Tests

Tests are broadly divided into *unit tests* (test/unit), *functional tests* (test/functional), and *old tests* (src/nvim/testdir/).

- *Unit* testing is achieved by compiling the tests as a shared library which is loaded and called by LuaJit FFI.
- *Functional* tests are driven by RPC, so they do not require LuaJit (as opposed to Lua).

You can learn the key concepts of Lua in 15 minutes. Use any existing test as a template to start writing new tests.

Tests are run by `/cmake/RunTests.cmake` file, using `busted` (a Lua test-runner). For some failures, `.nvimlog` (or `$NVIM_LOG_FILE`) may provide insight.

Depending on the presence of binaries (e.g., `xclip`) some tests will be ignored. You must compile with libintl to prevent `E319: The command is not available in this version` errors.

- Running tests
- Writing tests
- Lint
- Configuration

Layout

- `/test/benchmark` : benchmarks
- `/test/functional` : functional tests
- `/test/unit` : unit tests
- `/test/config` : contains `*.in` files which are transformed into `*.lua` files using `configure_file` CMake command: this is for accessing CMake variables in lua tests.
- `/test/includes` : include-files for use by luajit `ffi.cdef` C definitions parser: normally used to make macros not accessible via this mechanism accessible the other way.
- `/test/*/preload.lua` : modules preloaded by busted `--helper` option
- `/test/**/helpers.lua` : common utility functions for test code
- `/test/*/**/*_spec.lua` : actual tests. Files that do not end with `_spec.lua` are libraries like `/test/**/helpers.lua`, except that they have some common topic.
- `/src/nvim/testdir` : old tests (from Vim)

Running tests

Executing Tests

To run all tests (except "old" tests):

```
make test
```

To run only *unit* tests:

```
make unittest
```

To run only *functional* tests:

```
make functionaltest
```

Legacy tests

To run all legacy Vim tests:

```
make oldtest
```

To run a *single* legacy test file you can use either:

```
make oldtest TEST_FILE=test_syntax.vim
```

or:

```
make src/nvim/testdir/test_syntax.vim
```

- Specify only the test file name, not the full path.

Debugging tests

- You can set `$GDB` to run tests under gdbserver. And if `$VALGRIND` is set it will pass `--vgdb=yes` to valgrind instead of starting gdbserver directly.
- Hanging tests often happen due to unexpected `:h press-enter` prompts. The default screen width is 50 columns. Commands that try to print lines longer than 50 columns in the command-line, e.g. `:edit very...long...path`, will trigger the prompt. In this case, a shorter path or `:silent edit` should be used.
- If you can't figure out what is going on, try to visualize the screen. Put this at the beginning of your test:

  ```
  local Screen = require('test.functional.ui.screen')
  local screen = Screen.new()
  screen:attach()
  ```

  Afterwards, put `screen:snapshot_util()` at any position in your test. See the comment at the top of `test/functional/ui/screen.lua` for more.

Filtering Tests

Filter by name

Another filter method is by setting a pattern of test name to `TEST_FILTER` or `TEST_FILTER_OUT`.

```
it('foo api',function()
  ...
end)
it('bar api',function()
  ...
end)
```

To run only test with filter name:

```
TEST_FILTER='foo.*api' make functionaltest
```

To run all tests except ones matching a filter:

```
TEST_FILTER_OUT='foo.*api' make functionaltest
```

Filter by file

To run a *specific* unit test:

```
TEST_FILE=test/unit/foo.lua make unittest
```

To run a *specific* functional test:

```
TEST_FILE=test/functional/foo.lua make functionaltest
```

To *repeat* a test:

```
BUSTED_ARGS="--repeat=100 --no-keep-going" TEST_FILE=test/functional/foo_spec.lua make functionaltest
```

Filter by tag

Tests can be "tagged" by adding `#` before a token in the test description.

```
it('#foo bar baz', function()
  ...
end)
it('#foo another test', function()
  ...
end)
```

To run only the tagged tests:

```
TEST_TAG=foo make functionaltest
```

NOTE:

- `TEST_FILE` is not a pattern string like `TEST_TAG` or `TEST_FILTER`. The given value to `TEST_FILE` must be a path to an existing file.
- Both `TEST_TAG` and `TEST_FILTER` filter tests by the string descriptions found in `it()` and `describe()`.

Writing tests

Guidelines

- Luajit needs to know about type and constant declarations used in function prototypes. The helpers.lua file automatically parses `types.h`, so types used in the tested functions could be moved to it to avoid having

to rewrite the declarations in the test files.
    - `#define` constants must be rewritten `const` or `enum` so they can be "visible" to the tests.
- Use **pending()** to skip tests (example). Do not silently skip the test with `if-else`. If a functional test depends on some external factor (e.g. the existence of `md5sum` on `$PATH`), *and* you can't mock or fake the dependency, then skip the test via `pending()` if the external factor is missing. This ensures that the *total* test-count (success + fail + error + pending) is the same in all environments.
    - *Note:* `pending()` is ignored if it is missing an argument, unless it is contained in an `it()` block. Provide empty function argument if the `pending()` call is outside `it()` (example).
- Really long `source([=[...]=])` blocks may break Vim's Lua syntax highlighting. Try `:syntax sync fromstart` to fix it.

Where tests go

Tests in `/test/unit` and `/test/functional` are divided into groups by the semantic component they are testing.

- *Unit tests* (test/unit) should match 1-to-1 with the structure of `src/nvim/`, because they are testing functions directly. E.g. unit-tests for `src/nvim/undo.c` should live in `test/unit/undo_spec.lua`.
- *Functional tests* (test/functional) are higher-level (plugins and user input) than unit tests; they are organized by concept.
    - Try to find an existing `test/functional/*/*_spec.lua` group that makes sense, before creating a new one.

Lint

`make lint` (and `make lualint`) runs luacheck on the test code.

If a luacheck warning must be ignored, specify the warning code. Example:

```
-- luacheck: ignore 621
```

http://luacheck.readthedocs.io/en/stable/warnings.html

Ignore the smallest applicable scope (e.g. inside a function, not at the top of the file).

Configuration

Test behaviour is affected by environment variables. Currently supported (Functional, Unit, Benchmarks) (when Defined; when set to *1*; when defined, treated as Integer; when defined, treated as String; when defined, treated as Number; !must be defined to function properly):

- `BUSTED_ARGS` (F) (U): arguments forwarded to `busted`.
- `GDB` (F) (D): makes nvim instances to be run under `gdbserver`. It will be accessible on `localhost:7777`: use `gdb build/bin/nvim`, type `target remote :7777` inside.
- `GDBSERVER_PORT` (F) (I): overrides port used for `GDB`.
- `VALGRIND` (F) (D): makes nvim instances to be run under `valgrind`. Log files are named `valgrind-%p.log` in this case. Note that non-empty valgrind log may fail tests. Valgrind arguments may be seen in

`/test/functional/helpers.lua`. May be used in conjunction with `GDB`.

- `VALGRIND_LOG` (F) (S): overrides valgrind log file name used for `VALGRIND`.
- `TEST_COLORS` (F) (U) (D): enable pretty colors in test runner.
- `TEST_SKIP_FRAGILE` (F) (D): makes test suite skip some fragile tests.
- `TEST_TIMEOUT` (FU) (I): specifies maximum time, in seconds, before the test suite run is killed
- `NVIM_LUA_NOTRACK` (F) (D): disable reference counting of Lua objects
- `NVIM_PROG`, `NVIM_PRG` (F) (S): override path to Neovim executable (default to `build/bin/nvim`).
- `CC` (U) (S): specifies which C compiler to use to preprocess files. Currently only compilers with gcc-compatible arguments are supported.
- `NVIM_TEST_MAIN_CDEFS` (U) (1): makes `ffi.cdef` run in main process. This raises a possibility of bugs due to conflicts in header definitions, despite the counters, but greatly speeds up unit tests by not requiring `ffi.cdef` to do parsing of big strings with C definitions.
- `NVIM_TEST_PRINT_I` (U) (1): makes `cimport` print preprocessed, but not yet filtered through `formatc` headers. Used to debug `formatc`. Printing is done with the line numbers.
- `NVIM_TEST_PRINT_CDEF` (U) (1): makes `cimport` print final lines which will be then passed to `ffi.cdef`. Used to debug errors `ffi.cdef` happens to throw sometimes.
- `NVIM_TEST_PRINT_SYSCALLS` (U) (1): makes it print to stderr when syscall wrappers are called and what they returned. Used to debug code which makes unit tests be executed in separate processes.
- `NVIM_TEST_RUN_FAILING_TESTS` (U) (1): makes `itp` run tests which are known to fail (marked by setting third argument to `true`).
- `LOG_DIR` (FU) (S!): specifies where to seek for valgrind and ASAN log files.
- `NVIM_TEST_CORE_*` (FU) (S): a set of environment variables which specify where to search for core files. Are supposed to be defined all at once.
- `NVIM_TEST_CORE_GLOB_DIRECTORY` (FU) (S): directory where core files are located. May be `.`. This directory is then recursively searched for core files. Note: this variable must be defined for any of the following to have any effect.
- `NVIM_TEST_CORE_GLOB_RE` (FU) (S): regular expression which must be matched by core files. E.g. `/core[^/]*$`. May be absent, in which case any file is considered to be matched.
- `NVIM_TEST_CORE_EXC_RE` (FU) (S): regular expression which excludes certain directories from searching for core files inside. E.g. use `^/%.deps$` to not search inside `/.deps`. If absent, nothing is excluded.
- `NVIM_TEST_CORE_DB_CMD` (FU) (S): command to get backtrace out of the debugger. E.g. `gdb -n -batch -ex "thread apply all bt full" "$_NVIM_TEST_APP" -c "$_NVIM_TEST_CORE"`. Defaults to the example command. This debug command may use environment variables `_NVIM_TEST_APP` (path to application which is being debugged: normally either nvim or luajit) and `_NVIM_TEST_CORE` (core file to get backtrace from).

- `NVIM_TEST_CORE_RANDOM_SKIP` (FU) (D): makes `check_cores` not check cores after approximately 90% of the tests. Should be used when finding cores is too hard for some reason. Normally (on OS X or when `NVIM_TEST_CORE_GLOB_DIRECTORY` is defined and this variable is not) cores are checked for after each test.
- `NVIM_TEST_RUN_TESTTEST` (U) (1): allows running `test/unit/testtest_spec.lua` used to check how testing infrastructure works.
- `NVIM_TEST_TRACE_LEVEL` (U) (N): specifies unit tests tracing level:
  - `0` disables tracing (the fastest, but you get no data if tests crash and there no core dump was generated),
  - `1` leaves only C function calls and returns in the trace (faster than recording everything),
  - `2` records all function calls, returns and executed Lua source lines.
- `NVIM_TEST_TRACE_ON_ERROR` (U) (1): makes unit tests yield trace on error in addition to regular error message.
- `NVIM_TEST_MAXTRACE` (U) (N): specifies maximum number of trace lines to keep. Default is 1024.