "sourceHash" :
↪"e474fc13b1ea4c60efe2ba925dd48d6f9c1b12317dcd631f5eeeb3722a790a37"
 },
```

Info section is generated with the test and contains information about test and it's generators.

Fields

`comment`	comment from the test source. (can be edited at source)
`filling-rpc-server`	tool that has been used to generate the test (version)
`filling-tool-version`	the test generator (retesteth) version
`lllcversion`	lllc version that was used to compile LLL code in test fillers
`source`	path to the source filler in the test repository
`sourceHash`	hash of the json of source file (used to track that tests are updated)

12.2.3 Pre/preState Section

```
"pre" : {
    "0x095e7baea6a6c7c4c2dfeb977efac326af552d87" : {
        "balance" : "0x0de0b6b3a7640000",
        "code" : "0x600160010160005500",
        "nonce" : "0x00",
        "storage" : {
            "0x00" : "0x01"
        }
    },
    "0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b" : {
        "balance" : "0x0de0b6b3a7640000",
        "code" : "0x",
        "nonce" : "0x00",
        "storage" : {
        }
    }
},
```

Pre section describes a full state of accounts used in test.

Its a map of <Account> => <AccountFields>

AccountFields are always complete (*balance*, *code*, *nonce*, *storage* must present) in this section and can not have a missing field.

- All values are 0x prefixed hex.
- Empty code defined as 0x.
- Zero storage record defined as 0x00.

Fields

`address hash`	**HASH20**	is 20 bytes ethereum address 0x prefixed
`balance`	**VALUE**	account balance in evm state
`code`	**BYTES**	account code in evm state
`nonce`	**VALUE**	account nonce in evm state
`storage`	**map**	map of storage records **VALUE => VALUE**

TYPE	Empty	Length	Format description
`VALUE`	0x00	Any*	0x prefixed hex up to 32 bytes long with no leading zeros.
`BYTES`	0x	Any*	0x prefixed bytes of any length
`HASH8`	0x00...00	Fixed 8	0x prefixed bytes of length 8
`HASH20`	0x00...00	Fixed 20	0x prefixed bytes of length 20
`HASH32`	0x00...00	Fixed 32	0x prefixed bytes of length 32
`HASH256`	0x00...00	Fixed 256	0x prefixed bytes of length 256

- Size can be limited by the meaning of field in tests. (like gasLimit ceil, tx signature v - value)

12.2.4 GenesisBlockHeader Section

```
"genesisRLP" :
↪"0xf901fcf901f7a00000000000000000000000000000000000000000000000000000000000000000a01dcc4de8dec75d7a
↪",
"genesisBlockHeader" :
{
    "bloom" :
↪"0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
    "coinbase" : "0x2adc25665018aa1fe0e6bc666dac8fc2697ff9ba",
    "difficulty" : "0x020000",
    "extraData" : "0x42",
    "gasLimit" : "0x7fffffffffffffff",
    "gasUsed" : "0x00",
    "hash" : "0xd1b5003dbed66eb89c9b0062798a4fddf737157ef6550187d098d91ae2c3b853",
    "mixHash" : "0x0000000000000000000000000000000000000000000000000000000000000000",
    "nonce" : "0x0000000000000000",
    "number" : "0x00",
    "parentHash" : "0x0000000000000000000000000000000000000000000000000000000000000000
↪",
    "receiptTrie" :
↪"0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421",
    "stateRoot" : "0x423c3994e5db8348336dd13c498b840988ea7f716a0b876af847c6b8a3446e43
↪",
    "timestamp" : "0x03b6",
    "transactionsTrie" :
↪"0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421",
    "uncleHash" : "0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347"
}
```

Genesis blockheader is basically the block header struct of the first block. It is constructed from test_setChainParams or config files and specify the genesis block header fields. Genesis block hash is later used in the tests. The key field here is *genesisRLP*, it contains the block information. *genesisBlockHeader* exist merely for humans to read the genesisRLP content. The content of *genesisRLP* **MUST** be equal to the one in *genesisBlockHeader* and is actually used in testers.

- all fields are 0x prefixed HEX
- 0 values are 0x00 rounded
- empty extraData 0x

Fields

Fields are similar to the standard ethereum block header.

12.2.5 Valid Block Section

```
{
    "chainname" : "default",
    "blocknumber" : "1",
    "blockHeader" : {
        "bloom" :
↪"0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
        "coinbase" : "0x2adc25665018aa1fe0e6bc666dac8fc2697ff9ba",
        "difficulty" : "0x020000",
        "extraData" : "0x42",
```

(continues on next page)

(continued from previous page)


```
        "gasLimit" : "0x7fffffffffffffff",
        "gasUsed" : "0x556d",
        "hash" : "0x46a8332db77844422d2550849594af5ae994a23274255fd369d3fbe964eb7cb1",
        "mixHash" :
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
        "nonce" : "0x0000000000000000",
        "number" : "0x01",
        "parentHash" :
↪"0xd1b5003dbed66eb89c9b0062798a4fddf737157ef6550187d098d91ae2c3b853",
        "receiptTrie" :
↪"0x5fa98a3ba1e25059bf41376c6f9a57b500a02c14c6a87afd1665a3906c4afd51",
        "stateRoot" :
↪"0x6a117466ec6b8dbea3a613366a1b468bdeb6282e0156772c81cc2139caa7c88e",
        "timestamp" : "0x03e8",
        "transactionsTrie" :
↪"0x76660905a74ef4a24b19295d8740a349b1170d3298118106201d43cd7886afbf",
        "uncleHash" :
↪"0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347"
    },
    "transactions" : [
        {
            "data" :
↪"0x0000000000000000000000000000000000000000000000000000000000000000bc03712fac13c68425054e372b0861af0564
↪",
            "gasLimit" : "0x030d40",
            "gasPrice" : "0x01",
            "nonce" : "0x00",
            "r" : "0x3d55a2ac293c7ad82632b18705e67ad2a0e6177d44f601dca043934c8cd8c07a
↪",
            "s" : "0x1c069ed47162b350a1f496e9a55f53685189e9c3076a4931334a43719b9a158e
↪",
            "to" : "0x1baf27b88c48dd02b744999cf3522766929d2b2a",
            "v" : "0x1c",
            "value" : "0x00"
        }
    ],
    "uncleHeaders" : [
    ],
    "rlp" :
↪"0xf902c5f901fca0d1b5003dbed66eb89c9b0062798a4fddf737157ef6550187d098d91ae2c3b853a01dcc4de8dec75d7a
↪",
}
```


Contains the block information that is going to be imported into test chain. The key field here is **rlp**. All testers take *rlp* field data and import it on top of genesis that was described in test by **genesisRLP**

Fields *blockHeader*, *transactions*, *uncleHeaders* describe the content of *rlp* for humans to read. Its content **MUST** be equal to *rlp* field binary. All testers use **rlp** field data to import it into the test chain.

Fields *chainname*, *blocknumber* are irrelevant for the test client.

- all fields are 0x prefixed HEX
- 0 values are 0x00 rounded
- empty extraData 0x
- transaction creation *to* is “”
- the *transaction*

Valid Blocks with Invalid Transactions

```
"transactionSequence" : [
    {
        "valid" : "false",
        "rawBytes" : "0x02f87001018502540be4008502540be40086246139ca800094cc...
        "exception" : "TR_NoFunds"
    },
    {
        "valid" : "true",
        "rawBytes" : "0x02f87001018502540be4008502540be40086246139ca800094cc...
    }
]
```

Blocks that include invalid transactions have an additional field, *transactionSequence*. This field is an array of transaction information. Each entry in the array has two fields:

- *valid*, true for valid transactions (which also appear in the *transactions* field), false for invalid ones.
- *rawBytes*, the rlp-encoded raw bytes of the transaction

Entries for invalid transactions have this additional field:

- *exception*, for invalid transactions, the name of the exception they create.

12.2.6 Invalid Block Section

```
{
    "chainname" : "default",
    "blocknumber" : "1",
    "expectException" : "InvalidDifficulty",
    "rlp" :
↪"0xf902c5f901fca0d1b5003dbed66eb89c9b0062798a4fddf737157ef6550187d098d91ae2c3b853a01dcc4de8dec75d7a
↪"
}
```

The block *rlp* can represent invalid block that is expected to fail upon import. In this case there **MUST BE NO** *blockHeader*, *transactions*, *uncleHeaders* fields.

Fields *chainname*, *blocknumber*, *expectException* are irrelevant for the test client.

expectException is index in client's exception string map defined in retesteth client configs.

Fields

Fields are similar to the standard ethereum block header / transaction

12.2.7 Transaction Section

```
{
    "transaction" : {
        "data" : "0x",
        "gasLimit" : "0x061a80",
        "gasPrice" : "0x01",
        "nonce" : "0x00",
        "secretKey" :
↪"0x45a915e4d060149eb4365960e6a7a45f334393093061116b197e3240065ff2d8",
```

(continues on next page)

(continued from previous page)

```
        "to" : "0x095e7baea6a6c7c4c2dfeb977efac326af552d87",
        "value" : "0x0186a0"
    }
},
{
    "transaction" : {
        "data" : "0x",
        "gasLimit" : "0x061a80",
        "gasPrice" : "0x01",
        "nonce" : "0x00",
        "v" : "0x1c",
        "r" : "0x3d55a2ac293c7ad82632b18705e67ad2a0e6177d44f601dca043934c8cd8c07a",
        "s" : "0x1c069ed47162b350a1f496e9a55f53685189e9c3076a4931334a43719b9a158e",
        "to" : "",
        "value" : "0x0186a0"
    }
}
```

Transaction section defines single transaction to be executed in BlockchainTest's block.

- All fields are 0x prefixed HEX of even length (can be like 0x0122)
- empty data is defined as 0x
- transaction creation *to* defined as ""

Note: Fields *r*, *s* are u256 and can be less than 32 bytes!

Note: There is an EIP limiting *s* max value (source?). From a certain fork transactions with *s* value > *sMaxValue* are considered to be invalid.

Fields

`data`	**BYTES**	data/input code of the transaction
`gasLimit`	**VALUE**	gasLimit of transaction.
`gasPrice`	**VALUE**	Transaction's gas price
`nonce`	**VALUE**	Transaction's nonce
`secretKey`	**HASH32**	SecretKey criptic value used to sign tx data by v,r,s
`v`	**VALUE**	Cryptic value `1 byte in length`
`r`	**VALUE**	Values corresponding to the signature of the transaction and used to determine the sender of the transaction.
`s`	**VALUE**	Values corresponding to the signature of the transaction and used to determine the sender of the transaction.
`to`	**FH20**	Transaction's *to* destination address. `set to "" if creation`.
`value`	**VALUE**	Value of the transaction.

TYPE	Empty	Length	Format description
VALUE	0x00	Any*	0x prefixed hex up to 32 bytes long with no leading zeros.
BYTES	0x	Any*	0x prefixed bytes of any length
HASH8	0x00...00	Fixed 8	0x prefixed bytes of length 8
HASH20	0x00...00	Fixed 20	0x prefixed bytes of length 20
HASH32	0x00...00	Fixed 32	0x prefixed bytes of length 32
HASH256	0x00...00	Fixed 256	0x prefixed bytes of length 256

- Size can be limited by the meaning of field in tests. (like gasLimit ceil, tx signature v - value)

CHAPTER 13

State Transition Tests

13.1 State Transition Tests Source Code

Location: src/GeneralStateTestsFiller

State transition tests include a single transaction that is supposed to change the state of the blockchain from the **pre** state to the **expect** state.

Note: Except for the values in the **indexes** section, all the field values in the tests' source code are strings. When a value is numeric, such as a value in storage or an address's balance, it can be specified either in hexademical (starting with **0x**), or in decimal (starting with a digit). For the sake of legibility, numeric values can also have underscores. For example, you can use **1_000_000_000** for 10^9, which is a lot more readable than **1000000000**.

In YAML string is the default field type. In JSON string values are enclosed by quotes.

13.1.1 Test Structure

You can write tests either in JSON format or in YAML format. All tests are inside a structure with their name

Format

Format	JSON	YAML
Filename	name-of-testFiller.json	name-of-testFiller.yml

Format	<pre>{
	  "name-of-test": {
	  Sections go here
	  }
	}</pre>
		<pre>name-of-test:
		  Sections go here</pre>

13.1.2 Env

This section contains the environment, the block just before the one that runs the VM or executes the transaction.

Format

JSON

```
{
   "name-of-test": {
      <other sections>,
      "env" : {
         "currentCoinbase" :
↪"0x2adc25665018aa1fe0e6bc666dac8fc2697ff9
↪",
         "currentDifficulty" : "0x020000
↪",
         "currentGasLimit" : "0x05f5e100
↪",
         "currentNumber" : "0x01",
         "currentTimestamp" : "0x03e8",
         "previousHash" :
↪"0x5e20a0453cecd065ea59c37ac63e079ee08998b6045136a8ce6635c7912ec0b6
↪",,
         "currentBaseFee" : "1000"
      }
   }
}
```

YAML

```
name-of-test:
   <other sections>
   env:
      currentCoinbase:␣
↪2adc25665018aa1fe0e6bc666dac8fc2697ff9ba
      currentDifficulty: 0x20000
      currentGasLimit: 100000000
      currentNumber: 1
      currentTimestamp: 1000
      previousHash:␣
↪5e20a0453cecd065ea59c37ac63e079ee08998b6045136a8ce6635
      currentBaseFee: 1000
```

Fields

You can read the definition of Ethereum block header fields here.

Note that this section only contains the fields that are relevant to single transaction tests.

Name in Env Section	Meaning
currentCoinbase	beneficiary of mining fee
currentDifficulty	difficulty of previous block
currentGasLimit	limit of gas usage per block
currentNumber	number of ancestory blocks
currentTimestamp	Unix time
previousHash	hash of previous block
currentBaseFee	London and afterwards, the block base fee

13.1.3 Pre

This section contains the initial information of the blockchain.

Format

JSON

```
{
   "name-of-test": {
      <other sections>,
      "pre": {
         "address 1": {
             "balance": "0xba1a9ce000",
             "nonce": "0",
             "code": ":raw␣
↪0x600160010160005500"
             "storage: {
                     "0":       "12345",
                     "0x12": "0x121212"
         },
         "address 2": {
             <address fields go here>
         }
      }
   }
}
```

YAML

```
name-of-test:
   <other sections>
   pre:
     address 1:
       balance: 0xba1a9ce000,
       nonce: 0,
       code: :raw 0x600160010160005500
       storage:
         0:       12345
         0x12: 0x121212
     address 2:
       <address fields go here>
```

Address Fields

- **balance**:

  Wei balance at the start of the test

- **code**:

  The code of the contract. In the **expect:** section this has to be raw virtual machine code.

- **nonce**:

  The nonce counter for the address. This value is used to make sure each transaction is processed only once. The first transaction the address sends has a nonce equal to this value, the second one is the nonce plus one, etc.

- **storage**:

  Values in the storage of the address

  JSON

  ```
  storage: {
      "1": 5,
      "0x20": 0x10
  }
  ```

  YAML

  ```
  storage:
     1: 5
     0x20: 0x10
  ```

- **code**:

  The code of the contract. There are several possibilities:

  – If the account is not a contract, this value is **0x**

- Raw virtual machine code. This is for cases where it is impossible to provide source code, or the source code is in a language retesteth does not recognize, such as Vyper.

```
:raw 0x600160010160005500
```

- Lisp Like Language (lll), for example:

```
{
   ; Add 2+2 and store the value in storage location 0
   [[0]] (ADD 2 2)
}
```

- Yul, which is documented here, for example:

```
:yul {
   // Add 2+2 and store the value in storage location 0
   sstore(0, add(2,2))
}
```

- Solidity, which you can learn here. Solidity code can be provided to a test in two ways:
  - Put the solidity code itself in the contract definition (same place as the LLL or Yul code).
  - Put a **:solidity** section with the contract source code. In that case, the value in **code:** is **:solidity <name of contract>**.

13.1.4 Transaction

This is the data of the transaction.

Format

JSON

```
{
    "name-of-test": {
       <other sections>,
       "transaction":
          {
             "data": ["0xDA7A", "0xDA7A",
↪ ":label hex 0xDA7A",
                   ":abi f(uint) 0xDA7A",
                   {
                        "data": "0xDA7A",
                        "accessList": [
                           {
                              "address":
↪"0x0000000000000000000000000000000000000001
↪",
                              "storageKeys
↪": [0x60A7, 0xBEEF]
                           },
                           {
                              "address":
↪"0x0000000000000000000000000000000000000001
↪"
                           }
                        ]
                    }
             ],
             "gasLimit": ["0x6a506a50"],
             "value": ["1"],
             "to":
↪"add13ess01233210add13ess01233210",
             "secretKey": "5ec13e7 ...␣
↪5ec13e7"
             "nonce": '0x909ce'
             "maxPriorityFeePerGas": "10
↪",
             "maxFeePerGas": "2000",
}
```

YAML

```
name-of-test:
  <other sections>
  transaction:
    data:
    - 0xDA7A
    - 0xDA7A
    - :label hex 0xDA7A
    - :abi f(uint) 0xDA7A
    - data: :label acl 0xDA7A
      accessList:
      - address:␣
↪0x0000000000000000000000000000000000000101
        storageKeys:
        - 0x60A7
        - 0xBEEF
      - address:␣
↪0x0000000000000000000000000000000000000102
    gasLimit:
    - '0xga50ga50'
    value:
    - "1"
    to: "add13ess01233210add13ess01233210
↪"
    secretKey: "5ec13e7 ... 5ec13e7"
    nonce: '0x909ce'
    maxPriorityFeePerGas: 10
    maxFeePerGas: 2000
```

Fields

- **data**:

  The data, either in hexadecimal or an ABI call with this format: **:abi <function signature> <function parameters separated by spaces>**. The value can also be labeled: **:label <value>**. This value is specified as a list to enable files with multiple tests

  The data can also have an EIP2930 access list. In that case the data field itself is a structure with two fields: **data** (the data) and **accessList**. The **accessList** is a list of structures, each of which has to have an **address** and may have a list of **storageKeys**.

- **gasLimit**:

Gas limit for the transaction. This value is specified as a list to enable files with multiple tests

- **gasPrice**:

  Gas price in Wei, only in Berlin and earlier (replaced by maxFeePerGas in London)

- **value**:

  The value the transaction transmits in Wei. This value is specified as a list to enable files with multiple tests

- **to**:

  The destination address, typically a contract

- **secretKey**:

  The secret key for the sending address. That address is derived from the secret key and therefore does not need to be specified explicitely (see here).

- **nonce**:

  The nonce value for the transaction. The first transaction for an address has the nonce value of the address itself, the second transaction has the nonce plus one, etc.

- **maxPriorityFeePerGas**:

  The maximum priority fee per gas (a.k.a. tip) the transaction is willing to pay to be included in the block (London and later, added by eip 1559).

- **maxFeePerGas**:

  The maximum total fee per gas the transaction is willing to pay to be included in the block (London and later, added by eip 1559).

13.1.5 Expect

This section contains the information we expect to see after the test is concluded.

Format

JSON

```
{
   "name-of-test": {
      <other sections>,
      "expect": [
        {
           "indexes": {
             "data": [0, "2-3", ":label␣
↪foo"],
             "gas": -1,
             "value": -1
           },
           "network": ["Istanbul",
↪<other forks, see below>],
           "result": {
                "address 1": {
                    "balance":
↪"0xba1a9ce000",
                    "nonce": "0",
                    "storage: {
                        "0x0":     12345,
                        "10" : "0x121212"
                    },
                    "code": "0x00"
                },
                "address 2": {
                    <address fields go␣
↪here>
                }
        },
        { <forks & results> }
      ]
   }
}
```

YAML

```
name-of-test:
   <other sections>
   expect:
   - indexes:
       data:
       - !!int 0
       - 2-3
       - :label foo
       gas: !!int -1
       value: !!int -1
     network:
     - Istanbul
     - <another fork>
     result:
       address 1:
         balance: 0xba1a9ce000,
         nonce: 0,
         storage:
           0x0:  12345
           10: 0x121212
         code: 0x00
       address 2:
          <address fields go here>
   - <forks & results>
```

The Network Specification

The string that identifies a fork (version) within a **network:** list is one of three option:

- The specific version: **Istanbul**
- The version or anything later: **>=Frontier**
- Anything up to (but not including) the version **<Constantinople**

The Indexes

The transaction can have multiple values for **data**, **gasLimit**, and **value**. The **indexes:** section specifies which of these values are covered by a particular item in **expect**, for each field it can be either a single specification or a list of specifications. Each of those specifications uses any of these options:

JSON	YAML	Meaning
-1	!!int -1	All the (**data**, **gas**, or **value**) values in the transaction
<n>	!!int <n>	The n'th value in the list (counting from zero)
"<a>-<b>"	a-b	Everthing from the a'th value to the b'th value (counting from zero)
":label foo"	:label foo	Any value in the list that is specified as **:label foo <value>**

Address Fields

It is not necessary to include all fields for every address. Only include those fields you wish to test.

- **balance**:

  Wei balance at the start of the test

- **code**:

  The code of the contract. In the **expect:** section this has to be raw virtual machine code.

- **nonce**:

  The nonce counter for the address. This value is used to make sure each transaction is processed only once. The first transaction the address sends has a nonce equal to this value, the second one is the nonce plus one, etc.

- **storage**:

  Values in the storage of the address

JSON	YAML

`storage: {`
`  "1": 5,`
`  "0x20": 0x10`
`}`
	`storage:`
	`  1: 5`
	`  0x20: 0x10`

Expect Exception

This field specifies the exception we expect to see raised by for this transaction. It is optional - you only add it if an exception is expected.

JSON	YAML

`"expectException": {`
`  ">=London": TR_TipGtFeeCap`
`}`
	`expectException:`
	`  ">=London": TR_TipGtFeeCap`

The fields are fork specifications:

Type of specification	Example
Specific fork	Berlin
A specific fork and all forks after it	>=London
Anything prior to a specific fork (not including that fork)	<Berlin

The value is an exception name. You can see the list in the retesteth code.

13.2 Generated State Transition Tests

Location /GeneralStateTests

13.2.1 Test Structure

Contains **transactions** that are to be executed on a state **pre** given the environment **env** and must end up with post results **post**

Although its a simple transaction execution on stateA to stateB, due to the generation of this tests into blockchain format, the transaction execution is performed as if it was a single block with single transaction. This means that mining reward and touch rules after EIP-161 are applied. (mining reward is 0)

- A test file must contain **only one** test *testname*
- Test file name must be **identical** for the test name *testname*

```
{
  "testname" : {
    "_info" : { ... },
    "env" : { ... },
    "post" : { ... },
    "pre" : { ... },
    "transaction" : { ... }
  }
}
```

13.2.2 Info Section

```
"_info" : {
         "comment" : "A test for (add 1 1) opcode result",
         "filling-rpc-server" : "Geth-1.9.14-unstable-8cf83419-20200512",
         "filling-tool-version" : "retesteth-0.0.3+commit.672a84dd.Linux.g++",
         "lllcversion" : "Version: 0.5.14-develop.2019.11.27+commit.8f259595.Linux.g++
↪",
         "source" : "src/GeneralStateTestsFiller/stExample/add11Filler.json",
         "sourceHash" :
↪"e474fc13b1ea4c60efe2ba925dd48d6f9c1b12317dcd631f5eeeb3722a790a37"
 },
```

Info section is generated with the test and contains information about test and it's generators.

Fields

`comment`	comment from the test source. (can be edited at source)
`filling-rpc-server`	tool that has been used to generate the test (version)
`filling-tool-version`	the test generator (retesteth) version
`lllcversion`	lllc version that was used to compile LLL code in test fillers
`source`	path to the source filler in the test repository
`sourceHash`	hash of the json of source file (used to track that tests are updated)

13.2.3 Env Section

```
"env" : {
    "currentCoinbase" : "0x2adc25665018aa1fe0e6bc666dac8fc2697ff9ba",
    "currentDifficulty" : "0x020000",
    "currentGasLimit" : "0xff112233445566",
    "currentNumber" : "0x01",
    "currentTimestamp" : "0x03e8",
    "previousHash" :
↪"0x5e20a0453cecd065ea59c37ac63e079ee08998b6045136a8ce6635c7912ec0b6"
},
```

Env section describe information required to construct a genesis block, or VM env for transaction execution.

- The fields are always 0x prefixed HEX.

Fields

currentCoinbase	author/miner/coinbase address
currentDifficulty	transaction executed in a block with this difficulty
currentGasLimit	transaction executed in a block with this gas limit
currentNumber	transaction executed in a block with this number
currentTimestamp	transaction executed in a block with this timestamp
previousHash	hash of the previous block (deprecated)

13.2.4 Post Section

```
"post" : {
    "London" : [
        {
            "indexes" : {
                "data" : 0,
                "gas" : 0,
                "value" : 0
            },
            "hash" :
↪"0xe4c855f0d0e96d48d73778772ee570c45acb7c57f87092e08fed6b2205d390f4",
            "logs" :
↪"0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347",
            "expectException" : "TR_IntrinsicGas"
        }
    ]
},
```

Post section is a map *<FORK> => [TransactionResults]*

The test can have many fork results and each fork result can have many transaction results.

In generated test indexes are a single digit and could not be array. Thus define a single transaction from the test. See transaction section which define transactions by *data*, *gasLimit*, *value* arrays.

Fields

London	fork name as defined by client config (test standard names)
indexes	define an index of the transaction in txs vector that has been used for this result
data	index in transaction data vector
gas	index in transaction gas vector
value	index in transaction value vector
hash	hash of the post state after transaction execution
logs	log hash of the transaction logs
expectException	for a transaction that is supposed to fail, the exception

13.2.5 Pre/preState Section

```
"pre" : {
    "0x095e7baea6a6c7c4c2dfeb977efac326af552d87" : {
        "balance" : "0x0de0b6b3a7640000",
        "code" : "0x600160010160005500",
        "nonce" : "0x00",
        "storage" : {
            "0x00" : "0x01"
        }
    },
    "0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b" : {
        "balance" : "0x0de0b6b3a7640000",
        "code" : "0x",
        "nonce" : "0x00",
        "storage" : {
        }
    }
},
```

Pre section describes a full state of accounts used in test.

Its a map of <Account> => <AccountFields>

AccountFields are always complete (*balance*, *code*, *nonce*, *storage* must present) in this section and can not have a missing field.

- All values are 0x prefixed hex.
- Empty code defined as 0x.
- Zero storage record defined as 0x00.

Fields

address hash	**HASH20**	is 20 bytes ethereum address 0x prefixed
balance	**VALUE**	account balance in evm state
code	**BYTES**	account code in evm state
nonce	**VALUE**	account nonce in evm state
storage	**map**	map of storage records **VALUE => VALUE**

TYPE	Empty	Length	Format description
VALUE	0x00	Any*	0x prefixed hex up to 32 bytes long with no leading zeros.
BYTES	0x	Any*	0x prefixed bytes of any length
HASH8	0x00...00	Fixed 8	0x prefixed bytes of length 8
HASH20	0x00...00	Fixed 20	0x prefixed bytes of length 20
HASH32	0x00...00	Fixed 32	0x prefixed bytes of length 32
HASH256	0x00...00	Fixed 256	0x prefixed bytes of length 256

- Size can be limited by the meaning of field in tests. (like gasLimit ceil, tx signature v - value)

13.2.6 Transaction Section

```
"transaction" : {
    "data" : [
        "0x"
    ],
    "gasLimit" : [
        "0x061a80"
    ],
    "gasPrice" : "0x01",
    "nonce" : "0x00",
    "secretKey" : "0x45a915e4d060149eb4365960e6a7a45f334393093061116b197e3240065ff2d8
↪",
    "to" : "0x095e7baea6a6c7c4c2dfeb977efac326af552d87",
    "value" : [
        "0x0186a0"
    ]
}
```

Transaction section defines a vector of transaction to be executed in GeneralStateTest From this section it is possible to construct many transaction using values from data,gasLimit,value array. Indexes in this array used in the post section to point out which transaction has been used to calculate the post state hash.

- All fields are 0x prefixed HEX of even length (can be like 0x0122)
- empty data is defined as 0x
- transaction creation *to* defined as “”

Fields

data	array(**BYTES**)	Array of data/input of transaction. In Post section indexes::data index indicates index in this array.
gasLimit	array(**VALUE**)	Array of gasLimit of transaction. In Post section indexes::gas index indicates index in this array
gasPrice	**VALUE**	Transaction’s gas price
nonce	**VALUE**	Transaction’s nonce
secretKey	**FH32**	SecretKey criptic value used to sign tx data by v,r,s
to	**FH20**	Transaction’s *to* destination address. set to “” if creation.
value	**VALUE**	Array of value of transaction. In Post section indexes::value index indicates index in this array

Note that the state transition tests now also include the VM tests. They are located in **GeneralStateTests/VMTests**.

CHAPTER 14

Sample Values

These tests cannot be executed automatically by **retesteth**. Instead, they are known valid values that client programmers can plug into their own unit tests to check various aspects of their client.

14.1 ABI Tests

Location /ABITests/basic_abi_tests.json

A number of test cases for the application binary interface. These test cases only include the encoded arguments, not the the first four bytes, which are a hash of the function name and parameter types.

The format of each test value is:

```
"<name of test>": {
```

The data types of the arguments, a list of strings.

```
"types": [
   "uint256",
   "bytes",
   "uint32[]"
],
```

The values of the arguments. These can be integers, strings, or arrays:

```
"args": [
   0xda7a0000da7a0000,
   "a string",
   [16, 256]
],
```

The encoded arguments, a hexadecimal string:

```
"result":
↪"0000000000000000000000000000000000000000000000000da7a0000da7a0000000000000000000000000000000000000000
↪"
}
```

14.2 Difficulty Test

Location /BasicTests

14.2.1 Test Implementation

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

```
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 (parentDifficulty, 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)
`difficultyOlimpic.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)

14.2.2 Test Structure

```
{
        "difficultyTest" : {
                "parentTimestamp" : "42",
                "parentDifficulty" : "1000000",
                "currentTimestamp" : "43",
                "currentBlockNumber" : "42",
                "currentDifficulty" : "1000488"
        }
}
```

Fileds

`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

14.3 RLP Test

Location /RLPTests

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

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

14.3.2 Test Structure

```
{
   "rlpTest": {
        "in": "dog",
        "out": "0x83646f67"
   },

   "multilist": {
        "in": [ "zw", [ 4 ], 1 ],
        "out": "0xc6827a77c10401"
   },

   "validRLP": {
        "in": "VALID",
        "out": "0xc7c0c1c0c3c0c1c0"
   },

   "invalidRLP": {
        "in": "INVALID",
        "out": "0xbf0f000000000000021111"
   },
   ...
}
```

Fields

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

When a json string starts with `0x`, the rest of the string is interpreted as hex bytes, and when one starts with `#`, the rest is interpreted as a decimal number. For example `5050` and `"#5050"` both represent the decimal number `5050`. Strings with `#` prefixes should be used for numbers that would be too big to represented as `int` values, and would require a "bigint" representation.

The `out` strings normally start with `0x` to be interpreted as hex bytes.

14.4 Transaction Test

Warning: There is a filler directory for these tests /src/TransactionTestsFiller, but it has not been maintained and is no longer supported. If you need to create a transaction test, you need to create a filled version.

Location /TransactionTests

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

14.4.1 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 in the fork section.

14.4.2 Test Structure

```
{
   "transactionTest1": {
      "_info" : { ... },
          "rlp" : "bytearray",
          "Istanbul" : {
              "sender" : "2ea991808ba979ba103147edfd72304ebd95c028",
              "hash" :
↪"4782cb5edcaeda1f0aef204b161214f124cefade9e146245183abbb9ca01bca5"
          }
   },

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

Fields

`_info`		Info section about test origin generated by test generator
`rlp`	**BYTES**	RLP encoded data of this transaction
`hash`	**FH32**	Hash of the transaction derived from *rlp* field
`sender`	**FH20**	The address of the sender, derived from the v,r,s values.

TYPE	Empty	Length	Format description
`VALUE`	0x00	Any*	0x prefixed hex up to 32 bytes long with no leading zeros.
`BYTES`	0x	Any*	0x prefixed bytes of any length
`HASH8`	0x00...00	Fixed 8	0x prefixed bytes of length 8
`HASH20`	0x00...00	Fixed 20	0x prefixed bytes of length 20
`HASH32`	0x00...00	Fixed 32	0x prefixed bytes of length 32
`HASH256`	0x00...00	Fixed 256	0x prefixed bytes of length 256

- Size can be limited by the meaning of field in tests. (like gasLimit ceil, tx signature v - value)

Next are descriptions of transaction fields that are encoded in *rlp*:

14.4.3 Transaction Section

```
{
    "transaction" : {
        "data" : "0x",
        "gasLimit" : "0x061a80",
        "gasPrice" : "0x01",
        "nonce" : "0x00",
        "secretKey" :
↪"0x45a915e4d060149eb4365960e6a7a45f334393093061116b197e3240065ff2d8",
        "to" : "0x095e7baea6a6c7c4c2dfeb977efac326af552d87",
        "value" : "0x0186a0"
    }
},
{
    "transaction" : {
        "data" : "0x",
        "gasLimit" : "0x061a80",
        "gasPrice" : "0x01",
        "nonce" : "0x00",
        "v" : "0x1c",
        "r" : "0x3d55a2ac293c7ad82632b18705e67ad2a0e6177d44f601dca043934c8cd8c07a",
        "s" : "0x1c069ed47162b350a1f496e9a55f53685189e9c3076a4931334a43719b9a158e",
        "to" : "",
        "value" : "0x0186a0"
    }
}
```

Transaction section defines single transaction to be executed in BlockchainTest's block.

- All fields are 0x prefixed HEX of even length (can be like 0x0122)
- empty data is defined as 0x
- transaction creation *to* defined as ""

Note: Fields *r*, *s* are u256 and can be less than 32 bytes!

Note: There is an EIP limiting *s* max value (source?). From a certain fork transactions with *s* value > *sMaxValue* are considered to be invalid.

Fields

`data`	**BYTES**	data/input code of the transaction
`gasLimit`	**VALUE**	gasLimit of transaction.
`gasPrice`	**VALUE**	Transaction's gas price
`nonce`	**VALUE**	Transaction's nonce
`secretKey`	**HASH32**	SecretKey criptic value used to sign tx data by v,r,s
`v`	**VALUE**	Cryptic value `1 byte in length`
`r`	**VALUE**	Values corresponding to the signature of the transaction and used to determine the sender of the transaction.
`s`	**VALUE**	Values corresponding to the signature of the transaction and used to determine the sender of the transaction.
`to`	**FH20**	Transaction's *to* destination address. `set to "" if creation.`
`value`	**VALUE**	Value of the transaction.

TYPE	Empty	Length	Format description
`VALUE`	0x00	Any*	0x prefixed hex up to 32 bytes long with no leading zeros.
`BYTES`	0x	Any*	0x prefixed bytes of any length
`HASH8`	0x00...00	Fixed 8	0x prefixed bytes of length 8
`HASH20`	0x00...00	Fixed 20	0x prefixed bytes of length 20
`HASH32`	0x00...00	Fixed 32	0x prefixed bytes of length 32
`HASH256`	0x00...00	Fixed 256	0x prefixed bytes of length 256

- Size can be limited by the meaning of field in tests. (like gasLimit ceil, tx signature v - value)

14.5 Trie Tests

Location /TrieTests/

These are sample trie structures.

This is the format of most of those tests:

```
{
  "name of test": {
    "in": [
      ["do", "verb"],
      ["ether", "wookiedoo"],
      ["horse", "stallion"],
      ["shaman", "horse"],
      ["doge", "coin"],
      ["ether", null],
      ["dog", "puppy"],
      ["shaman", null]
```

(continues on next page)

(continued from previous page)

```
    ],
    "root": "0x29b235a58c3c25ab83010c327d5932bcf05324b7d6b1185e650798034783ca9d"
  }
}
```

The fields are:

- **in**, The data to store in the trie, which can be either a map object or a list in which each item contains a list of a key and the corresponding value.
- **root**, the hash expected at the root of the trie after adding all of those items
- **hexEncoded** (optional), if this field exists and is **true**, it means the strings for the keys and values are already encoded hexadecimal, rather than ASCII strings.

14.5.1 Next and Previous Test

The test /TrieTests/trietestnextprev.json is formatted differently. Instead of testing the entire trie data structure, this file is used to test individual operations within this structure.

14.6 Miscellaneous Tests

These are various test values that do not belong under any of the major categories.

14.6.1 Cryptographic Tests

- /BasicTests/crypto.json

/BasicTests/crypto.json <https://github.com/ethereum/tests/blob/develop/KeyStoreTests/basic_tests.json>‘_

14.6.2 Encoding Tests

Tests for the encoding of various data types.

- /BasicTests/hexencodetest.json
- /BasicTests/keyaddrtest.json
- /BasicTests/txtest.json

14.6.3 Genesis Block Tests

Tests related to the genesis block at the beginning of a block chain:

- /BasicTests/blockgenesistest.json
- /BasicTests/genesishashestest.json
- /GenesisTests/basic_genesis_tests.json

CHAPTER 15

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 16

Indices and tables

- genindex
- modindex
- search