

---

# **Spintop OpenHTF**

**Tack Verification Inc.**

**Oct 23, 2020**



## CONTENTS:

<b>1</b>	<b>Getting Started</b>	<b>1</b>
1.1	Basic Concepts . . . . .	1
1.2	Python Installation . . . . .	3
<b>2</b>	<b>Reference</b>	<b>5</b>
2.1	Test Plan . . . . .	5
2.2	Configuration . . . . .	9
2.3	Plugs . . . . .	13
<b>3</b>	<b>1. First Testbench Tutorial</b>	<b>19</b>
3.1	Running a First Test Bench . . . . .	19
<b>4</b>	<b>2. Web Interface Tutorial</b>	<b>21</b>
4.1	Exploration of the Interface . . . . .	21
4.2	Exploration of Past Results . . . . .	23
<b>5</b>	<b>3. Forms and Tester Feedback Tutorial</b>	<b>27</b>
5.1	Using Custom Forms . . . . .	27
5.2	Extracting Data from the Custom Forms Responses . . . . .	29
5.3	Form Reference . . . . .	29
<b>6</b>	<b>4. Test Bench Definition Tutorial</b>	<b>35</b>
6.1	Trigger Phase . . . . .	35
6.2	Test Case Declaration . . . . .	39
6.3	Test Flow Management . . . . .	39
<b>7</b>	<b>5. Test Bench Documentation Tutorial</b>	<b>49</b>
7.1	Documenting a Test Case . . . . .	49
<b>8</b>	<b>6. Proposed Project Structure</b>	<b>51</b>
8.1	Test Bench Source Files Categories . . . . .	51
8.2	Proposed Single-Repository Structure . . . . .	53
8.3	Proposed Multiple-Repository Structure . . . . .	54
<b>9</b>	<b>7. Test Bench Configuration Tutorial</b>	<b>61</b>
9.1	Static configuration . . . . .	61
9.2	Test Station Configuration . . . . .	62
<b>10</b>	<b>8. Plugs Tutorial</b>	<b>65</b>
10.1	About Plugs . . . . .	65
10.2	Using Plugs . . . . .	65

10.3	Creating Plugs . . . . .	66
10.4	Wrapping spintop-openhtf Plugs . . . . .	67
<b>11</b>	<b>9. Test Criteria Tutorial</b>	<b>69</b>
11.1	Defining Test Criteria . . . . .	69
11.2	Criteria types . . . . .	70
11.3	Documentation . . . . .	71
11.4	Using a criteria definition file . . . . .	71
<b>12</b>	<b>10. Test Results Tutorial</b>	<b>73</b>
12.1	Exploring the Test Results . . . . .	73
12.2	Appending Data to Test Record . . . . .	76
<b>13</b>	<b>Indices and tables</b>	<b>77</b>
	<b>Python Module Index</b>	<b>79</b>
	<b>Index</b>	<b>81</b>

## GETTING STARTED

### 1.1 Basic Concepts

We first introduce the basic spintop-openhtf concepts with which test benches can be created.

#### 1.1.1 Test plan

In the context of a test bench implementation on spintop-openhtf, the test plan is the object to which the test phases are loaded and which executes them.

See *Running a First Test Bench*.

#### 1.1.2 Test phases

The test phases implement the different test cases. They are defined and loaded in the test plan object which executes them one after the other

See test-case-label.

#### 1.1.3 Test Sequences

The test sequences are intermediary levels of test phases between the test plan and the test cases. They can help implement complex test hierarchies.

See test-hierarchy-label.

#### 1.1.4 Trigger phase

The trigger phase refers to the first phase of the test bench, in which the dynamic configuration of the test is loaded. Such information can be for example:

- The operator name
- The board or system serial number
- The board or system device type
- The test type to execute on the board or system

See trigger-phase-label.

### 1.1.5 Test flow management

Test flow management refers to the dynamic selection of which test cases are executed depending on the inputs given at the beginning of the test bench in the trigger phase and by the results of the test themselves. Such inputs that determine test flow are for example the Device Under Test type and a FAIL result of a critical test.

See [test-flow-label](#).

### 1.1.6 Configuration

The configuration refers to all predetermined parameters used to control the flow or the execution of the test. The different configuration types are:

- The parameters configuring the test station itself, that is parameters changing from station to station and test jig to test jig, such as ip addresses, com port, etc.
- The parameters statically configuring the execution of the test plan, such as for example, the maximum number of iterations for a calibration algorithm.
- The parameters dynamically configuring the execution of the test plan, such as those gathered in the trigger phase.

See [static-config-label](#) and [teststation-config-label](#)

### 1.1.7 Forms

Forms are use to interact with the test operator. They permit the implementation of complex dialogs which allow to operator to both execute manual operations on the test jig to allow the test to continue or to input test result data for verification,

See [Using Custom Forms](#).

### 1.1.8 Plugs

In the spintop-openhtf context, plugs allow the interaction of the test logic with the surrounding test equipment. They basically wrap the access to the test equipment automation libraries.

See [About Plugs](#).

### 1.1.9 Criteria & measures

The criteria refer to the thresholds against which measures are compared to declare a test case PASS or FAIL. In the spintop-openhtf context, the measures module implements the criteria and the comparison against the selected values.

See [Defining Test Criteria](#).

### 1.1.10 Results

The spintop-openhtf framework outputs a standardized test result record for each test.

See results-label.

## 1.2 Python Installation

To install spintop-openhtf, you need Python. We officially support **Python 3.6+** for now. If you already have Python installed on your PC, you can skip this step. Officially supported OSes are:

- **Windows 10**

Install Python using the Windows Installer: <https://www.python.org/downloads/windows/>

- **Raspbian & Ubuntu**

Install through apt-get

### 1.2.1 IDE: VS Code with Extensions

We use and recommend Visual Studio Code for the development of spintop-openhtf testbenches. Using it will allow you to:

- Seamlessly debug with breakpoints the code you will be writing
- Remotely develop on a Raspberry Pi if you wish to
- Use a modern, extendable and free IDE

#### Installation

1. Download [VS Code](#)
2. Run and follow the installation executable
3. Install the [Python Extension](#)

### 1.2.2 Project Folder, Virtual Env & Installation (Windows)

First, create or select a folder where you want your sources to lie in.

Once you have your base python available, best practices are to create a virtualenv for each testbench you create. We will use `python` as the python executable, but if you installed a separate, non-path python 3 for example, you should replace that with your base executable.

Here are the installation steps on Windows:

1. Create Folder

```
mkdir myproject
cd myproject
```

2. Create venv

```
# Creates new venv in the folder 'venv'  
python -m venv venv  
venv\Scripts\activate
```

### 3. Install spintop-openhtf

```
python -m pip install spintop-openhtf[server]
```



## REFERENCE

## 2.1 Test Plan

**class** spintop\_openhtf.**TestPlan** (*name: str = 'testplan', store\_result: bool = True, store\_location: Union[str, Callable] = None*)

The core spintop-openhtf interface to create an openhtf sequence.

*TestPlan* simplifies the creating of openhtf sequences. Instead of declaring functions and adding them as an array to openhtf, the test plan allows declarative addition of function using decorators.

The TestPlan is itself a *TestSequence* and therefore implements all methods defined there.

In OpenHTF, you would do the following:

```
import openhtf as htf

@htf.plug(my_plug=...)
def example_test(test, my_plug):
    time.sleep(.2)

@htf.plug(my_plug=...)
def example_test2(test, my_plug):
    time.sleep(.2)

test = htf.Test(example_test, example_test2)
test.run() # Run once only
```

With the TestPlan, you can do the following (equivalent):

```
from spintop_openhtf import TestPlan

plan = TestPlan('my-test-name')

@plan.testcase('Test1')
@plan.plug(my_plug=...)
def example_test(test, my_plug):
    time.sleep(.2)

@plan.testcase('Test2')
@plan.plug(my_plug=...)
def example_test2(test, my_plug):
    time.sleep(.2)

plan.run_once()
```

**Parameters**

- **name** – The name of the test plan. Used to identify the ‘type’ of the test.
- **store\_result** – Whether results should automatically be stored as JSON in the spin-top\_openhtf site directory.
- **store\_location** – When store\_result is True, where test results will be stored. This can either be a callable function that receives the test record attributes as kwargs arguments (e.g. `fn(**test_record)`) or a string that will be formatted with the test record attributes (e.g. `{metadata[test_name]}/{outcome}`). This uses `LocalStorageOutput`, which instead of only writing the result as a JSON file as OpenHTF does, writes that json file as `result.json` in the folder and writes all test-attached files next to it.

**add\_callbacks** (*\*callbacks*)

Add custom callbacks to the underlying openhtf test.

**Parameters** **callbacks** – The callbacks to add.

**execute** ()

Execute the configured test using the `test_start` function as a trigger.

**property execute\_test**

Returns a function that takes no arguments and that executes the test described by this test plan.

**property history\_path**

The base path where results of the tests defined by this object are stored.

**image\_url** (*url: str*) → str

Creates a temporary hosted image based on the specified file path.

**Parameters** **url** – The image path.

**Returns** The temporary url associated with this image. Can be used in custom forms to show images.

**property is\_runnable**

Whether this test plan contains any runnable phases.

**no\_trigger** ()

Removes the need for a trigger and removes the default trigger.

```
plan = TestPlan('my-plan')
plan.no_trigger()

# ...

plan.run() # Will start immediately.
```

**run** (*launch\_browser=True, once=False*)

Run this test with the OpenHTF frontend.

Requires the [server] extra to work. If you do not need the server, you should use `run_console()` which disables completely the GUI server and increases execution speed therefore.

**Parameters**

- **launch\_browser** – When True, a browser page will open automatically when this is called.
- **once** – When False, the test will run in a loop; i.e. when a test ends a new one will start immediately.

**run\_console** (*once=False*)

Run this test without a frontend server and enables console input.

To enable the server, use `run()` instead.

**Parameters** *once* – When False, the test will run in a loop; i.e. when a test ends a new one will start immediately.

**run\_once** (*launch\_browser=True*)

Shortcut for `run()` with `once=True`.

**trigger** (*name: str, \*\*options*)

Decorator factory for the trigger phase.

Similar to `testcase()`, except that this function will be used as the test trigger.

The test trigger is a special test phase that is executed before test officially start. Once this phase is complete, the test will start. Usually used to configure the test with the DUT id for example.

**property trigger\_phase**

Returns the defined

**class** `spintop_openhtf.TestSequence` (*name*)

The base sequence object: defines a sequence of setup, test and teardown openhtf phases.

TestSequences can be nested together in two different ways:

**Option 1.** If the sequence should be re-usable, create an explicit `TestSequence` object and add `setup()`, `testcase()` and `teardown()` phases using the decorator methods. Afterwards, add it to a specific `TestPlan` (or sub `TestSequence`) using `append()`:

```
from spintop_openhtf import TestSequence, TestPlan

my_sequence = TestSequence('my-sequence')

@my_sequence.testcase('Test1')
def do_something(test):
    pass

# Elsewhere probably

plan = TestPlan('my-plan')
plan.append(my_sequence)
```

**Option 2.** If the sequence is defined simply to nest phases inside the same test plan, the `sub_sequence()` simplifies the declaration and usage of a sub sequence. This code block is equivalent to the previous one, but does not allow re-usage of the `sub_sequence` object:

```
from spintop_openhtf import TestPlan

plan = TestPlan('my-plan')

my_sequence = plan.sub_sequence('my-sequence')

@my_sequence.testcase('Test1')
def do_something(test):
    pass
```

Since `TestPlans` are `TestSequences`, and sub sequences are also `TestSequences`, they can be infinitely nested using `append()` and `sub_sequence()`.

**Parameters** *name* – The name of this test node.

**append** (\**phases*)

Append normal phases (or sequences) to this test plan.

**Parameters** *phases* – The phases to append.

**measures** (\**args*, **\*\*kwargs**)

Helper method: shortcut to `openhtf.measures()`

**plug** (\**args*, **\*\*kwargs**)

Helper method: shortcut to `openhtf.plug()`

**setup** (*name*, **\*\*options**)

Decorator factory for a setup function.

A setup function will be executed if the sequence is entered. All setup functions are executed before the testcases, regardless of the order of declaration.

See `testcase()` for usage.

**sub\_sequence** (*name*)

Create new empty TestSequence and append it to this sequence.

The following two snippets are equivalent:

```
my_sequence = TestSequence('Parent')
sub_sequence = my_sequence.sub_sequence('Child')
```

```
my_sequence = TestSequence('Parent')
sub_sequence = TestSequence('Child')
my_sequence.append(sub_sequence)
```

**teardown** (*name*, **\*\*options**)

Decorator factory for a teardown phase.

A teardown function will always be executed if the sequence is entered, regardless of the outcome of normal or setup phases.

See `testcase()` for usage.

**testcase** (*name*, **\*\*options**)

Decorator factory for a normal phase.

A testcase function is a normal openhtf phase.

The options parameter is a proxy for the PhaseOptions arguments.

```
my_sequence = TestSequence('Parent')

@my_sequence.testcase('my-testcase-name')
def setup_fn(test):
    (...)
```

Options used to override default test phase behaviors.

**name**

Override for the name of the phase. Can be formatted in several different ways as defined in `util.format_string`.

**timeout\_s**

Timeout to use for the phase, in seconds.

**run\_if**

Callback that decides whether to run the phase or not; if not run, the phase will also not be logged.

Optionally, this callback may take a single parameter: the `test.state` dictionary. This allows dynamic test selection based on variables in the user defined state.

**requires\_state**

If True, pass the whole `TestState` into the first argument, otherwise only the `TestApi` will be passed in. This is useful if a phase needs to wrap another phase for some reason, as `PhaseDescriptors` can only be invoked with a `TestState` instance.

**repeat\_limit**

Maximum number of repeats. None indicates a phase will be repeated infinitely as long as `PhaseResult.REPEAT` is returned.

**run\_under\_pdb**

If True, run the phase under the Python Debugger (pdb). When setting this option, increase the phase timeout as well because the timeout will still apply when under the debugger.

**Example Usages:** `@PhaseOptions(timeout_s=1) def PhaseFunc(test):`

`pass`

`@PhaseOptions(name='Phase({port})') def PhaseFunc(test, port, other_info):`

`pass`

## 2.2 Configuration

### 2.2.1 Config Module

A singleton class to replace the 'conf' module.

This class provides the configuration interface described in the module docstring. All attributes/methods must not begin with a lowercase letter so as to avoid naming conflicts with configuration keys.

**exception** `openhft.util.conf.ConfigurationInvalidError`

Indicates the configuration format was invalid or couldn't be read.

**class** `openhft.util.conf.Declaration`

Record type encapsulating information about a config declaration.

**exception** `openhft.util.conf.InvalidKeyError`

Raised when an invalid key is declared or accessed.

**exception** `openhft.util.conf.KeyAlreadyDeclaredError`

Indicates that a configuration key was already declared.

**exception** `openhft.util.conf.UndeclaredKeyError`

Indicates that a key was required but not predeclared.

**exception** `openhft.util.conf.UnsetKeyError`

Raised when a key value is requested but we have no value for it.

`openhft.util.conf.declare()`

Declare a configuration key with the given name.

**Parameters**

- **name** – Configuration key to declare, must not have been already declared.
- **description** – If provided, use this as the description for this key.

- **\*\*kwargs** – Other kwargs to pass to the Declaration, only `default_value` is currently supported.

`openhtf.util.conf.inject_positional_args()`

Decorator for injecting positional arguments from the configuration.

This decorator wraps the given method, so that any positional arguments are passed with corresponding values from the configuration. The name of the positional argument must match the configuration key.

Keyword arguments are *NEVER* modified, even if their names match configuration keys. Avoid naming keyword args names that are also configuration keys to avoid confusion.

Additional positional arguments may be used that do not appear in the configuration, but those arguments *MUST* be specified as keyword arguments upon invocation of the method. This is to avoid ambiguity in which positional arguments are getting which values.

**Parameters** `method` – The method to wrap.

**Returns** A wrapper that, when invoked, will call the wrapped method, passing in configuration values for positional arguments.

`openhtf.util.conf.load()`

load configuration values from kwargs, see `load_from_dict()`.

`openhtf.util.conf.load_flag_values()`

Load flag values given from command line flags.

**Parameters** `flags` – An argparse Namespace containing the command line flags.

`openhtf.util.conf.load_from_dict()`

Loads the config with values from a dictionary instead of a file.

This is meant for testing and bin purposes and shouldn't be used in most applications.

**Parameters**

- **dictionary** – The dictionary containing config keys/values to update.
- **\_override** – If True, new values will override previous values.
- **\_allow\_undeclared** – If True, silently load undeclared keys, otherwise warn and ignore the value. Typically used for loading config files before declarations have been evaluated.

`openhtf.util.conf.load_from_file()`

Loads the configuration from a file.

Parsed contents must be a single dict mapping config key to value.

**Parameters**

- **yamlfile** – The opened file object to load configuration from.
- **load\_from\_dict() for other args' descriptions.** (See) –

**Raises** `ConfigurationInvalidError` – If configuration file can't be read, or can't be parsed as either YAML (or JSON, which is a subset of YAML).

`openhtf.util.conf.load_from_filename()`

Opens the filename and calls `load_from_file`.

`openhtf.util.conf.reset()`

Reset the loaded state of the configuration to what it was at import.

Note that this does *not* reset values set by commandline flags or loaded from `–config-file` (in fact, any values loaded from `–config-file` that have been overridden are reset to their value from `–config-file`).

`openhtf.util.conf.save_and_restore()`

Decorator for saving conf state and restoring it after a function.

This decorator is primarily for use in tests, where conf keys may be updated for individual test cases, but those values need to be reverted after the test case is done.

## Examples

```
conf.declare('my_conf_key')
```

```
@conf.save_and_restore def MyTestFunc():
```

```
    conf.load(my_conf_key='baz') SomeFuncUnderTestThatUsesMyConfKey()
```

```
conf.load(my_conf_key='foo') MyTestFunc() print conf.my_conf_key # Prints 'foo', NOT 'baz'
```

# Without the `save_and_restore` decorator, `MyTestFunc()` would have had the # side effect of altering the conf value of `'my_conf_key'` to `'baz'`.

# Config keys can also be initialized for the context inline at decoration # time. This is the same as setting them at the beginning of the # function, but is a little clearer syntax if you know ahead of time what # config keys and values you need to set.

```
@conf.save_and_restore(my_conf_key='baz') def MyOtherTestFunc():
```

```
    print conf.my_conf_key # Prints 'baz'
```

```
MyOtherTestFunc() print conf.my_conf_key # Prints 'foo' again, for the same reason.
```

### Parameters

- **\_func** – The function to wrap. The returned wrapper will invoke the function and restore the config to the state it was in at invocation.
- **\*\*config\_values** – Config keys can be set inline at decoration time, see examples. Note that config keys can't begin with underscore, so there can be no name collision with `_func`.

**Returns** Wrapper to replace `_func`, as per Python decorator semantics.

## 2.2.2 Built-in Configuration Keys

```
class spintop_openhtf.testplan._default_conf.ConfigHelpText
```

**plug\_tearardown\_timeout\_s**

Timeout (in seconds) for each plug `tearDown` function if `> 0`; otherwise, will wait an unlimited time.

Default Value= 0

**allow\_unset\_measurements**

If True, unset measurements do not cause Tests to FAIL.

Default Value= False

**station\_id**

The name of this test station

Default Value= 'build-12175381-project-581279-spintop-openhtf'

**cancel\_timeout\_s**

Timeout (in seconds) when the test has been cancelled to wait for the running phase to exit.

Default Value= 2

**stop\_on\_first\_failure**

Stop current test execution and return Outcome FAIL on first phase with failed measurement.

Default Value= False

**capture\_source**

Whether to capture the source of phases and the test module. This defaults to False since this potentially reads many files and makes large string copies. If True, will capture docstring also. Set to 'true' if you want to capture your test's source.

Default Value= False

**capture\_docstring**

Whether to capture the docstring of phases and the test module. If True, will capture docstring. Set to 'true' if you want to capture your test's docstring.

Default Value= False

**teardown\_timeout\_s**

Default timeout (in seconds) for test teardown functions; this option is deprecated and only applies to the deprecated Test level teardown function.

Default Value= 30

**user\_input\_enable\_console**

If True, enables user input collection in console prompt.

Default Value= True

**frontend\_throttle\_s**

Min wait time between successive updates to the frontend.

Default Value= 0.15

**station\_server\_port**

Port on which to serve the app. If set to zero (the default) then an arbitrary port will be chosen.

Default Value= 0

**station\_discovery\_address**

(no description)

Default Value= None

**station\_discovery\_port**

(no description)

Default Value= None

**station\_discovery\_ttl**

(no description)

Default Value= None



## 2.3 Plugs

### 2.3.1 Base Interface

**class** spintop\_openhtf.plugs.base.UnboundPlug

A generic interface base class that allows intelligent creation of OpenHTF plugs without limiting its usage to OpenHTF.

**logger**

If inside an OpenHTF plug, this is the OpenHTF provided logger. If not, it is a logger with the object ID as name.

**classmethod** as\_plug(name, \*\*kwargs\_values)

Create a bound plug that will retrieve values from conf or the passed values here.

Take SSHInterface for example.

```
from spintop_openhtf.plugs import from_conf
from spintop_openhtf.plugs.ssh import SSHInterface

MySSHInterface = SSHInterface.as_plug(
    'MySSHInterface', # The name of this plug as it will appear in logs
    addr=from_conf( # from_conf will retrieve the conf value named like this.
        'my_ssh_addr',
        description="The addr of my device."
    ),
    username='x', # Always the same
    password='y' # Always the same
)
```

**close()**

Abstract method: Close resources related to this interface.

**close\_log()**

Close all *logger* handlers.

**log\_to\_filename**(filename, \*\*kwargs)

Create and add to *logger* a FileHandler that logs to filename.

**log\_to\_stream**(stream=None, \*\*kwargs)

Create and add to *logger* a StreamHandler that streams to stream

**open**(\*args, \*\*kwargs)

Abstract method: Open resources related to this interface.

**tearDown()**

Tear down the plug instance. This is part of the OpenHTF Plug contract

## 2.3.2 COM Port Interface

**class** spintop\_openhtf.plugs.comport.**ComportInterface** (*comport, baudrate=115200*)

An interface to a comport.

Allows reading and writing. A background thread reads any data that comes in and those lines can be accessed using the *next\_line* function.

**classmethod** **as\_plug** (*name, \*\*kwargs\_values*)

Create a bound plug that will retrieve values from conf or the passed values here.

Take SSHInterface for example.

```
from spintop_openhtf.plugs import from_conf
from spintop_openhtf.plugs.ssh import SSHInterface

MySSHInterface = SSHInterface.as_plug(
    'MySSHInterface', # The name of this plug as it will appear in logs
    addr=from_conf( # from_conf will retrieve the conf value named like this.
        'my_ssh_addr',
        description="The addr of my device."
    ),
    username='x', # Always the same
    password='y' # Always the same
)
```

**clear\_lines** ()

Clear all lines in the buffer.

**close** ()

Attempts to close the serial port if it exists.

**close\_log** ()

Close all logger handlers.

**com\_target** (\*args, \*\*kwargs)

Alias for message\_target

**connection\_lost** (*exc*)

Called when the connection is lost or closed.

The argument is an exception object or None (the latter meaning a regular EOF is received or the connection was aborted or closed).

**connection\_made** (*transport*)

Called when a connection is made.

The argument is the transport representing the pipe connection. To receive data, wait for data\_received() calls. When the connection is closed, connection\_lost() is called.

**data\_received** (*data*)

Called when some data is received.

The argument is a bytes object.

**eof\_received** ()

Called when the other end calls write\_eof() or equivalent.

If this returns a false value (including None), the transport will close itself. If it returns a true value, closing the transport is up to the protocol.

**execute\_command** (*command=None, timeout=None, target=None*)

Adds the self.eol to command and call message\_target using either target as target or self.\_target if defined.

This is used for executing commands in a shell-like environment and to wait for the prompt. `self._target` or `target` should be the expected prompt.

**keep\_lines** (*lines\_to\_keep*)

Clear all lines in the buffer except the last `lines_to_keep` lines.

**log\_to\_filename** (*filename*, *\*\*kwargs*)

Create and add to logger a `FileHandler` that logs to `filename`.

**log\_to\_stream** (*stream=None*, *\*\*kwargs*)

Create and add to logger a `StreamHandler` that streams to `stream`

**message\_target** (*message*, *target*, *timeout=None*, *keeps\_lines=0*, *\_timeout\_raises=True*)

Sends the message string and waits for any string in `target` to be received.

#### Parameters

- **message** – The string to write into the io interface
- **target** – A string or a list of string to check for in the read lines.
- **timeout** – (default=None, no timeout) Wait up to this seconds for the targets. If busted, this will raise a `IOTargetTimeout`.
- **keeps\_lines** – (default=0, discard all) Before sending the message, call `self.keep_lines` with this value.
- **\_timeout\_raises** – (default=True) If True, a timeout will raise an error.

**next\_line** (*timeout=10*)

Waits up to `timeout` seconds and return the next line available in the buffer.

**open** (*\_serial=None*)

Opens the serial port using the `comport` and `baudrate` object attributes.

**Parameters** **\_serial** – Optionnal underlying `serial.Serial` object to use. Used for mock testing.

**pause\_writing** ()

Called when the transport's buffer goes over the high-water mark.

Pause and resume calls are paired – `pause_writing()` is called once when the buffer goes strictly over the high-water mark (even if subsequent writes increases the buffer size even more), and eventually `resume_writing()` is called once when the buffer size reaches the low-water mark.

Note that if the buffer size equals the high-water mark, `pause_writing()` is not called – it must go strictly over. Conversely, `resume_writing()` is called when the buffer size is equal or lower than the low-water mark. These end conditions are important to ensure that things go as expected when either mark is zero.

NOTE: This is the only Protocol callback that is not called through `EventLoop.call_soon()` – if it were, it would have no effect when it's most needed (when the app keeps writing without yielding until `pause_writing()` is called).

**resume\_writing** ()

Called when the transport's buffer drains below the low-water mark.

See `pause_writing()` for details.

**tearDown** ()

Tear down the plug instance.

**write** (*string*)

Write the string into the io interface.

### 2.3.3 SSH Interface

```
class spintop_openhtf.plugs.ssh.SSHInterface(addr, username, password, create_timeout=3, port=22)
```

An interface to an SSH Server.

```
class SSHResponse(exit_code: int, err_output: str, std_output: str)
```

```
err_output: str = None
```

The command stderr output

```
exit_code: int = None
```

The command exit code

```
property output
```

Combines both the err output and the std\_output.

```
std_output: str = None
```

The command stdout output

```
classmethod as_plug(name, **kwargs_values)
```

Create a bound plug that will retrieve values from conf or the passed values here.

Take SSHInterface for example.

```
from spintop_openhtf.plugs import from_conf
from spintop_openhtf.plugs.ssh import SSHInterface

MySSHInterface = SSHInterface.as_plug(
    'MySSHInterface', # The name of this plug as it will appear in logs
    addr=from_conf( # from_conf will retrieve the conf value named like this.
        'my_ssh_addr',
        description="The addr of my device."
    ),
    username='x', # Always the same
    password='y' # Always the same
)
```

```
close()
```

Abstract method: Close resources related to this interface.

```
close_log()
```

Close all logger handlers.

```
execute_command(command: str, timeout: float = 60, stdin: List = [], get_pty: bool = False, assert_exitcode: Union[List[int], int, None] = 0)
```

Send a command and wait for it to execute.

#### Parameters

- **command** – The command to send. End of lines are automatically managed. For example `execute_command('ls')` will executed the ls command.
- **timeout** – The timeout in second to wait for the command to finish executing.
- **stdin** – A list of inputs to send into stdin after the command is started. Each entry in this list will be separated with an `r'n'` character.
- **get\_pty** – Usually required when providing `stdin`.

- **assertexitcode** – Unless this is None, defines one or a list of exit codes that are expected. After the command is executed, an `SSHError` will be raised if the exit code is not as expected.

#### Raises

- **SSHTimeoutError** – Raised when `timeout` is reached.
- **SSHError** – Raised when the exit code of the command is not in `assertexitcode` and `assertexitcode` is not None.

**log\_to\_filename** (*filename*, *\*\*kwargs*)

Create and add to logger a `FileHandler` that logs to `filename`.

**log\_to\_stream** (*stream=None*, *\*\*kwargs*)

Create and add to logger a `StreamHandler` that streams to `stream`

**open** (*\_client=None*)

Abstract method: Open resources related to this interface.

**tearDown** ()

Tear down the plug instance. This is part of the OpenHTF Plug contract

## 2.3.4 Telnet Interface

**class** `spintop_openhtf.plugs.telnet.TelnetInterface` (*addr*, *port=4343*)

**classmethod** **as\_plug** (*name*, *\*\*kwargs\_values*)

Create a bound plug that will retrieve values from `conf` or the passed values here.

Take `SSHInterface` for example.

```
from spintop_openhtf.plugs import from_conf
from spintop_openhtf.plugs.ssh import SSHInterface

MySSHInterface = SSHInterface.as_plug(
    'MySSHInterface', # The name of this plug as it will appear in logs
    addr=from_conf( # from_conf will retrieve the conf value named like this.
        'my_ssh_addr',
        description="The addr of my device."
    ),
    username='x', # Always the same
    password='y' # Always the same
)
```

**close** ()

Abstract method: Close resources related to this interface.

**close\_log** ()

Close all logger handlers.

**execute\_command** (*command: str*, *expected\_output: str = ""*, *until\_timeout: float = 5*)

Send a `command` and wait for it to execute.

#### Parameters

- **command** (*str*) – The command to send. End of lines are automatically managed. For example `execute_command('ls')`
- **executed the ls command.** (*will*) –

- **expected\_output** (*str*, *optional*) – If not none, the command executor will read until a given string, `expected_output`, is encountered or until `until_timeout` (s) have passed.
- **until\_timeout** (*float*, *optional*) – The timeout in seconds to wait for the command to finish reading output. Defaults to 5.

**Returns** output response from the Telnet client

**Return type** output (str)

**Raises** **TelnetError** – Raised when reading the output goes wrong.

**log\_to\_filename** (*filename*, *\*\*kwargs*)

Create and add to logger a `FileHandler` that logs to `filename`.

**log\_to\_stream** (*stream=None*, *\*\*kwargs*)

Create and add to logger a `StreamHandler` that streams to `stream`

**open** (*\_client=None*)

Abstract method: Open resources related to this interface.

**tearDown** ()

Tear down the plug instance. This is part of the OpenHTF Plug contract

## 1. FIRST TESTBENCH TUTORIAL

### 3.1 Running a First Test Bench

Let's create our first testbench and explore the basic concept of the test plan.

Create a file called `main.py` in the folder in which you installed `spintop-openhtf` and copy this code.

#### 3.1.1 Basic Testbench

```
# main.py
from openhtf.plugs.user_input import UserInput
from spintop_openhtf import TestPlan

""" Test Plan """

# This defines the name of the testbench.
plan = TestPlan('hello')

@plan.testcase('Hello-Test')
@plan.plugin(prompts=UserInput)
def hello_world(test, prompts):
    prompts.prompt('Hello Operator!')
    test.dut_id = 'hello' # Manually set the DUT Id to same value every test

if __name__ == '__main__':
    plan.no_trigger()
    plan.run()
```

In the code above, a test plan is first declared.

```
plan = TestPlan('hello')
```

Then, a test case is declared and added to the test plan.

```
@plan.testcase('Hello-Test')
@plan.plugin(prompts=UserInput)
def hello_world(test, prompts):
    prompts.prompt('Hello Operator!')
    test.dut_id = 'hello' # Manually set the DUT Id to same value every test
```

And finally the test plan is executed when the script is launched.

```
if __name__ == '__main__':  
    plan.no_trigger()  
    plan.run_console()
```

This simple test bench will simply interact with the operator by telling him Hello Operator!. Run it using the created virtual environment :

```
# Windows Activate  
venv\Scripts\activate  
  
python main.py
```

This test bench does not use the spintop-openhtf GUI therefore all interactions are made through the command line. The test prints Hello Operator! and indicates a PASS.

```
(venv) C:\GIT_TACK\doc_exp>python main.py  
Hello Operator!  
-->  
  
===== test: hello  outcome: PASS =====
```

Tutorial source



## 2. WEB INTERFACE TUTORIAL

spintop-openhtf uses a web interface to interact with the test operator.


- It guides him or her through the manual operations to perform on the test bench.
- It displays the logs and the results of all test phases as well as for the complete test bench.

### 4.1 Exploration of the Interface

Let's explore the web application. To do so, make sure you have gone through the *Running a First Test Bench* tutorial. Modify it, replacing the main to run with the web application.

```
if __name__ == '__main__':  
    plan.no_trigger()  
    plan.run()
```

Run the new testbench. The web interface should load automatically in your default browser. The following image indicates the different sections of the web interface.



A spintop implementation using OpenHTF, the Hardware Testing Framework.

[Refresh station](#)

P089

Status: Connected  
localhost:4444

Operator input

Hello Operator! 1

2 OKAY

Current test: hello (4m 40s) 3 Running

DUT: — Started: Mar 11, 2020, 9:06:15 PM

Ran 0 of 1 phases

Current phase: Hello-Test : (4s) 4 Running

Phases EXPAND ALL

Hello-Test : (12m 50s) Running

Logs EXPAND 5

9:06:15 PM openhtf.plugins.user\_input Displaying prompt (da82fa3cd69d49189715365a169430e8): "Hello Operator!"

Not showing 4 additional log messages.

Attachments

There are no attachments yet.

History

History from disk is disabled.

1. The Operator Input section is where the prompts and forms are showcased. The Hello Operator! prompt defines in the test bench appears.
2. The test will continue (and end) when the tester clicks on the Okay button.
3. The test is shown as running with the elapsed time displayed in real time.
4. The current phase being executed is displayed. The current testbench holds only one test phase. In more complex test plans, the executed and to be executed phases are displayed to facilitate the operator following the test.
5. The test logs list the different operations executed by the test bench as well as the information that was decided to be logged by the test developer.

22

Chapter 4. 2. Web Interface Tutorial

## 4.2 Exploration of Past Results

Run the test a few more times, clicking **OKAY** again and again until 10 tests are executed. Notice that the web application logs the past 5 test results on the right of the interface. T

**Operator input**

Hello Operator!

OKAY

**Current test: hello (7s)**

Running

DUT: — Started: Apr 12, 2020, 3:21:33 PM

Ran 0 of 1 phases

**Current phase: Hello-Test : (7s)**

Running

**Phases**

EXPAND ALL

**Hello-Test : (7s)**

Running

**Logs**

EXPAND

**History**

EXPAND

hello Started —	Pass
hello Started —	Pass
hello Started —	Pass
hello Started —	Pass
hello Started —	Pass

Not showing 5 additional test runs.

History from disk is disabled.

Hit the **Expand** button on the **History** box to see all past test results.

History	COLLAPSE
hello Started —	Pass
hello Started —	Pass
hello Started —	Pass
hello Started —	Pass
hello Started —	Pass
hello Started —	Pass
hello Started —	Pass
hello Started —	Pass
hello Started —	Pass
History from disk is disabled.	

To access the full results of any of the tests on the list, click on the test you want to consult. The web application will load the result and display it, phases, logs and everything.

Displaying test record for a previous test run (—)

RETURN TO CURRENT TEST

Current test: hello (1s)

Pass

DUT: hello

Started: Apr 12, 2020, 3:21:28 PM

Ran 1 of 1 phase

Phases

EXPAND ALL

Hello-Test: (1s)

Pass

Logs

EXPAND

3:21:29 PM  
openhtf.core.test\_descriptor

Test completed for hello, outputting now.

Not showing 12 additional log messages.

History

COLLAPSE

hello Started —	Pass
hello Started —	Pass
hello Started —	Pass
hello Started —	Pass
hello Started —	Pass
hello Started —	Pass
hello Started —	Pass
hello Started —	Pass
hello Started —	Pass
hello Started —	Pass
hello Started —	Pass

Tutorial source



## 3. FORMS AND TESTER FEEDBACK TUTORIAL

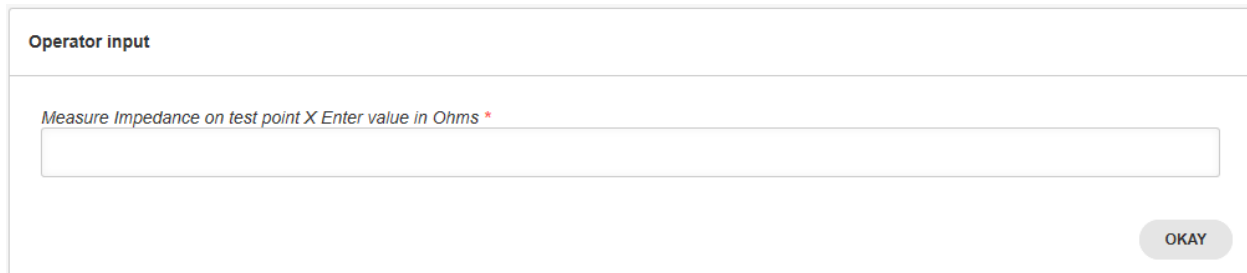
### 5.1 Using Custom Forms

#### 5.1.1 Introduction

Customizable forms allows spintop-openhtf developers to include complex form inputs in their test plans in order to interact with the test operators using simple dictionary definitions. For example, the following form shows an input field, allowing the tester to enter the measured impedance.

```
FORM_LAYOUT = {
  'schema': {
    'title': "Impedance",
    'type': "object",
    'required': ["impedance"],
    'properties': {
      'impedance': {
        'type': "string",
        'title': "Measure Impedance on test point X\nEnter value in Ohms"
      },
    },
  },
  'layout': [
    "impedance"
  ]
}
```

When executed the above form entry will result in the following being displayed on the web interface.



Operator input

Measure Impedance on test point X Enter value in Ohms \*

OKAY

This `FORM_LAYOUT` variable contains two top level attributes which are essential to differentiate: `schema` and `layout`. The `schema` defines the fields while `layout` defines the ordering and display of these fields.

### 5.1.2 JSON Schema Forms

The schema part is actually a generic JSON schema vocabulary used to validate JSON documents named *\*JSON Schema\**.

As quoted from their homepage,

JSON Schema is a vocabulary that allows you to annotate and validate JSON documents.

#### Example Schema

The format allows you to build the schema of complex JSON structures and afterward validate that a certain JSON document respects or not that structure. Let's take the schema part of our previous example:

```
{
  'title': "Impedance",
  'type': "object",
  'required': ["impedance"],
  'properties': {
    'impedance': {
      'type': "string",
      'title': "Measure Impedance on test point X\nEnter value in Ohms"
    },
  },
}
```

In order, this defines that:

- "title": "Impedance": The title of this form is 'Impedance'.
- "type": "object": The type of the top level JSON object is object, which means that it contains other properties.
- "required": ["impedance"]: The property named `impedance` is required.
- "properties": {}: Begins the list of properties in this object. Note that these are *unordered*.
- "impedance": { "type": "string", "title": "Measure Impedance on test point X\nEnter value in Ohms" }

The property named `impedance` is a string with as label the instructions passed to the operator :  
'Measure Impedance on test point XnEnter value in Ohms'

- And so on.

---

**Note:** Of all the keys shown here, only *required* is specific to JSON Schema *Forms*. The rest is part of the JSON Schema format. You can use the following playground to experiment with the forms: [JSON Schema Form](#). However, the renderer is not the same that spintop-openhtf internally uses and therefore results may vary. You can use the getting started example in the spintop-openhtf repo for a quick demo with forms.

---

To add the custom form to the test bench defined previously in the [Running a First Test Bench](#) tutorial, first insert the FORM\_LAYOUT definition at the top of the main.py file, then modify the test case definition to use the new custom form prompts as shown below.

```
@plan.testcase('Hello-Test')
@plan.plugin(prompts=UserInput)
def hello_world(test, prompts):
```

(continues on next page)



(continued from previous page)

```
"""Displays the custom form defined above"""
prompts.prompt_form(FORM_LAYOUT)
```

Run the testbench again to see the new form appear.

[Tutorial source](#)

## 5.2 Extracting Data from the Custom Forms Responses

The custom forms output a response in the form of a dictionary when the form display returns. The response dictionary is returned once the operator hits the Okay button.

Operator input

Measure Impedance on test point X Enter value in Ohms \*

OKAY

To gather the response from the form, simply create a ‘response’ variable to receive the prompt return, such as illustrated below.

```
@plan.testcase('Hello-Test')
@plan.plugin(prompts=UserInput)
def hello_world(test, prompts):
    """Displays the custom form defined above"""
    response = prompts.prompt_form(FORM_LAYOUT)
```

In the case where the operator enters a value of 1200 Ohms as impedance, the dictionary returned by the will be the following:

```
{'impedance': '1200'}
```

To access the returned value, simply index the dictionary with the key used in the form definition:

```
test.impedance = response['impedance']
```

To define different types of custom forms to receive types different responses, refer to the [Form Reference](#) article.

[Tutorial source](#)

## 5.3 Form Reference

This page lists the different form types that can be used in a spintop-openhtf test bench.

### 5.3.1 Example Data

The previous form would then successfully validate the following JSON Data:

```
{
  "firstname": "foo",
  "lastname": "bar"
}
```

This is the dictionary that is returned when you call `TextInput.prompt_form(...)`.

### 5.3.2 Layout

The layout aspect of our previous example is specific to JSON Schema Forms, and, more specifically, [to the renderer we use](#).

#### Select Choices (Dropdown)

If we re-use the previous form and wish to limit the values allowed for a specific string field, we can use the layout attribute to impose a select field.

In the following snippet, the simple `lastname` key is replaced by a complex object which identifies the field using the `"key": "lastname"` attribute. By adding the `"type": "select"` with the `titleMap`, we impose specific choices to the user.

This does not make much sense in the case of a last name, but we use the same example for consistency.

```
FORM_LAYOUT = {
  'schema': {
    'title': "First and Last Name",
    'type': "object",
    'required': ["firstname", "lastname"],
    'properties': {
      'firstname': {
        'type': "string",
        'title': "First Name"
      },
      'lastname': {
        'type': "string",
        'title': "Last Name"
      },
    },
  },
  'layout': [
    "firstname",
    {
      "key": "lastname",
      "type": "select",
      "titleMap": [
        { "value": "Andersson", "name": "Andersson" },
        { "value": "Johansson", "name": "Johansson" },
        { "value": "other", "name": "Something else..." }
      ]
    }
  ]
}
```

First Name \*

Last Name \*

Andersson  
Johansson  
Something else...

SUBMIT

## Radio Buttons

Same example with lastname:

```
FORM_LAYOUT = {
  'schema': {
    'title': "First and Last Name",
    'type': "object",
    'required': ["firstname", "lastname"],
    'properties': {
      'firstname': {
        'type': "string",
        'title': "First Name"
      },
      'lastname': {
        'type': "string",
        'title': "Last Name"
      },
    },
  },
  'layout': [
    "firstname",
    {
      "key": "lastname",
      "type": "radiobuttons",
      "titleMap": [
        { "value": "one", "name": "One" },
        { "value": "two", "name": "More..." }
      ]
    }
  ]
}
```

First Name \*

Last Name \*

☐ One
☒ More...

SUBMIT

### Text

Adding text within the form is very useful to guide or otherwise give more information to the user. This can be done using the "type": "help" layout.

---

**Note:** The *markdown* function was added in spintop-openhtf version 0.5.5. It transforms the text into HTML, which is the only understood format of the helpvalue.

---

```
from spintop_openhtf.util.markdown import markdown

FORM_LAYOUT = {
    'schema': {
        'title': "First and Last Name",
        'type': "object",
        'required': ["firstname", "lastname"],
        'properties': {
            'firstname': {
                'type': "string",
                'title': "First Name"
            },
            'lastname': {
                'type': "string",
                'title': "Last Name"
            },
        },
    },
    'layout': [
        "firstname",
        {
            "type": "help",
            "helpvalue": markdown("# Well Hello There!")
        },
        "lastname"
    ]
}
```

First Name \*

Well Hello There!

Last Name \*

SUBMIT

## Images

To seamlessly serve one or more image in your custom form or prompt message, the test plan `image_url` method needs to be used. This will create a temporary url that points to the local file you are targeting and allow browsers to load this image successfully.

**Warning:** The url returned by `image_url` is strictly temporary. It represents an in-memory mapping between the url and the filepath you specified. It follows the lifecycle of the TestPlan object, which means that as long as you keep the same test plan object, the url will live.

There are no cleanup mechanisms. However, each image is a simple key: value entry in a dictionary, which means that its memory footprint is negligible.

```
from spintop_openhtf.util.markdown import markdown, image_url

plan = TestPlan('examples.getting_started')

helpvalue = markdown("""
# Well Hello There


"" % plan.image_url('spinhub-app-icon.png'))

FORM_LAYOUT = {
    'schema': {
        'title': "First and Last Name",
        'type': "object",
        'required': ["firstname", "lastname"],
        'properties': {
            'firstname': {
                'type': "string",
                'title': "First Name"
            },
            'lastname': {
                'type': "string",
                'title': "Last Name"
            },
        },
    },
    'layout': [
        "firstname",
        {
            "type": "help",
            "helpvalue": helpvalue
        },
        {
            "key": "lastname",
            "type": "radiobuttons",
            "titleMap": [
                { "value": "one", "name": "One" },
                { "value": "two", "name": "More..." }
            ]
        }
    ]
}
```

(continues on next page)

(continued from previous page)

```
} 1
```

First Name \*

Well Hello There



Last Name \*

☐ One ☐ More...

SUBMIT

## 4. TEST BENCH DEFINITION TUTORIAL

### 6.1 Trigger Phase

The trigger phase is the first phase executed in most test benches. It is a test phase which is outside of the test execution, meaning it cannot fail and does not consume test execution time. The trigger phase is added by default by spintop-openhtf. However it was disabled in our first example in the tutorial.

#### 6.1.1 Default Trigger Phase

Let's start the trigger phase experimentation by removing the trigger phase disable in the test bench main. Replace the main from the *Running a First Test Bench* tutorial example by the one below and run the bench again.

```
if __name__ == '__main__':  
    plan.run()
```

The web interface now displays the default trigger phase, asking for a DUT id to start the test.

P089 Status: Connected  
localhost:4444

Operator input

Enter a DUT ID in order to start the test. \*

1

OKAY

Current test: hello 2 Waiting

DUT: — Started: —

Ran 0 of 2 phases

Phases 3 EXPAND ALL

trigger_phase : Test start trigger that prompts the user for a DUT ID. (1m 23s)	Running
Hello-Test : Displays Hello Operator in operator prompt and waits for Okay	Waiting

History

History from disk is disabled.

1. The DUT id is asked for in the operator input dialog.
2. The current test is displayed as waiting.
3. The Phases dialog now displays the trigger phase as well as our hello-test phase.

Enter DUT1 as the DUT id and press OK. The test will continue to the hello-test phase and display the Hello Operator! prompt.

Current test: hello (5s)		Running
DUT: DUT1	1	Started: Mar 12, 2020, 10:31:12 PM
Ran 1 of 2 phase		
Current phase: Hello-Test : Displays Hello Operator in operator prompt and waits fo...		Running
Phases <span>EXPAND ALL</span>		
trigger_phase : Test start trigger that prompts the user for a DUT ID. (6s)	2	Pass
Hello-Test : Displays Hello Operator in operator prompt and waits for Okay (5s)		Running

1. The DUT id is displayed.
2. The trigger phase is marked as executed and passed.

Tutorial source

### 6.1.2 Custom Trigger Phase

With the custom forms, it is possible to define a custom trigger phase to create a specific dynamic configuration for your test bench.

Let's first define what our trigger phase will do. The following information will be asked from the operator:

- The full operator name from a drop down list
- The Device Under Test serial number
- The tested product from a drop down list

The form is defined as the following:

```
FORM_LAYOUT = {
  'schema': {
    'title': "Test configuration",
    'type': "object",
    'required': ["operator", "utid", "product"],
    'properties': {
```

(continues on next page)



(continued from previous page)

```

        'operator': {
            'type': "string",
            'title': "Enter the operator name"
        },
        'dutid': {
            'type': "string",
            'title': "Enter the device under test serial number"
        },
        'product': {
            'type': "string",
            'title': "Enter the product name"
        }
    },
    'layout': [
        "operator", "dutid", "product",
    ]
}

```

To call the form in the trigger phase, define a new test case, but instead of using the `plan.testcase` decorator, use the `plan.trigger` one. In the example below, we modified the test case used in the [Extracting Data from the Custom Forms Responses](#) tutorial to define a configuration phase to be used as a trigger, using the custom form defined above.

```

@plan.trigger('Configuration')
@plan.plugin(prompts=UserInput)
def trigger(test, prompts):
    """Displays the configuration form"""
    response = prompts.prompt_form(FORM_LAYOUT)
    pprint(response)

```

Since the phase using the custom form is now the trigger, a new test case must be defined to implement the test bench. A simple sleep test is added for this purpose.

```

from time import sleep

@plan.testcase('Sleep')
def sleep_test(test):
    """Waits five seconds"""
    sleep(5)

```

To run the test plan with the new trigger phase, remove the `plan.no_trigger()` call.

```

if __name__ == '__main__':
    plan.run()

```

Now run the new test plan. The test will display the following form:

**Operator input**

*Enter the operator name*

*Enter the device under test serial number*

*Enter the product name*

OKAY

Enter the operator name, the device serial number and the product ID and hit okay. The sleep test will wait 5 seconds and terminate. The status of the two phases can be seen in the web interface.

**Current test: hello (2s)**
Running

DUT: —
Started: Mar 16, 2020, 9:28:23 PM

Ran 1 of 2 phase

**Current phase: Sleep : Waits five seconds (2s)**
Running

**Phases**
EXPAND ALL

**Configuration : Displays the custom from defined above (10s)**
Pass

**Sleep : Waits five seconds (2s)**
Running

As the test bench executes, the following verbose will be outputted on the command line.

```
W 01:14:11 test_executor - Start trigger did not set a DUT ID.
W 01:14:16 test_executor - DUT ID is still not set; using default.
```

This occurs because the trigger phase is designed in part to enter the DUT ID and log it in the test variables. The form has indeed asked for the DUT ID but has not logged it. Add the following line at the end of the trigger phase and rerun the test bench.

```
response = prompts.prompt_form(FORM_LAYOUT)
test.dut_id = response['dutid']
```

The missing dut id verbose is gone and the id has been logged in the test variables.

[Tutorial source](#)

## 6.2 Test Case Declaration

The different test cases are defined and declared one by one in the test plan. Let's review the test case declared in the *Running a First Test Bench* tutorial.

The test case is declared as:

```
@plan.testcase('Hello-Test')
@plan.plugin(prompts=UserInput)
def hello_world(test, prompts):
    prompts.prompt('Hello Operator!')
    test.dut_id = 'hello' # Manually set the DUT Id to same value every test
```

spintop-openhtf uses the name “test phase” to refer to the different test cases in the test bench.

## 6.3 Test Flow Management

In the context of a spintop-openhtf test bench, test flow management consists in selecting the test cases to execute dynamically depending on

- the input of the operator during the trigger phase,
- the results and outcomes of the previous test phases.

A test bench can be implemented for a family of product instead of one test bench per product version. In these test benches, the DUT type selected in the trigger phase will determine which test cases are executed. For example, for a version, all test cases can be run and for another, a test is removed. Test flow management allows the definition of such test benches.

### 6.3.1 Dynamic Test Flow Management with Phase Outcomes

The outcome and result of a phase can impact the execution of the rest of the test benches. Let's first explore the concepts of phase outcomes and phase results.

#### Phase results

The result of a test phase is controlled by the value it returns. The developer can determine through the test logic which result is returned.

It can return None (or no return statement) for the standard CONTINUE. If an uncaught exception occurs during the test, the phase will be marked as ERROR. OpenHTF will infer the outcome based on the test phase result.

- **PhaseResult.CONTINUE:** Causes the framework to process the phase measurement outcomes and execute the next phase.
- **PhaseResult.FAIL\_AND\_CONTINUE:** Causes the framework to mark the phase with a fail outcome and execute the next phase.
- **PhaseResult.REPEAT:** Causes the framework to execute the same phase again, ignoring the measurement outcomes for this instance.
- **PhaseResult.SKIP:** Causes the framework to ignore the measurement outcomes and execute the next phase.
- **PhaseResult.STOP:** Causes the framework to stop executing, indicating a failure. The next phases will not be executed.

## Phase outcomes

The outcome of a test phase can be one of the following values, which is determined by the `PhaseResult` described above:

- **PhaseOutcome.PASS:** The phase result was `CONTINUE` and all measurements validators passed.
- **PhaseOutcome.FAIL:** The phase result was `CONTINUE` and one or more measurements outcome failed, or the phase result was `FAIL_AND_CONTINUE` or `STOP`.
- **PhaseOutcome.SKIP:** The phase result was `SKIP` or `REPEAT`.
- **PhaseOutcome.ERROR:** An uncaught exception was raised, and that exception is not part of the `failure_exceptions` list. The test will stop in such a case.

Although you should not need to import this enum for test sequencing, you can import it using `from openhtf.core.test_record import PhaseOutcome`. Usage would be to analyze results.

Phase outcomes are inherited through all levels by the parent phase up to the test plan itself.

## Difference between outcome and result

Simply put,

- the phase outcome determines if the phase has failed or passed, and is propagated to all parent phases.
- the phase result first creates the phase outcome and then dictates the remaining test flow.

The phase results impact on the test flow is discussed and exemplified in details below.

## 6.3.2 Managing the Test Flow Based on Phase Results

The phase results can be used by the test bench developer to manipulate the flow of the test bench based on intermediary results.

### PhaseResult.STOP

The **PhaseResult.STOP** result can be imposed by the developer in the case of the failure of a critical test. Tests are critical when continuing the test after a failure could be dangerous for the unit under test and the test bench. The developer could also decide to terminate the test on a failure when continuing would be a waste of time. The flow management based on phase results allows the termination of the test bench on such a critical phase failure.

### PhaseResult.FAIL\_AND\_CONTINUE

The **PhaseResult.FAIL\_AND\_CONTINUE** result is used to mark a phase as failed when no other element of the test would indicate it (criteria or exceptions). The developer uses it as the return code.

As an example, it is decided to return a failure in product “B” is selected in the trigger phase.

```
from openhtf import PhaseResult

@plan.testcase('Sleep')
def sleep_test(test):
    """Waits five seconds"""
    sleep(5)
    if test.state["product"] == "A":
```

(continues on next page)

(continued from previous page)

```

    return PhaseResult.CONTINUE
elif test.state["product"] == "B":
    return PhaseResult.FAIL_AND_CONTINUE

```

## PhaseResult.REPEAT

The **PhaseResult.REPEAT** result can be used to retry non-critical tests. Some tests can be retried until they pass (with a maximum retry number), without endangering the quality of the product. For example a calibration algorithm that converges through multiple tries.

In the case the developer requires a repeat of the test for it to converge to a PASS, the return code is used. The *repeat\_limit* option is used to limit the number of retries. If the **PhaseResult.REPEAT** is returned once more than the phase's *repeat\_limit*\*, this will be treated as a **PhaseResult.STOP**.

As an example, we create a test which generates a random number between 1-10 and declares a PASS if the result is 8 or higher. The test will fail 70% of the time, but will be repeated until it passes, if the PASS appears within 5 retries.

```

from openhtf import PhaseResult
import random

@plan.testcase('Random')
def random_test(test, repeat_limit = 5):
    """Generate a random number between 1 and 10. If number is 8, 9, or 10 it is a
    ↪PASS. If not repeat"""
    val = random.randint(1, 10)
    print (val)
    if val >= 8:
        return PhaseResult.CONTINUE
    else:
        return PhaseResult.REPEAT

```

The following is an excerpt of a log of a 5 retries test

```

Handling phase Random
Executing phase Random
3
Thread finished: <PhaseExecutorThread: (Random)>
Phase Random finished with result PhaseResult.REPEAT
Executing phase Random
2
Thread finished: <PhaseExecutorThread: (Random)>
Phase Random finished with result PhaseResult.REPEAT
Executing phase Random
3
Thread finished: <PhaseExecutorThread: (Random)>
Phase Random finished with result PhaseResult.REPEAT
Executing phase Random
7
Thread finished: <PhaseExecutorThread: (Random)>
Phase Random finished with result PhaseResult.REPEAT
Executing phase Random
9
Thread finished: <PhaseExecutorThread: (Random)>
Phase Random finished with result PhaseResult.CONTINUE

```

## PhaseResult.SKIP

The **PhaseResult.SKIP** result can be used by the developer to ignore a phase, depending on what goes on inside. Let's merge our last two examples, to create a test where, if product "A" is entered, the random test is executed and logged, and if product "B" is entered, the random test is executed but its result is ignored. No repeats are allowed.

```
from openhtf import PhaseResult
import random
@plan.testcase('Random')
def random_test(test):
    """Generate a random number between 1 and 10.

    If number is 8, 9, or 10 it is a PASS. If not repeat"""
    val = random.randint(1, 10)
    print (val)
    if test.state["product"] == "A":
        if val >= 8:
            return PhaseResult.CONTINUE
        else:
            return PhaseResult.FAIL_AND_CONTINUE
    elif test.state["product"] == "B":
        return PhaseResult.SKIP
```

The following are two log excerpts. The first one for product "A", the second for product "B".

```
Executing phase Random 2
7
Thread finished: <PhaseExecutorThread: (Random 2)>
Phase Random 2 finished with result PhaseResult.FAIL_AND_CONTINUE
```

```
Executing phase Random 2
7
Thread finished: <PhaseExecutorThread: (Random 2)>
Phase Random 2 finished with result PhaseResult.SKIP
```

## Interpreting exceptions as failures

Normally, exceptions are caught by spintop-openhtf which translates them to a **PhaseOutcome.ERROR** outcome. To identify certain exceptions as a FAIL instead of as an error, you can add failure exceptions to the test plan.

```
test_plan = TestPlan()
test_plan.failure_exceptions += (Exception,)

@test_plan.testcase('test1')
def my_test(test):
    raise Exception('!!!') # Will mark phase as FAIL instead of ERROR.
```

Tutorial source

### 6.3.3 Test Hierarchy

To build comprehensive test benches it is important to define a test hierarchy. We have already explored the declaration of a *test case* within a *test plan*, which creates a 2-level hierarchy. As we have defined our test benches, the *test plan* inherits the status of the underlying *test cases*. If a *test case* fails, the *test plan* fails. The test bench does not need to remain a 2-level hierarchy. The *test plan* can be comprised of complex *test sequences* which in turn are comprised of *sub-sequences*, *testcases* and so on.

```

Test Plan
├── Sequence 1
│   ├── Sub-sequence 1A
│   │   ├── Testcase 1A-1
│   │   ├── Testcase 1A-2
│   │   └── Testcase 1A-3
│   ├── Sub-sequence 1B
│   │   ├── Testcase 1B-1
│   │   └── Testcase 1B-2
│   └── Sequence 2
│       ├── Sub-sequence 2A
│       │   ├── Testcase 2A-1
│       │   └── Testcase 2A-2

```

Each level inherits the status of the underlying levels. They are all *test phases* and their statuses are defined by the phase outcome.

### 6.3.4 Defining Sequences or PhaseGroups

Spintop-openhtf uses **PhaseGroup** objects to instantiate *test sequences*. To define a *test sequence* within your *test plan*, simply use the `TestSequence` module.

```

from spintop_openhtf import TestPlan, TestSequence

sequence = TestSequence('Sleep Sequence')

```

To add *test cases* to the sequence, instead of to the *test plan* itself, simply use the sequence instead of the *test plan* in the *test case* decorator.

```

@sequence.testcase('Sleep Test 1')
def sleep_test_1(test):
    """Waits five seconds"""
    sleep(5)

@sequence.testcase('Sleep Test 2')
def sleep_test_2(test):
    """Waits ten seconds"""
    sleep(10)

```

This will create the following hierarchy

```

Test Plan
├── Sleep Sequence
│   ├── Sleep Test 1
│   └── Sleep Test 2

```

To execute it, connect the sequence to its parent, append it to the *test plan*.

```
plan.append(sequence)
```

Add the new sequence to your latest test bench and run it.

The stripped down log excerpt below shows the loading of the **PhaseGroup** defined as the Sleep Sequence, and executes both test cases.

```
Entering PhaseGroup Sleep Sequence
Executing main phases for Sleep Sequence
Handling phase Sleep Test 1
Executing phase Sleep Test 1
Thread finished: <PhaseExecutorThread: (Sleep Test 1)>
Phase Sleep Test 1 finished with result PhaseResult.CONTINUE
Handling phase Sleep Test 2
Executing phase Sleep Test 2
Thread finished: <PhaseExecutorThread: (Sleep Test 2)>
Phase Sleep Test 2 finished with result PhaseResult.CONTINUE
```

[Tutorial source](#)

### 6.3.5 Adding Levels to the Test Hierarchy

Further levels of hierarchy can be added using the *sub\_sequence* function

```
sequence = TestSequence('Sleep Sequence')
sub_seq = sequence.sub_sequence('Sleep Sub Sequence 1')
```

Use the new *sub\_sequence* in the test case declaration to it to the *sub\_sequence*.

```
sub_seq = sequence.sub_sequence('Sleep Sub Sequence 1')
@sub_seq.testcase('Sleep Test 1A')
def sleep_test_1A(test):
    """Waits five seconds"""
    sleep(5)

@sub_seq.testcase('Sleep Test 1B')
def sleep_test_1B(test):
    """Waits five seconds"""
    sleep(5)

sub_seq = sequence.sub_sequence('Sleep Sub Sequence 2')
@sub_seq.testcase('Sleep Test 2')
def sleep_test_2(test):
    """Waits five seconds"""
    sleep(5)
```

The above declarations will define the following hierarchy:

```
test plan
├── Sleep Sequence
│   ├── Sleep Sub Sequence 1
│   │   ├── Sleep Test 1A
│   │   └── Sleep Test 1B
│   └── Sleep Sub Sequence 2
│       └── Sleep Test 2
```

Add the new sub-sequences to your latest test bench and run it.



The stripped down log excerpt below shows the loading of the different **PhaseGroup** objects defined as the Sleep Sequence and both Sleep Sub Sequences and the execution of all test cases.

```
Entering PhaseGroup Sleep Sequence
Executing main phases for Sleep Sequence
Entering PhaseGroup Sleep Sub Sequence 1
Executing main phases for Sleep Sub Sequence 1
Handling phase Sleep Test 1A
Executing phase Sleep Test 1A
Thread finished: <PhaseExecutorThread: (Sleep Test 1A)>
Phase Sleep Test 1A finished with result PhaseResult.CONTINUE
Handling phase Sleep Test 1B
Executing phase Sleep Test 1B
Thread finished: <PhaseExecutorThread: (Sleep Test 1B)>
Phase Sleep Test 1B finished with result PhaseResult.CONTINUE
Entering PhaseGroup Sleep Sub Sequence 2
Executing main phases for Sleep Sub Sequence 2
Handling phase Sleep Test 2
Executing phase Sleep Test 2
Thread finished: <PhaseExecutorThread: (Sleep Test 2)>
Phase Sleep Test 2 finished with result PhaseResult.CONTINUE
```

Further hierarchy levels can be added by creating new sub\_sequences from the sub\_sequence object.

```
sub_seq = sequence.sub_sequence('Sleep Sub Sequence 1')
sub_sub_seq = sub_seq.subsequence('Sleep Sub Sequence 1A')
@sub_sub_seq.testcase('Sleep Test 1A-1')
def sleep_test_1A_1(test):
    """Waits five seconds"""
    sleep(5)
```

And so forth, to define the exact hierarchy needed for your test plan.

Tutorial source

## 6.3.6 Managing the Test Flow Based on the Trigger Phase

As we have seen previously, the trigger phase is used to input dynamic configuration parameters at the beginning of the test bench. This test configuration can be used to manipulate the test flow. For example, a choice of test to execute in the form of a dropdown list or a scanned entry of the product version can lead to a different execution.

The test.state object is used to communicate the information through the test bench. Let's first define a new variable which will allow the execution of *Sleep Test 2* if the product entered in the trigger phase is "A"

```
def trigger(test, prompts):
    """Displays the configuration form"""
    response = prompts.prompt_form(FORM_LAYOUT)
    test.dut_id = response['dutid']
    test.state["operator"] = response['operator']
    test.state["product"] = response['product']
    if test.state["product"] == "A":
        test.state["run_sleep_2"] = True
    else:
        test.state["run_sleep_2"] = False
    pprint(response)
```

The *run\_sleep\_2* entry of the test.state dict will determine whether the test is executed. To add the run time test management, redefine the test with the run\_if option as seen below.

```
@sub_seq.testcase('Sleep Test 2', run_if=lambda state: state.get('run_sleep_2', True))
def sleep_test_3(test):
    """Waits five seconds"""
    sleep(5)
```

This definition will lead to the test being executed if the `run_sleep_2` of the `test.state` dictionary is set to **True**, that is if the product was entered as “A”.

Modify your test bench to reflect the above changes and run it again. When prompted enter “A” as the product. *Sleep Test 2* is executed.

Phases	EXPAND ALL
<b>Configuration</b> : Displays the configuration form (7s)	Pass
<b>Sleep Test 1A</b> : Waits five seconds (5s)	Pass
<b>Sleep Test 1B</b> : Waits five seconds (5s)	Pass
<b>Sleep Test 2</b> : Waits five seconds (5s)	Pass

Re-execute it by entering “B” as the product

Phases	EXPAND ALL
<b>Configuration</b> : Displays the configuration form (2m 33s)	Pass
<b>Sleep Test 1A</b> : Waits five seconds (5s)	Pass
<b>Sleep Test 1B</b> : Waits five seconds (5s)	Pass
<b>Sleep Test 2</b> : Waits five seconds	Waiting

[Tutorial source](#)

### 6.3.7 Using a Set Up and a Teardown or Cleanup Phase

It is a good practice to define a setup and a teardown phase within your sequences.

- The *setup phase* is used to initialize the test environment to execute the test cases in the sequence. Setup failure cancels the execution of the group, including the teardown. Define the setup phase using the `setup()` function.
- The *teardown phase* is usually used to reset the test environment to a known status and is always executed at the end of a sequence if at least one sequence’s test phases is executed. It is executed even if one of the phase fails and the other intermediary phases are not. Define the teardown phase using the `teardown()` function.

## Adding a setup and a teardown phase to a sub-sequence

Add a setup and a teardown phase to the Sleep Sub Sequence 1

```
sub_seq = sequence.sub_sequence('Sleep Sub Sequence 1')

@sub_seq.setup('Sub-sequence Setup')
def sub_setup(test):
    """Says Sub setup."""
    test.logger.info('Sub setup')

@sub_seq.testcase('Sleep Test 1A')
def sleep_test_1A(test):
    """Waits five seconds"""
    sleep(5)

@sub_seq.testcase('Sleep Test 1B')
def sleep_test_1B(test):
    """Waits five seconds"""
    sleep(5)

@sub_seq.teardown('Sub-sequence Cleanup')
def sub_cleanup(test):
    """Says Sub cleanup."""
    test.logger.info('Sub cleanup')
```

```
Test Plan
├── Sleep Sequence
│   ├── Sleep Sub Sequence 1
│   │   ├── Sub-sequence Setup
│   │   ├── Sleep Test 1A
│   │   ├── Sleep Test 1B
│   │   └── Sub-sequence Cleanup
│   └── Sleep Sub Sequence 2
│       └── Sleep Test 2
```

## Final teardown

A teardown phase can also be defined for the *test plan* itself by calling the *teardown()* function from the *@plan* decorator. The plan teardown is used to securely shutdown the test bench (for example turning off all power sources, disconnecting from equipment, etc) whether the test has executed completely or has catastrophically

```
@plan.teardown('cleanup')
def cleanup(test):
    """Says Cleaned up."""
    test.logger.info('Cleaned up.')
```

Tutorial source



## 5. TEST BENCH DOCUMENTATION TUTORIAL

### 7.1 Documenting a Test Case

Documenting a test case in the code itself is very simple. The documentation can be used to generate procedures and is used in the spintop-openhtf web interface to display information to the operator.

The documentation process uses the python docstring feature. For more details consult <https://www.python.org/dev/peps/pep-0257/>.

The following code snippet illustrates how to document a test case using the docstring feature.

```
@plan.testcase('Hello-Test')
@plan.plugin(prompts=UserInput)
def hello_world(test, prompts):
    """Displays Hello Operator in operator prompt and waits for Okay"""
    prompts.prompt('Hello Operator!')
    test.dut_id = 'hello'
```

Add the docstring to the test bench implemented in the the *Running a First Test Bench* tutorial and run the test bench. As can be seen in the web interface, the docstring has been added to the test phase description in the current phase dialog and the executed phases dialog.

**Current test: hello (24s)** Running

DUT: — Started: Mar 12, 2020, 9:27:45 PM

Ran 0 of 1 phases

**Current phase: Hello-Test : Displays Hello Operator in operator prompt and waits fo...** Running

**Phases** EXPAND ALL

**Hello-Test : Displays Hello Operator in operator prompt and waits for Okay (24s)** Running

The docstring is also available throughout the test plan to be printed by the developer.

Tutorial source

## 6. PROPOSED PROJECT STRUCTURE

The following project structure is proposed by the Tack Verification team for complex test bench projects. The structure aims to separate parameters and libraries from the test logic itself.

### 8.1 Test Bench Source Files Categories

We propose to split the test bench source files in the following categories.

#### 8.1.1 Test Tools

The test tools are test function libraries that are either generic or specific to the Device under test or to the equipment used to test it.

Separating them from the test cases themselves allow the reuse of the tools for different test benches testing the same product or family of products, or using the same equipment. It also permits their use outside of the spintop-openhtf context.

The test tools often implement or wrap spintop-openhtf defined plugs. See [About Plugs](#)

#### 8.1.2 Test Cases

The test cases are the functions implementing the test logic. They are loaded by the test bench and executed one after the other.

Separating the test cases from the rest of the test bench allows the sharing of test cases between different test plans and test benches. It is possible to create test case libraries for use in multiple test benches of the same product or the same family of product. For example an Ethernet ping test can be formatted for use in a lot of different products test benches. Using test case libraries allows to reduce the duplication of code across all your test benches.

#### 8.1.3 Test Sequences

The test sequences are groups of test cases that are executed together. In our test bench design practices test sequences are specific to the test bench, and test cases are generic, possibly reused by multiple test benches.

### 8.1.4 Test Criteria

The thresholds against which measures are compared to declare a test case PASS or FAIL. In the spintop-openhtf context, the measures module implements the criteria and the comparison against the selected values.

Test criteria can be changed without modifying the test logic of the test bench. Separating them allows to better track the modifications of one or the other. Also, criteria can be shared between test cases (example a maximum temperature at all steps of the test). Hardcoding them in test cases is therefore not a good practice.

### 8.1.5 Test Station Configuration

The parameters configuring the test station itself, that is parameters changing from station to station and test jig to test jig, such as ip addresses, com port, etc.

Since multiple test benches can use the same station, and the same ip addresses, com port, etc., the test station configuration should also be shared between the different test benches. When a modification is made on the test station, such as changing the test PC IP address, the changes should be inherited by all test benches through a single update to the test station configuration file.

### 8.1.6 Static configuration

The static configuration are the parameters statically configuring the execution of the test plan. They are used to modify the flow of the test without changing the test cases or test tools code itself. The following examples illustrate test parameters that should be defined in the static configuration.

- For a test where there is number of retries permitted for a test step, define the maximum retries number in the static configuration.
- For a test that spans frequencies, define the number of iterations and the tested frequencies in the static configuration.

Separating the static configuration from the test cases allows for the reduction of hardcoded parameters deep in the code, especially if the hardcoded constant is used in several places. It permits the modification of the parameters in a central configuration file.

### 8.1.7 Product Definition

Some test benches are built for families of products which vary slightly from version to version. The product definition is the static configuration which defines the product's parameters.

Separating the product definition from the test sources makes sure that the version particularities are not hardcoded in the test logic. This permits the easier addition of new product versions in the future.

Typically one configuration file is specified per product version. This configuration file is imported once the product versions has been determined in the trigger phase.



## 8.2 Proposed Single-Repository Structure

Let's explore the single-repository test bench structure as proposed by the Tack Verification team. This proposed structure is meant to guide the test bench developers in the layout of their sources, not to restrict them in any way. We propose to use it when the aim is to implement a single test bench for a single product.

As implied, the file structure is implemented in a single Git repository. A folder is created for each of the source categories explained above.

```

repository
├── main.py: Calls and runs the test bench.
├── criteria: The criteria folder holds the global criteria for the test bench.
│   └── Sequence specific criteria can be defined at the sequence level.
│       ├── global_criteria.py
├── products: Each python file defines a different product through its static_
└── parameters.
    ├── product_A.py
    ├── product_B.py
    └── product_C.py

station_config: Each .yaml file defines a different test station.
├── station_1.py
└── station_2.py

static_config: The static config folder holds the product-independent static_
└── configuration for
    ├── the test bench as a whole. Each sequence uses its own static_
    └── configuration as well.
        ├── static_config.py
test_cases: The functions that implement the test cases are defined in test case_
└── libraries.
    ├── lib_A
    │   ├── cases_A1.py
    │   └── cases_A2.py
    ├── lib_B
    └── lib_C

test_sequences: The sequences are separated in folders which hold the sequence_
└── and test cases
    ├── declarations, the static configuration and the criterion specific_
    └── to the sequence.
        ├── sequence_A
        │   ├── sequence_A.py
        │   ├── A_static_config.py
        │   └── A_criteria.py
        ├── sequence_B
        │   ├── sequence_B.py
        │   ├── B_static_config.py
        │   └── B_criteria.py
test_tools: The test tools that are used to implement the test cases are defined_
└── in tool libraries.
  
```

(continues on next page)

(continued from previous page)

```

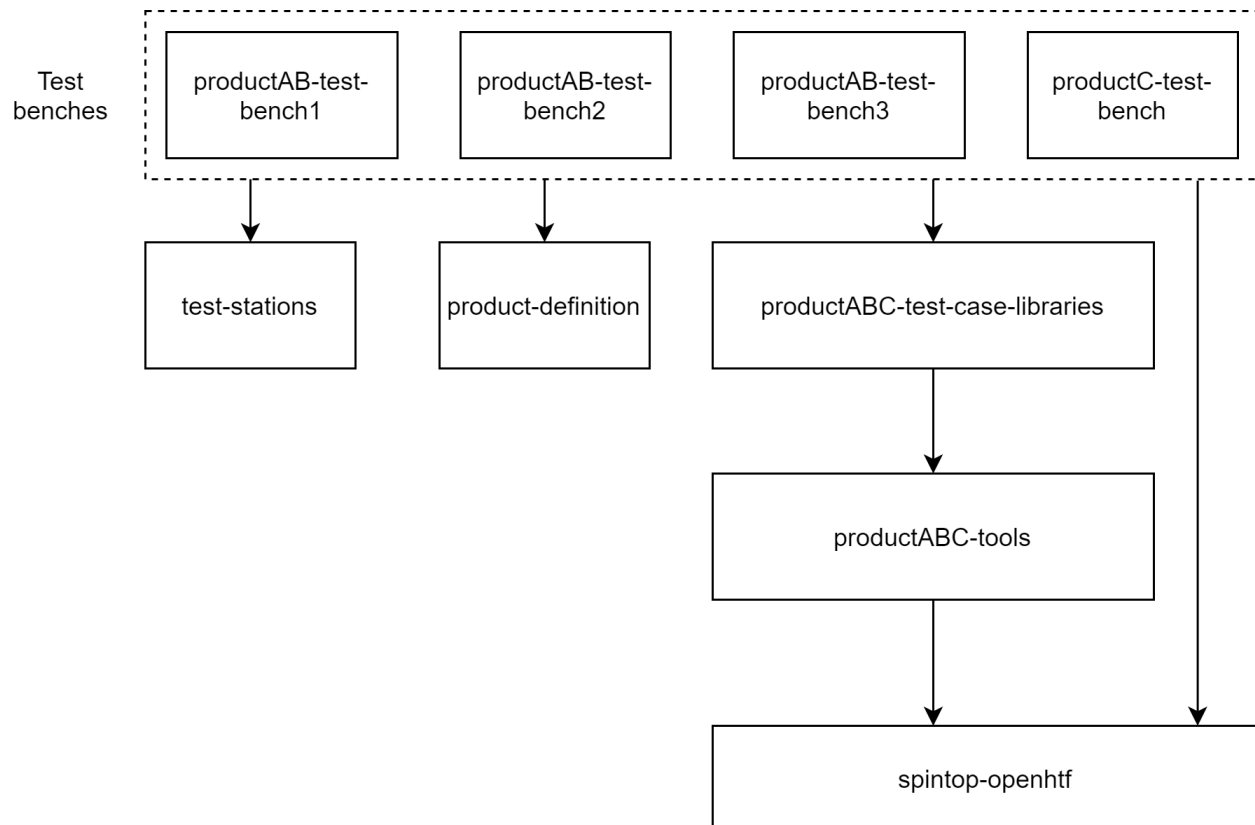
├── lib_A
│   ├── tools_A1.py
│   └── tools_A2.py
├── lib_B
└── lib_C

```

## 8.3 Proposed Multiple-Repository Structure

Let's explore the multiple-repository structure proposed by the Tack Verification team. Once again, the proposed structure is meant to guide, not to restrict. We propose to use it when multiple test benches are developed for a product family with multiple different versions.

The following example illustrates a set of test benches implementing the tests of a family of products. Product A and B are tested by the same set of 3 test benches which are run one after the other. Product C which is in the same product family as products A and B, is tested by a dedicated testb bench. It will however use the same test stations and test case libraries as the other products.



### 8.3.1 Test Bench Repository

At the top level of the repository architecture are the test bench repositories. The test case and tool libraries have been removed from the repository, as well as the test station configuration and product.

Each repository implements a specific test for a product or set of products.

```

repository
├── main.py: Calls and runs the test bench.
├── criteria: The criteria folder holds the global criteria for the test bench.
│   └── Sequence specific criteria can be defined at the sequence level.
│       └── global_criteria.py
├── static_config: The static config folder holds the product-independent static_
└─> configuration for
    │   └── the test bench as a whole. Each sequence uses its own static_
└─> configuration as well.
    └── static_config.py
├── test_sequences: The sequences are separated in folders which hold the sequence_
└─> and test cases
    │   └── declarations, the static configuration and the criterion specific_
└─> to the sequence.
        ├── sequence_A
        │   ├── sequence_A.py
        │   ├── A_static_config.py
        │   └── A_criteria.py
        └── sequence_B
            ├── sequence_B.py
            ├── B_static_config.py
            └── B_criteria.py

```

### 8.3.2 Test Station Configuration Repository

The test station configuration files are defined in a separate repository because any of the 4 test benches from our example can use any of the test stations. It prevents the duplication of information and allows the test bench maintainer to modify a single file when a station is modified instead of having to correct the file in each test bench repository.

```

repository
├── main.py: Calls and runs the test bench.
└── station_config: Each .yaml file defines a different test station.
    ├── station_1.py
    └── station_2.py

```

### 8.3.3 Product Definition Repository

The production definition configuration files are defined in a separate repository because any of the 4 test benches from our example can test one product or the other. It prevents the duplication of information and allows the test bench developer to modify a product definition in a single file which will impact all test benches.

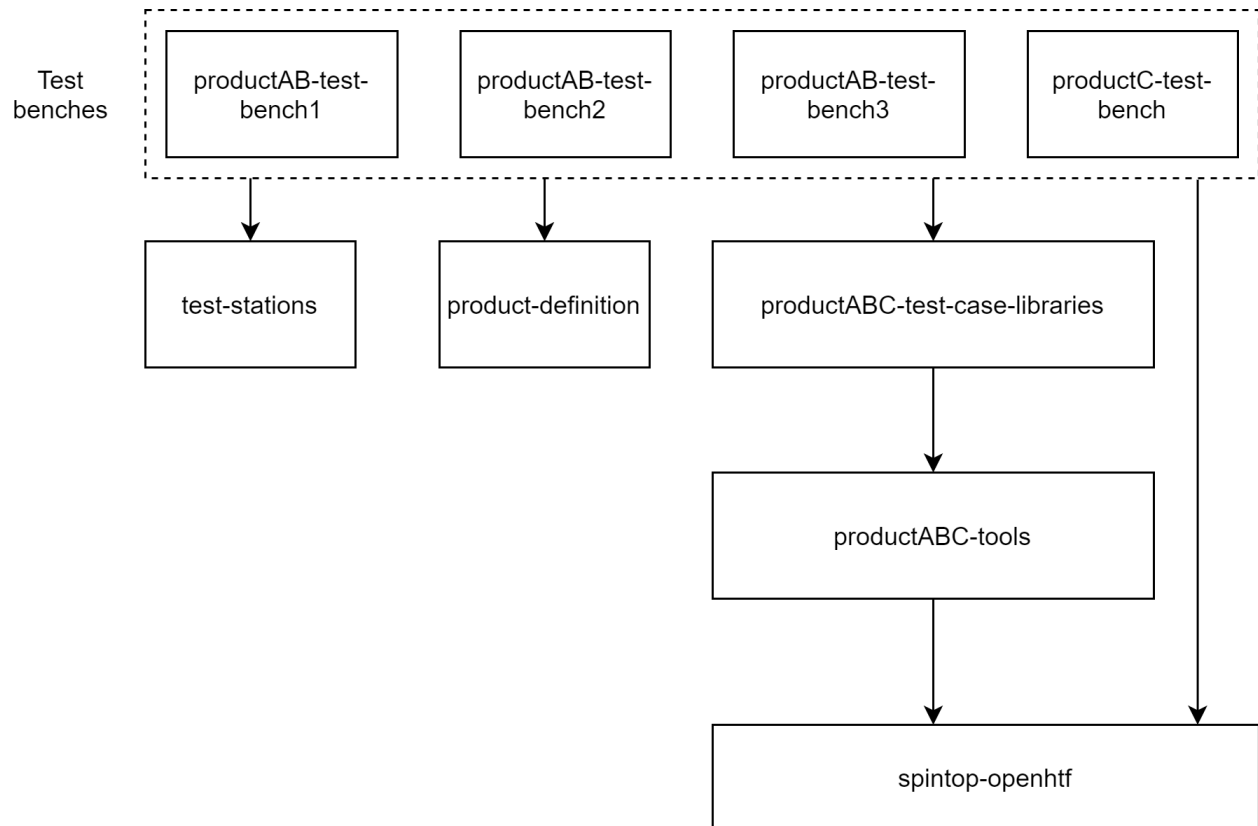
```
repository
|   main.py: Calls and runs the test bench.
|
|   products: Each python file defines a different product through its static_
↳ parameters.
|       product_A.py
|       product_B.py
|       product_C.py
```

### 8.3.4 Test Cases Library Repository

The test case libraries are imported by the test benches to be called by their sequences. Their implementation is made independent from the spintop-openhtf context, meaning that the definition of the test cases as spintop-openhtf phases is made in the sequences of the test bench repository. This allows the test case implementations in this repository to be used outside of a spintop-openhtf test bench, for example as a standalone debugging python executable.

```
repository
|   main.py: The main in the test case repository is used to test or use the test_
↳ cases
|       outside of a spintop-openhtf test bench.
|
|   test_cases: The functions that implement the test cases are defined in test case_
↳ libraries.
|       lib_A
|       |   cases_A1.py
|       |   cases_A2.py
|       lib_B
|       lib_C
```

The following is an example of the definition of a test case in a library and how the a test sequence from the test bench repository calls it.



```

# From the test bench repository
import test_case_lib
sequence = TestSequence('Sleep Sequence')
@sequence.testcase('Test #1 - Sleep Test') # Defines test #1 of the test bench.
def Test1_Sleep(test):
    """Waits five seconds"""
    test_case_lib.sleep_test(sleep_time=5)

```

```

# From a test case library

def sleep_test(sleep_time=5)
    sleep(sleep_time)

```

As can be seen, the spintop-openhtf test case is defined in the test bench repository. This allows its definition as test case #1 and its documentation to make sense in the test bench context. Defining the spintop-openhtf test case in the test case library would mean that the test would have to be test #1 of every test bench that used it.

### 8.3.5 Test Tools Library Repository

The test tools libraries implement functions that control and monitor the product of the equipment around it. They can be implemented as plugs and can wrap plugs from the spintop-openhtf tool libraries. They can be imported and used by the test case libraries or directly by the test bench repository. The test tools are meant to be usable outside of a spintop-openhtf test bench.

```

repository
|
|   main.py: The main in the tools repository is used to test or use the tools
|           outside of a spintop-openhtf test bench.

```

(continues on next page)

(continued from previous page)

```

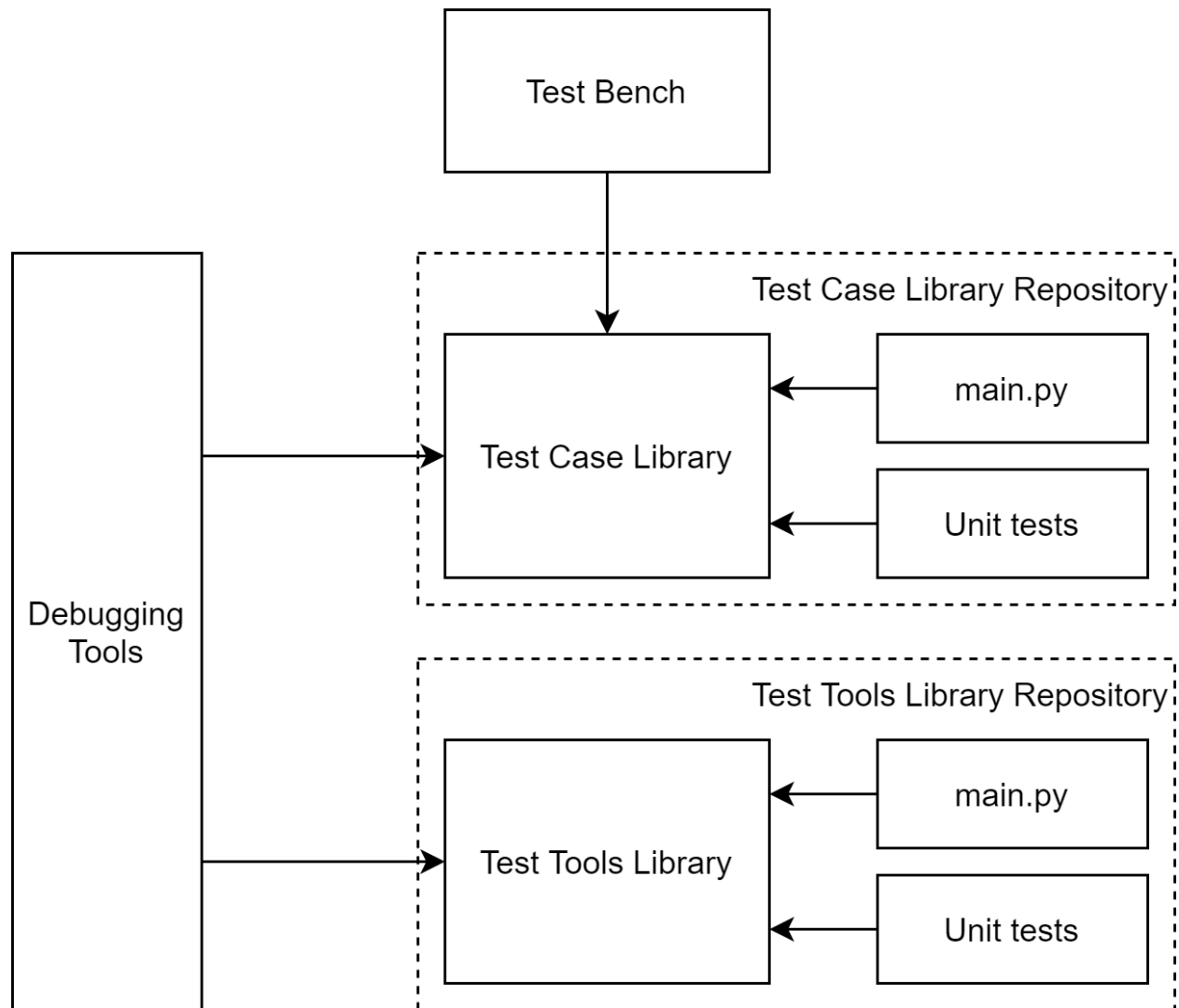
└─test_tools: The test tools that are used to implement the test cases are defined
└─in tool libraries.
    └─lib_A
        └─tools_A1.py
        └─tools_A2.py
    └─lib_B
    └─lib_C

```

### 8.3.6 Unit-testing and Debugging

The proposed structure facilitates the use of unit tests to verify test cases and test tools before a version is ready to be used in a test bench. It also helps the implementation of complete debugging tools using both libraries and bypassing the test bench entirely.

The following image showcases the different entry points of the test case and test tool library usage.



Each library is callable from

- the test bench itself
- the debugging tools to help debug failing units
- the unit test implementation
- the repository main which is used for development and debugging of the library itself

### 8.3.7 Notes on Multiple-Repository Structure

#### Import

We do not specify here the methods of linking the different repositories. This could be done by defining git submodules, or by serving a pypi server hosting the repositories. It is outside the scope of spintop-openhtf and is up to the developer.

#### Versioning

Versioning is of great importance in the case of a multiple-repository structure. Since the test bench can remain the same but its underlying libraries and tools can change, it is imperative that the repositories under the test bench be versioned and that only the required and tested version of these repositories be installed in a deployment or an update.

#### Unit tests

For the test case and test tool libraries, we believe it is a good practice to create a unit test structure to validate new commits of these repositories and create a new version importable by the test benches.





## 7. TEST BENCH CONFIGURATION TUTORIAL

### 9.1 Static configuration

The static configuration are the parameters statically configuring the execution of the test plan. They are used to modify the flow of the test without changing the test cases or test tools code itself. The following examples illustrate test parameters that should be defined in the static configuration.

#### 9.1.1 Definition

In the context of a test bench implementation on spintop-openhtf, the static configuration is defined as classes of parameters accessible across the test bench sources.

Create a new file called static.py in the same folder as your test bench main. In the file, let's define a class building the parameters of the sleep test. Add the following code to the file:

```
class SleepConfig():
    SLEEP_TIME = 5
    SLEEP_ITERATIONS = 2
```

#### 9.1.2 Use of parameters

To access the configuration in the test bench code, simply import the class.

```
from static import SleepConfig
```

and access directly the instantiated parameters.

```
@plan.testcase('Sleep')
def sleep_test(test):
    """Waits five seconds"""

    for x in range(SleepConfig.SLEEP_ITERATIONS):
        print ("Sleep iteration {} - sleep time {}".format(x,
                                                            SleepConfig.SLEEP_TIME))
        sleep(SleepConfig.SLEEP_TIME)
```

Add the use of the static configuration use to your latest test bench and run it.

The test will execute a 5 second sleep twice as determined

```
Sleep iteration 0 - sleep time 5
Sleep iteration 1 - sleep time 5
```

### 9.1.3 Use in Complex Test Benches

Multiple classes can be defined in the same configuration file. The pattern used for the definition is up to the developer. Each test case can have its own class of parameters, or they can be split according to product features across multiple test cases.

The static configuration as proposed is very useful in the definition of complex test benches, for example one managing the tests of different versions of the same products. In this case, the difference between the products can be parametrized using two classes with the same variables at different values.

```
class ProductVersionA():
    TRX_CNT = 2
    TEMP_SENSOR_I2C_ADDR = 0x49

class ProductVersionB():
    TRX_CNT = 1
    TEMP_SENSOR_I2C_ADDR = 0x48
```

Add the product version selection in the trigger phase and import dynamically the right class depending on the selected product. As illustrated below, using the custom trigger phase from the *Custom Trigger Phase* tutorial.

```
if test.state["product"] == "A":
    from static import ProductVersionA as ProductConfig
elif test.state["product"] == "B":
    from static import ProductVersionB as ProductConfig

print ("I2C Address: {}".format(ProductConfig.TEMP_SENSOR_I2C_ADDR))
```

Add the above code to the sleep test case and run it again. Enter “A” for the product when prompted and the right configuration for product A will be imported.

```
I2C Address: 0x49
```

Tutorial source

Configuration file

## 9.2 Test Station Configuration

The test station configuration is the list of all parameters configuring the test station itself, that is parameters changing from station to station and test jig to test jig, such as ip addresses, com port, etc.

### 9.2.1 Definition

In the context of a test bench implementation on spintop-openhtf, the test station configuration is defined as a yaml file. As an example, the following yaml snippet defines the configuration of the serial port and ip address of a test bench.

```
serial :
  comport: "COM4"
  baudrate : "115200"
ip_address : "192.168.0.100"
test_constant: 4
```

Create a new `.yaml` file, paste the above configurations in it and save it as `config.yaml` in the same directory as your test bench `main.py`. It will be imported in a test bench.

## 9.2.2 Load Configuration from File

To load the configurations in the test logic, the `openhth conf` module must be imported.

```
from openhtf.util import conf
```

The configuration parameters used must then be defined. A description of the configuration can be added to the declaration.

```
conf.declare("serial", 'A dict that contains two keys, "comport" and "baudrate"')
conf.declare("ip_address", 'The IP Address of the testbench')
conf.declare("test_constant", 'A test constant')
```

and the configuration file loaded.

```
conf.load_from_filename("config.yaml")
```

## 9.2.3 Use Configuration

Once loaded and declared, the test station parameters can be accessed through the `conf` object. For example, to print some of the configuration parameters, use the following in a test case:

```
print ("Serial port is {}".format(conf.serial["comport"]))
print ("IP address is {}".format(conf.ip_address))
print ("Test constant is {}".format(conf.test_constant))
```

Add the above code excerpts to your latest test bench and run it.

The test will print the information in the console window as

```
Serial port is COM4
IP address is 192.168.0.100
Test constant is 4
```

## 9.2.4 Built-in station id parameter

Spintop-openhtf uses a built-in parameter in the `conf` object that defines the test station id. The id used is the hostname of the PC on which the test bench runs. For example, printing the station ID of a PC whose hostname is “TutorialStation”

```
print ("Station ID is {}".format(conf.station_id))
```

will result in

```
Station ID is TutorialStation
```

Tutorial source

Configuration file



## 8. PLUGS TUTORIAL

### 10.1 About Plugs

Plugs are an OpenHTF concept. The OpenHTF team defines them as follow:

The essence of an OpenHTF test is to interact with a DUT to exercise it in various ways and observe the result. Sometimes this is done by communicating directly with the DUT, and other times it's done by communicating with a piece of test equipment to which the DUT is attached in some way. A **plug** is a piece of code written to enable OpenHTF to interact with a particular type of hardware, whether that be a DUT itself or a piece of test equipment.

Technically, plugs are a Python class that is instantiated once per test and shared between test phases. They have a strong sense of *cleanup* that allows them to execute specific teardown actions regardless of the test outcome.

Although OpenHTF references hardware directly, plugs are also used in various roles that are non-related to hardware, such as user input. Overall, a better explanation would be that they are used for resources that are shared by test cases:

- Plugs for test equipments
- Plugs for DUT interfacing, which can be subdivided in some cases:
  - COM Interface
  - SSH Interface
  - Etc.
- Plugs for user input
- Plugs for custom frontend interaction
- Plugs for sharing test context, such as calibration performed over multiple test cases

### 10.2 Using Plugs

Using plugs in spintop-openhtf is pretty straightforward. Let's take the UserInput plug as example since it is used in pretty much all tests.

1. First, the plug must be imported.

```
from openhtf.plugs.user_input import UserInput
```

2. Then, on testcases that require this plug, the plug decorator must be used.

```
from openhtf.plugs.user_input import UserInput
from spintop_openhtf import TestPlan

plan = TestPlan('mytest')

@plan.testcase('TEST1')
@plan.plugin(prompt=UserInput) # 'prompt' is user-defined
def test_1(test, prompt): # prompt will contain an instance of UserInput
```

The *class* of the plug is used as argument to the plug decorator. **This is important.** The executor will instantiate an instance of the class and use the same object across a test run. **On each new test, the plug will be re-instantiated.**

**Warning:** You choose the name you want to give to the argument. The name must have a match in the function definition. For example, **the following would FAIL:**

```
# Will complain that 'prompt' is not an argument
@plan.testcase('TEST1')
@plan.plugin(prompt=UserInput)
def test_1(test, user_input): # WRONG. No user_input argument exists
```

## 10.3 Creating Plugs

Creating plugs is the basis of reusing interface functionalities. As an example, we will create a plug that copies a file from a source folder to a destination.

### 10.3.1 Base Structure

Every plug must inherit from `BasePlug`. Moreover, the `__init__` method must take no arguments. The following excerpt illustrates the plug definition

```
import shutil

from openhtf.plugs import BasePlug

class FileCopier(BasePlug):
    def copy_file(self, source_file, destination_folder):
        shutil.copy(source_file, destination_folder)
```

The following excerpt instantiates the plug in a test case.

```
@plan.testcase('File Copy Test')
@plan.plugin(copy_plug=FileCopier)
def file_copy_test(test, copy_plug):
    copy_plug.copy_file(source, destination)
```

Tutorial source

## 10.4 Wrapping spintop-openhtf Plugs

A typical manner of creating custom plugs for a test bench is to wrap an existing, spintop-openhtf plug, to add applicative functionalities to it.

For example, a linux shell plug specific to a product can be created by wrapping the spintop-openhtf comport plug. The plug adds typical linux features to the comport plug such as login, file read, file copy, etc.

The plug is created

```
from spintop_openhtf.plugins.iointerface.comport import ComportInterface

class LinuxPlug(ComportInterface):

    def __init__(self, comport, baudrate=115200):
        super().__init__(comport, baudrate)

    def login(self, username):
        return self.com_target("{}".format(username), '{}@'.format(username),
        ↪timeout=10, keepelines=0)

    def file_read(self, file):
        return self.com_target("cat {}".format(file), '@', timeout=10, keepelines=0)

    def file_copy(self, source, destination):
        return self.com_target("cp {} {}".format(source, destination), '@',
        ↪timeout=10, keepelines=0)
```

and imported in a test case

```
linux_plug = LinuxPlug.as_plug('linux_plug', comport='COM5', baudrate=115200)

@plan.testcase('Linux_Test')
@plan.plug(linux=linux_plug)
def LinuxTest(test, linux):
    try:
        linux.open_with_log()
        test.logger.info ("COM Port open")
        print(linux.file_read('file.txt'))
        linux.file_copy('file.txt',destination)
    except:
        test.logger.info ("COM Port open failed")
        return PhaseResult.STOP
```





## 9. TEST CRITERIA TUTORIAL

### 11.1 Defining Test Criteria

The criteria refer to the thresholds against which measures are compared to declare a test case PASS or FAIL. In the spintop-openhtf context, the measures module implements the criteria and the comparison against the selected values.

To define a test criterion, first define an openhtf measurement object.

```
import openhtf as htf
criterion = htf.Measurement('test_criterion').in_range(18, 22)
```

Here the criterion defined will return a PASS if the value evaluated is between 18 and 22. The criterion name is “test\_criterion” and it has been stored in the *criterion* variable.

Use the htf.measures decorator to use the defined criterion in a test case.

```
@plan.testcase('Criteria test')
@htf.measures(criterion)
def criteria_test(test):
    value = 20
    test.measurements.test_criterion = value
```

The criterion object is loaded in the openhtf measures. To evaluate a value against the test criterion, simply equate the value to the criterion. Note that the criterion name (“test\_criterion”) is used and not the object variable.

Add the defined criteria and test case to your latest test bench and run it. You will see that the new test phase called *Criteria test* has passed.

Phases	EXPAND ALL
trigger_phase : Test start trigger that prompts the user for a DUT ID. (1m 13s)	Pass
Criteria test : (0s)	Pass

Hit expand all on the Phases box and see that the evaluated criteria has been added to the test phase result.

Phases				COLLAPSE ALL
<b>trigger_phase</b> : Test start trigger that prompts the user for a DUT ID. (1m 13s)				Pass
<b>Criteria test</b> : (0s)				Pass
<u>Measurement name</u> test_criterion	<u>Value</u> 20	<u>Validators</u> 18 <= x <= 22	<u>Result</u> Pass	

Modify the value in the test case code to use a value outside of the criteria range.

```
@plan.testcase('Criteria test')
@htf.measures(criteria)
def criteria_test(test):
    value = *12*
    test.measurements.test_criterion = value
```

Run the test bench again. The phase will now fail.

Phases				COLLAPSE ALL
<b>trigger_phase</b> : Test start trigger that prompts the user for a DUT ID. (4s)				Pass
<b>Criteria test</b> : (0s)				Fail
<u>Measurement name</u> test_criterion	<u>Value</u> 12	<u>Validators</u> 18 <= x <= 22	<u>Result</u> Fail	

[Tutorial source](#)

## 11.2 Criteria types

In the example above, a range was defined to instantiate the criteria. Multiple different validator types can be used instead of the range function.

A good practice is to use a validator function which will return *True* or *False* depending on the value evaluated. For example, our range criteria can be defined in another manner as

```
def range_validator(value):
    return 18 <= value <= 22

criterion = htf.Measurement('test_criterion').with_validator(range_validator)
```

Using the `with_validator` function helps you create complex criteria that match your exact needs.

For more details on the different types of criteria that can be implemented please refer the the Measurements reference

## 11.3 Documentation

It is possible to add documentation to the criterion with the *doc()* function

```
criterion = htf.Measurement('test_criterion').in_range(18, 22)
                .doc('This measurement helps illustrate the criteria usage in_
↳spintop-openhtf')
```

## 11.4 Using a criteria definition file

As we have seen in the *6. Proposed Project Structure* tutorial , we believe it is a good practice to separate the criteria definition from the actual test logic of the test bench. This guideline is proposed because it allows the modification and the tracking of all criteria in a single emplacement. It also eliminates the duplication of criteria.

To this purpose, we propose to create a file called, for example, *criteria.py*. In this file a function called *get\_criteria()* takes the criterion name as argument and returns the criterion which the **@htf.measures** decorator uses.

Create the file and implement the function as described below.

```
import openhtf as htf

def get_criteria(criterion):
    criteria_dict = {
        "test_criterion": htf.Measurement('test_criterion').in_range(18,22)
    }

    return criteria_dict[criterion]
```

The python dictionary acts as a criteria switch case. The function selects the required measurement object and returns it to the caller.

In your test bench, use the function in the **@htf.measures** decorator to load the criteria and use it directly.

```
from criteria import get_criteria

@plan.testcase('Criteria test')
@htf.measures(get_criteria('test_criterion'))
def criteria_test(test):
    value = 20
    test.measurements.test_criterion = value
```

Run the test bench again and the result should be the same as was obtained above.

Tutorial source

Criteria file



## 10. TEST RESULTS TUTORIAL

### 12.1 Exploring the Test Results

#### 12.1.1 Test Result Folder

Each execution of the test bench creates a *test record* is stored on the test station computer. This *test record* is saved in the %localappdata%\tackv\spintop-openhtf\openhtf-history\test-bench-name\ folder. For example, C:\Users\tack.user\AppData\Local\tackv\spintop-openhtf\openhtf-history\hello for our **hello** test bench.

For each *test record* a folder is created with its name formatted as *dut-id\_execution-date-and-time\_result*. For example, SN01\_2020\_04\_12\_213700\_409000\_PASS is a test for the SN01 unit executed on April 12th which resulted in a global PASS. In the folder are all the results gathered during the test.

#### 12.1.2 Test Result JSON file

The main result file is the **\*\*result.json\*\*** file. It holds the result of the test, of all phases, as well as all the logs and the complete measurements with their evaluated criteria.

##### Global results

At the top of the file are the global test results, namely:

- the Device Under Test ID
- the start and end time in Unix epoch time
- the global outcome of the test bench
- the station ID

```
"dut_id": "SN01",  
"start_time_millis": 1586727420409,  
"end_time_millis": 1586727435444,  
"outcome": "PASS",  
"outcome_details": [],  
"station_id": "P089",
```

##### Metadata

The metadata section of the results follow. The test name as well as the used test station configuration is logged there.

```
"metadata": {  
  "test_name": "Sleep",  
  "config": {
```

(continues on next page)

(continued from previous page)

```

        "plug_teardown_timeout_s": 0,
        "allow_unset_measurements": false,
        "station_id": "Tutorial Station",
        "cancel_timeout_s": 2,
        "stop_on_first_failure": false,
        "capture_source": false,
        "capture_docstring": true,
        "teardown_timeout_s": 30,
        "user_input_enable_console": false,
        "frontend_throttle_s": 0.15,
        "station_server_port": "4444"
    }
},

```

### Phases Results

The phase results follow where the file logs them as an array

```

"phases": [
    ...
],

```

For each phase is logged,

- the start and end time in Unix epoch time
- the phase name (*name*)
- the phase result (*result.phase\_result*)
- the phase outcome (*outcome*)

```

{
    "measurements": {},
    "options": {
        "name": "Sleep Test 1A",
        "run_under_pdb": false
    },
    "start_time_millis": 1586727420410,
    "end_time_millis": 1586727425412,
    "attachments": {},
    "result": {
        "phase_result": "CONTINUE"
    },
    "outcome": "PASS",
    "descriptor_id": 2136562880584,
    "name": "Sleep Test 1A",
    "codeinfo": {
        "name": "sleep_test_1",
        "docstring": "Waits five seconds",
        "sourcecode": ""
    }
},

```

The measurements and criteria details are also added with their parameters in case there was one defined for the phase.

- name (*measurements.criterionname.name*)
- outcome (*measurements.criterionname.outcome*)

- evaluated value (measurements.criterionname.measured\_value)
- validators (measurements.criterionname.validators)

```
{
  {
    "measurements": {
      "test_criterion": {
        "name": "test_criterion",
        "outcome": "PASS",
        "validators": [
          "18 <= x <= 22"
        ],
        "docstring": "This measurement helps illustrate the criteria usage_
→in spintop-openhtf",
        "measured_value": 20
      }
    },
    "options": {
      "name": "Criteria test",
      "run_under_pdb": false
    },
    "start_time_millis": 1586635840894,
    "end_time_millis": 1586635840894,
    "attachments": {},
    "result": {
      "phase_result": "CONTINUE"
    },
    "outcome": "PASS",
    "descriptor_id": 2150767337288,
    "name": "Criteria test",
    "codeinfo": {
      "name": "criteria_test",
      "docstring": null,
      "sourcecode": ""
    }
  }
}
```

## Logs

Each log entry is kept in the test record. For each log entry are recorded

- the logger name
- the logging level (info, warning, error, etc)
- the source (source code file and line)
- the timestamp
- the logged message

```
"log_records": [
  {
    "level": 10,
    "logger_name": "openhtf.core.phase_executor",
    "source": "phase_executor.py",
    "lineno": 249,
    "timestamp_millis": 1586727409681,
    "message": "Executing phase Configuration"
```

(continues on next page)

(continued from previous page)

```

    },
    {
        "level": 10,
        "logger_name": "openhtf.plugs.user_input",
        "source": "user_input.py",
        "lineno": 369,
        "timestamp_millis": 1586727409683,
        "message": "Displaying prompt (97f2a2ebc7c2491f8b3712efed20f34c): \"{'schema
↪': {'title': 'Test configuration', 'type': 'object', 'required': ['operator', uuid,
↪product'], 'properties': {'operator': {'type': 'string', 'title': 'Enter the
↪operator name'}, 'dutid': {'type': 'string', 'title': 'Enter the device under test
↪serial number'}, 'product': {'type': 'string', 'title': 'Enter the product name'}}},
↪ 'layout': ['operator', 'dutid', 'product']}]\""
    },

```

## 12.2 Appending Data to Test Record

In the case your test bench generates a file during its execution, it is possible to add it to the test record. To do so use the `test.attach` or `test.attach_from_file()` functions.

Using the `test.attach()` function as below will create a file name ***test\_attachment*** which holds *This is test attachment data.* as text data in the test record folder.

```
test.attach('test_attachment', 'This is test attachment data.'.encode('utf-8'))
```

Using the `test.attach_from_file()` function as below will take a previously generated data file named ***example\_attachment.txt*** and move it to the test record folder.

```
test.attach_from_file(os.path.join(os.path.dirname(__file__), 'example_attachment.txt
↪'))
```

Create the ***example\_attachment.txt*** file in the folder holding your test bench main, and add both lines to one of your test cases. Run the test bench and verify that the new test run has saved the test data correctly.

### 12.2.1 Loading Data from a Test Record File

Once a file has been saved in the test record, it becomes accessible from the source code. The ***test.get\_attachment()*** function allows the loading of a test record file. Add the following to your test case to load the file created above by the ***test.attach*** function.

```
test_attachment = test.get_attachment('test_attachment')
print(test_attachment)
```

You should see that the content of the ***test\_attachment*** file has been printed in the console, that is 'This is test attachment data.'

Tutorial source

Attachment file



## INDICES AND TABLES

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



## PYTHON MODULE INDEX

### O

`openhxf.util.conf`, 9



## INDEX

### A

[add\\_callbacks\(\)](#) (*spintop\_openhtf.TestPlan* method), 6  
[allow\\_unset\\_measurements](#) (*spintop\_openhtf.testplan.\_default\_conf.ConfigHelpText* attribute), 11  
[append\(\)](#) (*spintop\_openhtf.TestSequence* method), 8  
[as\\_plug\(\)](#) (*spintop\_openhtf.plugs.base.UnboundPlug* class method), 13  
[as\\_plug\(\)](#) (*spintop\_openhtf.plugs.comport.ComportInterface* class method), 14  
[as\\_plug\(\)](#) (*spintop\_openhtf.plugs.ssh.SSHInterface* class method), 16  
[as\\_plug\(\)](#) (*spintop\_openhtf.plugs.telnet.TelnetInterface* class method), 17  
[close\\_log\(\)](#) (*spintop\_openhtf.plugs.ssh.SSHInterface* method), 16  
[close\\_log\(\)](#) (*spintop\_openhtf.plugs.telnet.TelnetInterface* method), 17  
[com\\_target\(\)](#) (*spintop\_openhtf.plugs.comport.ComportInterface* method), 14  
[ComportInterface](#) (class in *spintop\_openhtf.plugs.comport*), 14  
[ConfigHelpText](#) (class in *spintop\_openhtf.testplan.\_default\_conf*), 11  
[ConfigurationInvalidError](#), 9  
[connection\\_lost\(\)](#) (*spintop\_openhtf.plugs.comport.ComportInterface* method), 14  
[connection\\_made\(\)](#) (*spintop\_openhtf.plugs.comport.ComportInterface* method), 14

### C

[cancel\\_timeout\\_s](#) (*spintop\_openhtf.testplan.\_default\_conf.ConfigHelpText* attribute), 11  
[capture\\_docstring](#) (*spintop\_openhtf.testplan.\_default\_conf.ConfigHelpText* attribute), 12  
[capture\\_source](#) (*spintop\_openhtf.testplan.\_default\_conf.ConfigHelpText* attribute), 12

[clear\\_lines\(\)](#) (*spintop\_openhtf.plugs.comport.ComportInterface* method), 14  
[close\(\)](#) (*spintop\_openhtf.plugs.base.UnboundPlug* method), 13  
[close\(\)](#) (*spintop\_openhtf.plugs.comport.ComportInterface* method), 14  
[close\(\)](#) (*spintop\_openhtf.plugs.ssh.SSHInterface* method), 16  
[close\(\)](#) (*spintop\_openhtf.plugs.telnet.TelnetInterface* method), 17  
[close\\_log\(\)](#) (*spintop\_openhtf.plugs.base.UnboundPlug* method), 13  
[close\\_log\(\)](#) (*spintop\_openhtf.plugs.comport.ComportInterface* method), 14

### D

[data\\_received\(\)](#) (*spintop\_openhtf.plugs.comport.ComportInterface* method), 14  
[Declaration](#) (class in *openhtf.util.conf*), 9  
[declare\(\)](#) (in module *openhtf.util.conf*), 9

### E

[eof\\_received\(\)](#) (*spintop\_openhtf.plugs.comport.ComportInterface* method), 14  
[err\\_output](#) (*spintop\_openhtf.plugs.ssh.SSHInterface.SSHResponse* attribute), 16  
[execute\(\)](#) (*spintop\_openhtf.TestPlan* method), 6  
[execute\\_command\(\)](#) (*spintop\_openhtf.plugs.comport.ComportInterface* method), 14  
[execute\\_command\(\)](#) (*spintop\_openhtf.plugs.ssh.SSHInterface* method), 16  
[execute\\_command\(\)](#) (*spintop\_openhtf.plugs.telnet.TelnetInterface* method), 17

`execute_test()` (*spintop\_openhtf.TestPlan* property), 6  
`exit_code` (*spintop\_openhtf.plugins.ssh.SSHInterface.SSHResponse* attribute), 16  
`log_to_stream()` (*spintop\_openhtf.plugins.ssh.SSHInterface* method), 17  
`log_to_stream()` (*spintop\_openhtf.plugins.telnet.TelnetInterface* method), 18

## F

`frontend_throttle_s` (*spintop\_openhtf.testplan.\_default\_conf.ConfigHelpText* attribute), 12  
`logger` (*spintop\_openhtf.plugins.base.UnboundPlug* attribute), 13

## H

`history_path()` (*spintop\_openhtf.TestPlan* property), 6

## I

`image_url()` (*spintop\_openhtf.TestPlan* method), 6  
`inject_positional_args()` (in module *openhtf.util.conf*), 10  
`InvalidKeyError`, 9  
`is_runnable()` (*spintop\_openhtf.TestPlan* property), 6

## K

`keep_lines()` (*spintop\_openhtf.plugins.comport.ComportInterface* method), 15  
`KeyAlreadyDeclaredError`, 9

## L

`load()` (in module *openhtf.util.conf*), 10  
`load_flag_values()` (in module *openhtf.util.conf*), 10  
`load_from_dict()` (in module *openhtf.util.conf*), 10  
`load_from_file()` (in module *openhtf.util.conf*), 10  
`load_from_filename()` (in module *openhtf.util.conf*), 10  
`log_to_filename()` (*spintop\_openhtf.plugins.base.UnboundPlug* method), 13  
`log_to_filename()` (*spintop\_openhtf.plugins.comport.ComportInterface* method), 15  
`log_to_filename()` (*spintop\_openhtf.plugins.ssh.SSHInterface* method), 17  
`log_to_filename()` (*spintop\_openhtf.plugins.telnet.TelnetInterface* method), 18  
`log_to_stream()` (*spintop\_openhtf.plugins.base.UnboundPlug* method), 13  
`log_to_stream()` (*spintop\_openhtf.plugins.comport.ComportInterface* method), 15

## M

`measures()` (*spintop\_openhtf.TestSequence* method), 8  
`message_target()` (*spintop\_openhtf.plugins.comport.ComportInterface* method), 15

## N

`name` (*spintop\_openhtf.TestSequence* attribute), 8  
`next_line()` (*spintop\_openhtf.plugins.comport.ComportInterface* method), 15  
`no_trigger()` (*spintop\_openhtf.TestPlan* method), 6

## O

`open()` (*spintop\_openhtf.plugins.base.UnboundPlug* method), 13  
`open()` (*spintop\_openhtf.plugins.comport.ComportInterface* method), 15  
`open()` (*spintop\_openhtf.plugins.ssh.SSHInterface* method), 17  
`open()` (*spintop\_openhtf.plugins.telnet.TelnetInterface* method), 18  
`openhtf.util.conf` (module), 9  
`output()` (*spintop\_openhtf.plugins.ssh.SSHInterface.SSHResponse* property), 16

## P

`pause_writing()` (*spintop\_openhtf.plugins.comport.ComportInterface* method), 15  
`plug()` (*spintop\_openhtf.TestSequence* method), 8  
`plug_teardown_timeout_s` (*spintop\_openhtf.testplan.\_default\_conf.ConfigHelpText* attribute), 11

## R

`repeat_limit` (*spintop\_openhtf.TestSequence* attribute), 9  
`requires_state` (*spintop\_openhtf.TestSequence* attribute), 9  
`reset()` (in module *openhtf.util.conf*), 10  
`resume_writing()` (*spintop\_openhtf.plugins.comport.ComportInterface* method), 15

`run()` (*spintop\_openhtf.TestPlan* method), 6  
`run_console()` (*spintop\_openhtf.TestPlan* method), 6  
`run_if` (*spintop\_openhtf.TestSequence* attribute), 8  
`run_once()` (*spintop\_openhtf.TestPlan* method), 7  
`run_under_pdb` (*spintop\_openhtf.TestSequence* attribute), 9

**S**

`save_and_restore()` (in module *openhtf.util.conf*), 11  
`setup()` (*spintop\_openhtf.TestSequence* method), 8  
`SSHInterface` (class in *spintop\_openhtf.plugs.ssh*), 16  
`SSHInterface.SSHResponse` (class in *spintop\_openhtf.plugs.ssh*), 16  
`station_discovery_address` (*spintop\_openhtf.testplan.\_default\_conf.ConfigHelpText* attribute), 12  
`station_discovery_port` (*spintop\_openhtf.testplan.\_default\_conf.ConfigHelpText* attribute), 12  
`station_discovery_ttl` (*spintop\_openhtf.testplan.\_default\_conf.ConfigHelpText* attribute), 12  
`station_id` (*spintop\_openhtf.testplan.\_default\_conf.ConfigHelpText* attribute), 11  
`station_server_port` (*spintop\_openhtf.testplan.\_default\_conf.ConfigHelpText* attribute), 12  
`std_output` (*spintop\_openhtf.plugs.ssh.SSHInterface.SSHResponse* attribute), 16  
`stop_on_first_failure` (*spintop\_openhtf.testplan.\_default\_conf.ConfigHelpText* attribute), 12  
`sub_sequence()` (*spintop\_openhtf.TestSequence* method), 8

**T**

`tearDown()` (*spintop\_openhtf.plugs.base.UnboundPlug* method), 13  
`tearDown()` (*spintop\_openhtf.plugs.comport.ComportInterface* method), 15  
`tearDown()` (*spintop\_openhtf.plugs.ssh.SSHInterface* method), 17  
`tearDown()` (*spintop\_openhtf.plugs.telnet.TelnetInterface* method), 18  
`teardown()` (*spintop\_openhtf.TestSequence* method), 8  
`teardown_timeout_s` (*spintop\_openhtf.testplan.\_default\_conf.ConfigHelpText* attribute), 12  
`TelnetInterface` (class in *spintop\_openhtf.plugs.telnet*), 17

`testcase()` (*spintop\_openhtf.TestSequence* method), 8  
`TestPlan` (class in *spintop\_openhtf*), 5  
`TestSequence` (class in *spintop\_openhtf*), 7  
`timeout_s` (*spintop\_openhtf.TestSequence* attribute), 8  
`trigger()` (*spintop\_openhtf.TestPlan* method), 7  
`trigger_phase()` (*spintop\_openhtf.TestPlan* property), 7

**U**

`UnboundPlug` (class in *spintop\_openhtf.plugs.base*), 13  
`UndeclaredKeyError`, 9  
`UnsetKeyError`, 9  
`user_input_enable_console` (*spintop\_openhtf.testplan.\_default\_conf.ConfigHelpText* attribute), 12

**W**

`write()` (*spintop\_openhtf.plugs.comport.ComportInterface* method), 15