This directory contains integration tests that test bitcoind and its utilities in their entirety. It does not contain unit tests, which can be found in /src/test, /src/wallet/test, etc.

This directory contains the following sets of tests:

- fuzz A runner to execute all fuzz targets from /src/test/fuzz.
- functional which test the functionality of bitcoind and bitcoin-qt by interacting with them through the RPC and P2P interfaces.
- util which tests the utilities (bitcoin-util, bitcoin-tx, ...).
- lint which perform various static analysis checks.

The util tests are run as part of `make check` target. The fuzz tests, functional tests and lint scripts can be run as explained in the sections below.

Running tests locally

Before tests can be run locally, Bitcoin Core must be built. See the building instructions for help.

Fuzz tests

See /doc/fuzzing.md

Functional tests

Dependencies and prerequisites

The ZMQ functional test requires a python ZMQ library. To install it:

- on Unix, run `sudo apt-get install python3-zmq`
- on mac OS, run `pip3 install pyzmq`

On Windows the `PYTHONUTF8` environment variable must be set to 1:

```
set PYTHONUTF8=1
```

Running the tests

Individual tests can be run by directly calling the test script, e.g.:

```
test/functional/feature_rbf.py
```

or can be run through the test_runner harness, eg:

```
test/functional/test_runner.py feature_rbf.py
```

You can run any combination (incl. duplicates) of tests by calling:

```
test/functional/test_runner.py <testname1> <testname2> <testname3> ...
```

Wildcard test names can be passed, if the paths are coherent and the test runner is called from a `bash` shell or similar that does the globbing. For example, to run all the wallet tests:

```
test/functional/test_runner.py test/functional/wallet*
functional/test_runner.py functional/wallet* (called from the test/ directory)
test_runner.py wallet* (called from the test/functional/ directory)
```

but not

```
test/functional/test_runner.py wallet*
```

Combinations of wildcards can be passed:

```
test/functional/test_runner.py ./test/functional/tool* test/functional/mempool*
test_runner.py tool* mempool*
```

Run the regression test suite with:

```
test/functional/test_runner.py
```

Run all possible tests with

```
test/functional/test_runner.py --extended
```

In order to run backwards compatibility tests, download the previous node binaries:

```
test/get_previous_releases.py -b v22.0 v0.21.0 v0.20.1 v0.19.1 v0.18.1 v0.17.2 v0.16.3
v0.15.2 v0.14.3
```

By default, up to 4 tests will be run in parallel by test_runner. To specify how many jobs to run, append `--jobs=n`

The individual tests and the test_runner harness have many command-line options. Run `test/functional/test_runner.py -h` to see them all.

Speed up test runs with a ramdisk

If you have available RAM on your system you can create a ramdisk to use as the `cache` and `tmp` directories for the functional tests in order to speed them up. Speed-up amount varies on each system (and according to your ram speed and other variables), but a 2-3x speed-up is not uncommon.

To create a 4GB ramdisk on Linux at `/mnt/tmp/` :

```
sudo mkdir -p /mnt/tmp
sudo mount -t tmpfs -o size=4g tmpfs /mnt/tmp/
```

Configure the size of the ramdisk using the `size=` option. The size of the ramdisk needed is relative to the number of concurrent jobs the test suite runs. For example running the test suite with `--jobs=100` might need a 4GB ramdisk, but running with `--jobs=32` will only need a 2.5GB ramdisk.

To use, run the test suite specifying the ramdisk as the `cachedir` and `tmpdir` :

```
test/functional/test_runner.py --cachedir=/mnt/tmp/cache --tmpdir=/mnt/tmp
```

Once finished with the tests and the disk, and to free the ram, simply unmount the disk:

```
sudo umount /mnt/tmp
```

Troubleshooting and debugging test failures

Resource contention

The P2P and RPC ports used by the bitcoind nodes-under-test are chosen to make conflicts with other processes unlikely. However, if there is another bitcoind process running on the system (perhaps from a previous test which hasn't successfully killed all its bitcoind nodes), then there may be a port conflict which will cause the test to fail. It is recommended that you run the tests on a system where no other bitcoind processes are running.

On linux, the test framework will warn if there is another bitcoind process running when the tests are started.

If there are zombie bitcoind processes after test failure, you can kill them by running the following commands. **Note that these commands will kill all bitcoind processes running on the system, so should not be used if any non-test bitcoind processes are being run.**

```
killall bitcoind
```

or

```
pkill -9 bitcoind
```

Data directory cache

A pre-mined blockchain with 200 blocks is generated the first time a functional test is run and is stored in test/cache. This speeds up test startup times since new blockchains don't need to be generated for each test. However, the cache may get into a bad state, in which case tests will fail. If this happens, remove the cache directory (and make sure bitcoind processes are stopped as above):

```
rm -rf test/cache
killall bitcoind
```

Test logging

The tests contain logging at five different levels (DEBUG, INFO, WARNING, ERROR and CRITICAL). From within your functional tests you can log to these different levels using the logger included in the test_framework, e.g. `self.log.debug(object)`. By default:

- when run through the test_runner harness, *all* logs are written to `test_framework.log` and no logs are output to the console.
- when run directly, *all* logs are written to `test_framework.log` and INFO level and above are output to the console.
- when run by our CI (Continuous Integration), no logs are output to the console. However, if a test fails, the `test_framework.log` and bitcoind `debug.log`s will all be dumped to the console to help troubleshooting.

These log files can be located under the test data directory (which is always printed in the first line of test output):

- `<test data directory>/test_framework.log`
- `<test data directory>/node<node number>/regtest/debug.log`.

The node number identifies the relevant test node, starting from `node0`, which corresponds to its position in the nodes list of the specific test, e.g. `self.nodes[0]`.

To change the level of logs output to the console, use the `-l` command line argument.

`test_framework.log` and bitcoind `debug.log`s can be combined into a single aggregate log by running the `combine_logs.py` script. The output can be plain text, colorized text or html. For example:

```
test/functional/combine_logs.py -c <test data directory> | less -r
```

will pipe the colorized logs from the test into less.

Use `--tracerpc` to trace out all the RPC calls and responses to the console. For some tests (eg any that use `submitblock` to submit a full block over RPC), this can result in a lot of screen output.

By default, the test data directory will be deleted after a successful run. Use `--nocleanup` to leave the test data directory intact. The test data directory is never deleted after a failed test.

Attaching a debugger

A python debugger can be attached to tests at any point. Just add the line:

```
import pdb; pdb.set_trace()
```

anywhere in the test. You will then be able to inspect variables, as well as call methods that interact with the bitcoind nodes-under-test.

If further introspection of the bitcoind instances themselves becomes necessary, this can be accomplished by first setting a pdb breakpoint at an appropriate location, running the test to that point, then using `gdb` (or `lldb` on macOS) to attach to the process and debug.

For instance, to attach to `self.node[1]` during a run you can get the pid of the node within `pdb`.

```
(pdb) self.node[1].process.pid
```

Alternatively, you can find the pid by inspecting the temp folder for the specific test you are running. The path to that folder is printed at the beginning of every test run:

```
2017-06-27 14:13:56.686000 TestFramework (INFO): Initializing test directory
/tmp/user/1000/testo9vsdjo3
```

Use the path to find the pid file in the temp folder:

```
cat /tmp/user/1000/testo9vsdjo3/node1/regtest/bitcoind.pid
```

Then you can use the pid to start `gdb`:

```
gdb /home/example/bitcoind <pid>
```

Note: gdb attach step may require ptrace_scope to be modified, or `sudo` preceding the `gdb`. See this link for considerations: https://www.kernel.org/doc/Documentation/security/Yama.txt

Often while debugging rpc calls from functional tests, the test might reach timeout before process can return a response. Use `--timeout-factor 0` to disable all rpc timeouts for that partcular functional test. Ex: `test/functional/wallet_hd.py --timeout-factor 0`.

Profiling

An easy way to profile node performance during functional tests is provided for Linux platforms using `perf`.

Perf will sample the running node and will generate profile data in the node's datadir. The profile data can then be presented using `perf report` or a graphical tool like hotspot.

To generate a profile during test suite runs, use the `--perf` flag.

To see render the output to text, run

```
perf report -i /path/to/datadir/send-big-msgs.perf.data.xxxx --stdio | c++filt |
less
```

For ways to generate more granular profiles, see the README in test/functional.

Util tests

Util tests can be run locally by running `test/util/test_runner.py`. Use the `-v` option for verbose output.

Lint tests

Dependencies

Lint test	Dependency
`lint-python.sh`	flake8
`lint-python.sh`	mypy
`lint-python.sh`	pyzmq
`lint-python-dead-code.py`	vulture
`lint-shell.sh`	ShellCheck
`lint-spelling.py`	codespell

In use versions and install instructions are available in the CI setup.

Please be aware that on Linux distributions all dependencies are usually available as packages, but could be outdated.

Running the tests

Individual tests can be run by directly calling the test script, e.g.:

```
test/lint/lint-files.py
```

You can run all the shell-based lint tests by running:

```
test/lint/lint-all.sh
```

Writing functional tests

You are encouraged to write functional tests for new or existing features. Further information about the functional test framework and individual tests is found in test/functional.