pykiso

Release 0.9.4

Sebastian Fischer, Daniel Bühler, Damien Kayser

Nov 23, 2021

CONTENTS:

CHAPTER

ONE

GETTING STARTED WITH PYKISO

1.1 Requirements

- Python 3.6+
- pip/pipenv (used to get the rest of the requirements)

1.2 Install

```
git clone https://github.com/eclipse/kiso-testing.git
cd kiso-testing
pip install .
```

Pipenv is more appropriate for developers as it automatically creates virtual environments.

```
git clone https://github.com/eclipse/kiso-testing.git
cd kiso-testing
pipenv install --dev
pipenv shell
```

1.3 Usage

Once installed the application is bound to `pykiso`, it can be called with the following arguments:

```
$ pykiso --help
Usage: pykiso [OPTIONS]

  Embedded Integration Test Framework - CLI Entry Point.

  :param test_configuration_file: path to the YAML config file :param
  log_path: path to directory or file to write logs to :param log_level: any
  of DEBUG, INFO, WARNING, ERROR :param report_type: if "test", the standard
  report, if "junit", a junit report is generated

Options:
  -c, --test-configuration-file FILE
                                  path to the test configuration file (in YAML
```

(continues on next page)

(continued from previous page)

```
                                format)  [required]
  -l, --log-path PATH           path to log-file or folder. If not set will
                                log to STDOUT
  --log-level [DEBUG|INFO|WARNING|ERROR]
                                set the verbosity of the logging
  --junit                       enables the generation of a junit report
  --text                        default, test results are only displayed in
                                the console
  --version                     Show the version and exit.
  --help                        Show this message and exit.
```

Suitable config files are available in the `test-examples` folder.

1.3.1 Demo using example config

```
invoke run
```

1.3.2 Running the Tests

```
invoke test
```

or

```
pytest
```

1.4 List of limitations / todos for the python side

- **When the auxiliary does not answer (ping or else), BasicTest.cleanup_and_skip() call will result in a lock and break the framework.**
- No test-section will be executed, needs to be removed later.
- test configuration files need to be reworked
- Names & configurations in the *cfg file json* are character precise class names & associated parameters.
- Spelling mistakes need to be fixed! *ongoing*
- Add verbosity parameters to pass to the unittest framework to get more details about the test.
- **Add result parsing for Jenkins (see: https://stackoverflow.com/questions/11241781/python-unittests-in-jenkins).**
- Create a python package - and host it on pip.

CHAPTER

TWO

GETTING STARTED WITH PYKISO FOR CONTRIBUTORS

2.1 Requirements

- Python 3.6+
- pipenv (used to get the rest of the requirements)

2.2 Install

```
git clone https://github.com/eclipse/kiso-testing.git
cd kiso-testing
pipenv install --dev
pipenv shell
```

2.2.1 Pre-Commit

To improve code-quality, a configuration of pre-commit hooks are available. The following pre-commit hooks are used:

- black
- trailing-whitespace
- end-of-file-fixer
- check-docstring-first
- check-json
- check-added-large-files
- check-yaml
- debug-statements
- flake8
- isort

If you don't have pre-commit installed, you can get it using pip:

```
pip install pre-commit
```

Start using the hooks with

```
pre-commit install
```

2.2.2 Demo using example config

```
invoke run
```

2.2.3 Running the Tests

```
invoke test
```

or

```
pytest
```

2.2.4 Building the Docs

```
invoke docs
```

CHAPTER

THREE

INTEGRATION TEST FRAMEWORK DEVELOPER'S GUIDE

3.1 Introduction

The Integration Test Framework provides the possibility to write and run tests on a HW target. It is built to orchestrate the entities and services involved in the tests. The framework can be used for both white-box and black-box testing as well as in the integration and system testing.

3.2 Design Overview

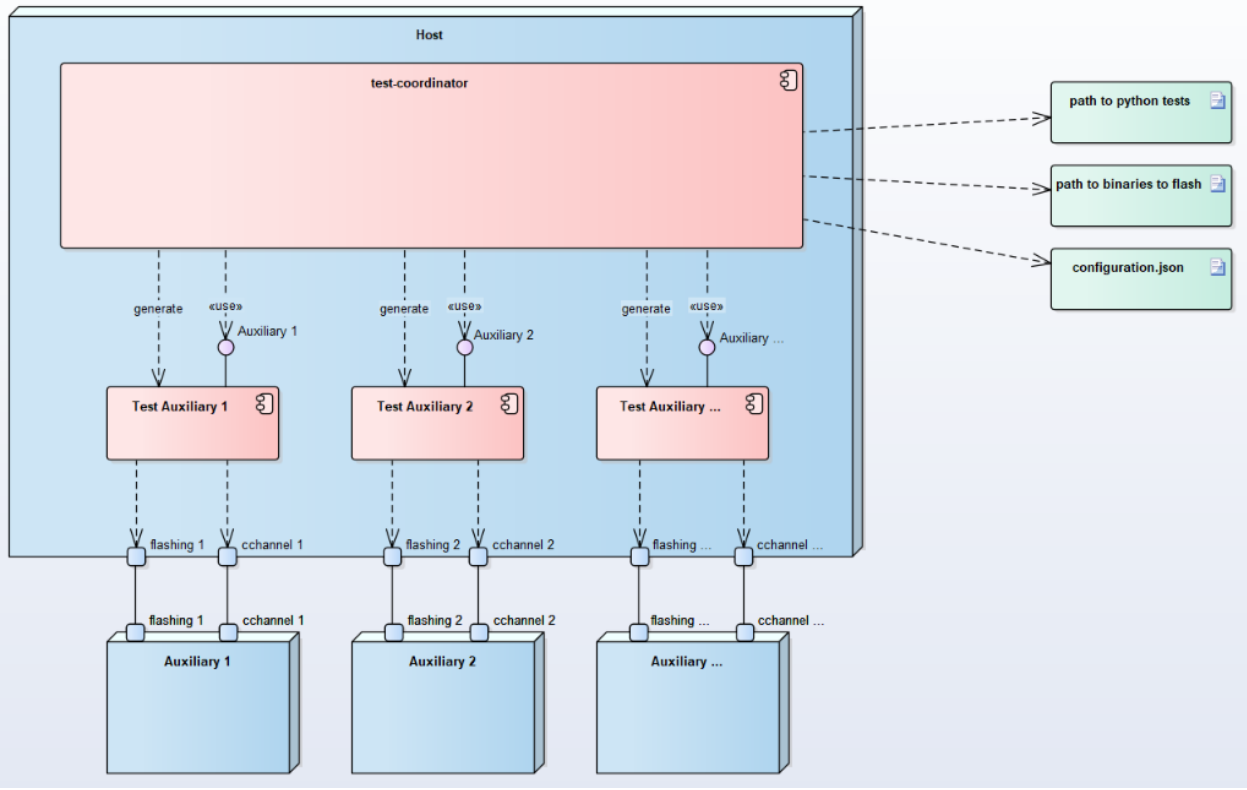


Fig. 1: Figure 1: Integration Test Framework Context

The *pykiso* Testing Framework is built in a modular and configurable way with abstractions both for entities (e.g. a handler for the device under test) and communication (e.g. UART or TCP/IP).

The tests leverage the python *unittest*-Framework which has a similar flavor as many available major unit testing frameworks and thus comes with an ecosystem of tools and utilities.

3.2.1 Test Coordinator

The **test-coordinator** is the central module setting up and running the tests. Based on a configuration file (in YAML), it does the following:

- instantiate the selected connectors
- instantiate the selected auxiliaries
- provide the auxiliaries with the matching connectors
- generate the list of tests to perform
- provide the testcases with the auxiliaries they need
- verify if the tests can be performed
- flash and run and synchronize the tests on the auxiliaries
- gather the reports and publish the results

3.2.2 Auxiliary

The **auxiliary** provides to the **test-coordinator** an interface to interact with the physical or digital auxiliary target. It is composed by 2 blocks:

- physical or digital instance creation / deletion (e.g. flash the *device under test* with the testing software, e.g. Start a docker container)
- connectors to facilitate interaction and communication with the device (e.g. flashing via *JTAG*, messaging with *UART*)

In case of the specific *device under test* auxiliary, we have:

- As communication channel (**cchannel**) usually *UART*
- As flashing channel (**flashing**) usually *JTAG*

For other auxiliaries like the one interacting with cloud services, maybe we just have:

- A communication channel (**channel**) like *REST*

3.2.3 Connector

Communication Channel

The Communication Channel - also known as **cchannel** - is the medium to communicate with auxiliary target. Example include *UART*, *UDP*, *USB*, *REST*,... The communication protocol itself can be auxiliary specific. In case of the *device under test*, we have a specific communication protocol. Please see the next paragraph.

Flashing

The Flasher Connectors usually provide only one method, `Flasher.flash()`, which will transfer the configured binary file to the target.

3.2.4 Dynamic Import Linking

The *pykiso* framework was developed with modularity and reusability in mind. To avoid close coupling between test-cases and auxiliaries as well as between auxiliaries and connectors, the linking between those components is defined in a config file (see *Test Configuration File*) and performed by the *TestCoordinator*.

Different instances of connectors and auxiliaries are given *aliases* which identify them within the test session.

Let's say we have this (abridged) config file:

```
connectors:
  my_chan:           # Alias of the connector
    type: ...
auxiliaries:
  my_aux:            # Alias of the auxiliary
    connectors:
        com: my_chan # Reference to the connector
    type: ...
```

The auxiliary *my_aux* will automatically be initialised with *my_chan* as its *com* channel.

When writing your testcases, the auxiliary will then be available under its defined alias.

```
from pykiso.auxiliaries import my_aux
```

The *pykiso.auxiliaries* is a magic package that only exists in the *pykiso* package after the *TestCoordinator* has processed the config file. It will include all *instances* of the defined auxiliares, available at their defined alias.

3.2.5 Message Protocol (If in used)

The message protocol is used (but not only) between the *device under test* HW and its **test-auxiliary**. The communication pattern is as follows:

1. The test manager sends a message that contains a test command to a test participant.
2. The test participant sends an acknowledgement message back.
3. The test participant may send a report message.
4. The test manager replies to a report message with an acknowledgement message.

The message structure is as follow:

```
0               1               2               3
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Ver| MT|  Res  |   Msg Token   |   Sub-Type    | Error code    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Test Section  |  Test Suite   |   Test Case   | Payload length|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

(continues on next page)

(continued from previous page)

```
| Payload (in TLV format)
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

It consist of:

Code	size (in bytes)	Explanation
Ver (Version)	2 bits	Indicates the version of the test c oordination protocol.
MT (Message Type)	2 bits	Indicates the type of the message.
Res (Reserved)	4 bits	
Msg Token (Message Token)	1	Arbitrary byte. It must not be repeated for 10 consecutive messages. In the ackn owledgement message the same token must be used.
Sub-Type (Message Sub Type)	1	Gives more information about the message type
Error Code	1	Error code that can be used by the auxiliaries to forward an error
Test Section	1	Indicates the test section number
Test Suite	1	Indicates the test suite number which permits to identify a test suite within a test section
Test Case	1	Indicates the test case number which permits to identify a test case within a test suite
Payload Length	1	Indicate the length of the payload composed of TLV elements. If 0, it means there is no payload
Payload	X	Optional, list of TLVs elements. One TLV has 1 byte for the *Tag*, 1 byte for the *length*, up to 255 bytes for the *Value*

The **message type** and **message sub-type** are linked and can take the following values:

Type	Type Id	Sub-type	Sub-type Id	Ex planation
COM-MAND	0	PING	0	For ping-pong between the auxiliary to verify if a comm unication is es tablished
		TEST_SECTION_SETUP	1	
		TEST_SUITE_SETUP	2	
		TEST_CASE_SETUP	3	
		TEST_SECTION_RUN	11	
		TEST_SUITE_RUN	12	
		TEST_CASE_RUN	13	
		TE ST_SECTION_TEARDOWN	21	
		TEST_SUITE_TEARDOWN	22	
		TEST_CASE_TEARDOWN	23	
		ABORT	99	
RE-PORT	1	TEST_PASS	0	
		TEST_FAILED	1	
		TEST_NOT_IMPLEMENTED	2	
ACK	2	ACK	0	
		NACK	1	
LOG	3	RESERVED	0	

The TLV only supported *Tag* are:

- TEST_REPORT = 110
- FAILURE_REASON = 112

3.2.6 Flashing

The flashing is usually needed to put the test-software containing the tests we would like to run into the *Device under test* . Flashing is done via a flashing connector, which has to be configured with the correct binary file. The flashing connector is in turn called from an appropriate auxiliary (usually in its setup phase).

3.3 Usage

3.3.1 Flow

1. Create a root-folder that will contain the tests. Let us call it *test-folder*.
2. Create, based on your test-specs, one folder per test-suite.
3. In each test-suite folder, implement the tests. (See how below)
4. write a configuration file (see *Test Configuration File*)
5. If your test-setup is ready, run `pykiso -c <ROOT_TEST_DIR>`
6. If the tests fail, you will see it in the the output. For more details, you can take a look at the log file (logs to STDOUT as default).

3.3.2 Define the test information

For each test fixture (setup, teardown or test_run), users have to define the test information using the decorator define_test_parameters. This decorator gives access to the following parameters:

- suite_id : current test suite identification number
- case_id : current test case identification number (optional for test suite setup and teardown)
- aux_list : list of used auxiliaries

Based on Message Protocol, users can configure the maximum time (in seconds) used to wait for a report. This "timeout" is configurable for each available fixtures :

- setup_timeout : the maximum time (in seconds) used to wait for a report during setup execution (optional)
- run_timeout : the maximum time (in seconds) used to wait for a report during test_run execution (optional)
- teardown_timeout : the maximum time (in seconds) used to wait for a report during teardown execution (optional)

Note: by default those timeout values are set to 10 seconds.

In order to link the architecture requirement to the test, an additional reference can be added into the test_run decorator:
- test_ids: [optional] requirements has to be defined like follow:

{"Component1": ["Req1", "Req2"], "Component2": ["Req3"]}

Find below a full example for a test suite/case declaration :

```
"""
Add test suite setup fixture, run once at test suite's beginning.
Test Suite Setup Information:
-> suite_id : set to 1
-> case_id : Parameter case_id is not mandatory for setup.
-> aux_list : used aux1 and aux2 is used
-> setup_timeout : time to wait for a report 5 seconds
-> run_timeout : Parameter run_timeout is not mandatory for test suite setup.
-> teardown_timeout : Parameter run_timeout is not mandatory for test suite setup.
"""
  @pykiso.define_test_parameters(suite_id=1, aux_list=[aux1, aux2], setup_timeout=5)
  class SuiteSetup(pykiso.BasicTestSuiteSetup):
      pass

"""
Add test suite teardown fixture, run once at test suite's end.
Test Suite Teardown Information:
-> suite_id : set to 1
-> case_id : Parameter case_id is not mandatory for setup.
-> aux_list : used aux1 and aux2 is used
-> setup_timeout : Parameter run_timeout is not mandatory for test suite teardown.
-> run_timeout : Parameter run_timeout is not mandatory for test suite teardown.
-> teardown_timeout : time to wait for a report 5 seconds
"""
  @pykiso.define_test_parameters(suite_id=1, aux_list=[aux1, aux2], teardown_timeout=5,)
  class SuiteTearDown(pykiso.BasicTestSuiteTeardown):
      pass

"""
Add a test case 1 from test suite 1 using auxiliary 1.
  Test Suite Teardown Information:
-> suite_id : set to 1
-> case_id : set to 1
-> aux_list : used aux1 and aux2 is used
-> setup_timeout : time to wait for a report 3 seconds during setup
-> run_timeout : time to wait for a report 10 seconds during test_run
-> teardown_timeout : time to wait for a report 3 seconds during teardown
-> test_ids: [optional] store the requirements into the report
"""
  @pykiso.define_test_parameters(
          suite_id=1, case_id=1, aux_list=[aux1, aux2], setup_timeout=3,
          run_timeout=10, teardown_timeout=3,
          test_ids={"Component1": ["Req1", "Req2"]}
  )
  class MyTest(pykiso.BasicTest):
      pass
```

3.3.3 Implementation of Basic Tests

Structure: *test-folder/test-suite-1/***test_suite_1.py**

test_suite_1.py:

```
 """
 I want to run the following tests documented in the following test-specs <TEST_CASE_
↪SPECS>.
 """
  import pykiso
  from pykiso.auxiliaries import aux1, aux2

"""
Add test suite setup fixture, run once at test suite's beginning.
Parameter case_id is not mandatory for setup.
"""
  @pykiso.define_test_parameters(suite_id=1, aux_list=[aux1, aux2], setup_timeout=1, run_
↪timeout=2, teardown_timeout=3)
  class SuiteSetup(pykiso.BasicTestSuiteSetup):
      pass

"""
Add test suite teardown fixture, run once at test suite's end.
Parameter case_id is not mandatory for teardown.
"""
  @pykiso.define_test_parameters(suite_id=1, aux_list=[aux1, aux2])
  class SuiteTearDown(pykiso.BasicTestSuiteTeardown):
      pass

"""
Add a test case 1 from test suite 1 using auxiliary 1.
"""
  @pykiso.define_test_parameters(suite_id=1, case_id=1, aux_list=[aux1])
  class MyTest(pykiso.BasicTest):
      pass

"""
Add a test case 2 from test suite 1 using auxiliary 2.
"""
  @pykiso.define_test_parameters(suite_id=1, case_id=2, aux_list=[aux2])
  class MyTest2(pykiso.BasicTest):
      pass
```

3.3.4 Implementation of Advanced Tests - Auxiliary Interaction

Using the dynamic importing capabilities of the framework we can interact with the auxiliaries directly.

For this test we will assume that we have configured a *pykiso.lib.auxiliaries.communication_auxiliary.CommunicationAuxiliary* and a connector that supports *raw* messaging.

```
"""
send a message, receive a response, compare to expected response
"""
import pykiso
from pykiso.auxiliaries import com_aux

@pykiso.define_test_parameters(suite_id=2, case_id=1, aux_list=[com_aux])
class ComTest(pykiso.BasicTest):

    STIMULUS = b"stimulus message"
    RESPONSE = b"expected reply"

    def test_run(self):
        com_aux.send_message(STIMULUS)
        resp = com_aux.receive_message()
        self.assertEqual(resp, RESPONSE)
```

We can use the configured and instantiated auxiliary *com_aux* (imported by it's alias) in the test directly.

3.3.5 Implementation of Advanced Tests - Custom Setup

If you need to have more complex tests, you can do the following:

- `BasicTest` is a specific implementation of `unittest.TestCase` therefore it contains 3 steps/methods **setUp()**, **tearDown()** and **test_run()** that can be overwritten.
- `BasicTest` will contain the list of **auxiliaries** you can use. It will be hold in the attribute `test_auxiliary_list`.
- `BasicTest` also contains the following information `test_section_id`, `test_suite_id`, `test_case_id`.
- Import *logging* or/and *message* (if needed) to communicate with the **auxiliary**

test_suite_2.py:

```
"""
I want to run the following tests documented in the following test-specs <TEST_CASE_
↪SPECS>.
"""
import pykiso
from pykiso import message
from pykiso.auxiliaries import aux1


@pykiso.define_test_parameters(suite_id=2, case_id=1, aux_list=[aux1])
class MyTest(pykiso.BasicTest):
    def setUp(self):
       # I loop through all the auxiliaries
       for aux in self.test_auxiliary_list:
```

(continues on next page)

(continued from previous page)

```
            if aux.name == "aux1": # If I find the auxiliary to which I need to send a␣
↪special message, I compose the message and send it.
                # Compose the message to send with some additional information
                tlv = { TEST_REPORT:"Give me something" }
                testcase_setup_special_message = message.Message(msg_type=message.
↪MessageType.COMMAND, sub_type=message.MessageCommandType.TEST_CASE_SETUP,
                                                                 test_section=self.test_section_id,
↪ test_suite=self.test_suite_id, test_case=self.test_case_id, tlv_dict=tlv)
                # Send the message
                aux.run_command(testcase_setup_special_message, blocking=True, timeout_in_
↪s=10)
            else: # Do not forget to send a setup message to the other auxiliaries!
                # Compose the normal message
                testcase_setup_basic_message = message.Message(msg_type=message.
↪MessageType.COMMAND, sub_type=message.MessageCommandType.TEST_CASE_SETUP,
                                                               test_section=self.test_section_id,
↪ test_suite=self.test_suite_id, test_case=self.test_case_id)
                # Send the message
                aux.run_command(testcase_setup_basic_message, blocking=True, timeout_in_
↪s=10)
```

3.3.6 Implementation of Advanced Tests - Test Templates

Because we are python based, you can until some extend, design and implement parts of the framework to fulfil your needs. For example:

test_suite_3.py:

```
import pykiso
from pykiso import message
from pykiso.auxiliaries import aux1


class MyTestTemplate(pykiso.BasicTest):
   def test_run(self):
       # Prepare message to send
       testcase_run_message = message.Message(msg_type=message.MessageType.COMMAND, sub_
↪type=message.MessageCommandType.TEST_CASE_RUN,
                                              test_section=self.test_section_id,␣
↪test_suite=self.test_suite_id, test_case=self.test_case_id)
       # Send test start through all auxiliaries
       for aux in self.test_auxiliary_list:
           if aux.run_command(testcase_run_message, blocking=True, timeout_in_s=10) is␣
↪not True:
               self.cleanup_and_skip("{} could not be run!".format(aux))
       # Device will reboot, wait for the reboot report
       for aux in self.test_auxiliary_list:
           if aux.name == "DeviceUnderTest":
               report = aux.wait_and_get_report(blocking=True, timeout_in_s=10) # Wait␣
↪for a report from the DeviceUnderTest
               break
```

(continues on next page)

(continued from previous page)

```
        # Check if the report for the reboot was received.
        report is not None and report.get_message_type() == message.MessageType.REPORT␣
↪and report.get_message_sub_type() == message.MessageReportType.TEST_PASS:
            pass # We can continue
        else:
            self.cleanup_and_skip("Device failed rebooting")
        # Loop until all reports are received
        list_of_aux_with_received_reports = [False]*len(self.test_auxiliary_list)
        while False in list_of_aux_with_received_reports:
            # Loop through all auxiliaries
            for i, aux in enumerate(self.test_auxiliary_list):
                if list_of_aux_with_received_reports[i] == False:
                    # Wait for a report
                    reported_message = aux.wait_and_get_report()
                    # Check the received message
                    list_of_aux_with_received_reports[i] = self.evaluate_message(aux,␣
↪reported_message)

@pykiso.define_test_parameters(suite_id=3, case_id=1, aux_list=[aux1])
class MyTest(MyTestTemplate):
   pass

@pykiso.define_test_parameters(suite_id=3, case_id=2, aux_list=[aux1])
class MyTest2(MyTestTemplate):
   pass
```

3.3.7 Add Config File

For details see *Test Configuration File*.

Example:

```
auxiliaries:
  aux1:
    connectors:
        com: chan1
    config: null
    type: ext_lib/example_test_auxiliary.py:ExampleAuxiliary
  aux2:
    connectors:
        com:   chan2
        flash: chan3
    type: pykiso.lib.auxiliaries.example_test_auxiliary:ExampleAuxiliary
  aux3:
    connectors:
        com:   chan4
    type: pykiso.lib.auxiliaries.dut_auxiliary:DUTAuxiliary
connectors:
  chan1:
    config: null
    type: ext_lib/cc_example.py:CCExample
```

(continues on next page)

(continued from previous page)

```
  chan2:
    type: ext_lib/cc_example.py:CCExample
  chan4:
    type: ext_lib/cc_example.py:CCExample
  chan3:
    config:
        configKey: "config value"
    type: ext_lib/cc_example.py:CCExample
test_suite_list:
- suite_dir: test_suite_1
  test_filter_pattern: '*.py'
  test_suite_id: 1
- suite_dir: test_suite_2
  test_filter_pattern: '*.py'
  test_suite_id: 2
```

3.3.8 Run the tests

```
pykiso -c <config_file>
```

CHAPTER

FOUR

TEST CONFIGURATION FILE

The test configuration files are written in YAML.

Let's use an example to understand the structure.

```yaml
auxiliaries:
  aux1:
    connectors:
        com: chan1
    config: null
    type: ext_lib/example_test_auxiliary.py:ExampleAuxiliary
  aux2:
    connectors:
        com:   chan2
        flash: chan3
    type: pykiso.lib.auxiliaries.example_test_auxiliary:ExampleAuxiliary
  aux3:
    connectors:
        com:   chan4
    type: pykiso.lib.auxiliaries.dut_auxiliary:DUTAuxiliary
connectors:
  chan1:
    config: null
    type: ext_lib/cc_example.py:CCExample
  chan2:
    type: ext_lib/cc_example.py:CCExample
  chan4:
    type: ext_lib/cc_example.py:CCExample
  chan3:
    config:
        configKey: "config value"
    type: ext_lib/cc_example.py:CCExample
test_suite_list:
- suite_dir: test_suite_1
  test_filter_pattern: '*.py'
  test_suite_id: 1
- suite_dir: test_suite_2
  test_filter_pattern: '*.py'
  test_suite_id: 2
```

4.1 Connectors

The connector definition is a named list (dictionary in python) of key-value pairs, namely config and type.

```
  aux3:
    connectors:
        com:   chan4
    type: pykiso.lib.auxiliaries.dut_auxiliary:DUTAuxiliary
connectors:
  chan1:
    config: null
    type: ext_lib/cc_example.py:CCExample
  chan2:
    type: ext_lib/cc_example.py:CCExample
```

The channel alias will identify this configuration for the auxiliaries.

The config can be omitted, *null*, or any number of key-value pairs.

The type consists of a module location and a class name that is expected to be found in the module. The location can be a path to a python file (Win/Linux, relative/absolute) or a python module on the python path (e.h. *pykiso.lib.connectors.cc_uart*).

```
<chan>:                     # channel alias
    config:                 # channel config, optional
        <key>: <value>      # collection of key-value pairs, e.g. "port: 80"
    type: <module:Class>    # location of the python class that represents this channel
```

4.2 Auxiliaries

The auxiliary definition is a named list (dictionary in python) of key-value pairs, namely config, connectors and type.

```
auxiliaries:
  aux1:
    connectors:
        com: chan1
    config: null
    type: ext_lib/example_test_auxiliary.py:ExampleAuxiliary
  aux2:
    connectors:
        com:   chan2
        flash: chan3
    type: pykiso.lib.auxiliaries.example_test_auxiliary:ExampleAuxiliary
```

The auxiliary alias will identify this configuration for the testcases. When running the tests the testcases can import an auxiliary instance defined here using

```
from pykiso.auxiliaries import <alias>
```

The connectors can be omitted, *null*, or any number of role-connector pairs. The roles are defined in the auxiliary implementation, usual examples are *com* and *flash*. The channel aliases are the ones you defined in the connectors section above.

The config can be omitted, *null*, or any number of key-value pairs.

The type consists of a module location and a class name that is expected to be found in the module. The location can be a path to a python file (Win/Linux, relative/absolute) or a python module on the python path (e.h. *pykiso.lib.auxiliaries.communication_auxiliary*).

```
<aux>:                          # aux alias
    connectors:                 # list of connectors this auxiliary needs
        <role>: <channel-alias> # <role> has to be the name defined in the Auxiliary␣
↪class,
                                #  <channel-alias> is the alias defined above
    config:                     # channel config, optional
        <key>: <value>          # collection of key-value pairs, e.g. "port: 80"
    type: <module:Class>        # location of the python class that represents this␣
↪auxiliary
```

4.3 Test Suites

The test suite definition is a list of key-value pairs.

```
  chan4:
    type: ext_lib/cc_example.py:CCExample
  chan3:
    config:
        configKey: "config value"
    type: ext_lib/cc_example.py:CCExample
test_suite_list:
- suite_dir: test_suite_1
  test_filter_pattern: '*.py'
  test_suite_id: 1
- suite_dir: test_suite_2
  test_filter_pattern: '*.py'
  test_suite_id: 2
```

Each test suite consists of a *test_suite_id*, a *suite_dir* and a *test_filter_pattern*.

4.4 Real-World Configuration File

```
auxiliaries:
  DUT:
    connectors:
        com: uart
    config: null
    type: pykiso.lib.auxiliaries.example_test_auxiliary:ExampleAuxiliary
connectors:
  uart:
    config:
      serialPort: COM3
      type: pykiso.lib.auxiliaries.cc_uart:CCUart
test_suite_list:
- suite_dir: test_suite_1
```

(continues on next page)

(continued from previous page)

```
14   test_filter_pattern: '*.py'
15   test_suite_id: 1
```

4.5 Deactivation of specific loggers

By default, every logger that does not belong to the *pykiso* package or that is not an *auxiliary* logger will see its level set to WARNING even if you have in the command line *pykiso –log-level DEBUG*. This aims to reduce redundant logs from additional modules during the test execution. For keeping specific loggers to the set log-level, it is possible to set the *activate_log* parameter in the *auxiliary* config. The following example deactivates the *jlink* logger from the *pylink* package, imported in *cc_rtt_segger.py*:

auxiliaries:

aux1:

connectors: com: rtt_channel

config: activate_log: # only specifying pylink will include child loggers - pylink.jlink - my_pkg

type: pykiso.lib.auxiliaries.dut_auxiliary:DUTAuxiliary

connectors:

rtt_channel: config: null type: pykiso.lib.connectors.cc_rtt_segger:CCRttSegger

Based on this example, by specifying *my_pkg*, all child loggers will also be set to the set log-level.

Note: If e.g. only the logger *my_pkg.module_1* should be set to the level, it should be entered as such.

4.6 Ability to use environment variables

It is possible to replace any value by an environment variable in the YAML files. When using environment variables, the following format should be respected: *ENV{my-env-var}*. In the following example, an environment variable called *TEST_SUITE_1* contains the path to the test suite 1 directory.

```
test_suite_list:
- suite_dir: ENV{TEST_SUITE_1}
  test_filter_pattern: '*.py'
  test_suite_id: 1
```

4.7 Specify files and folders

To specify files and folders you can use absolute or relative paths. Relative paths are always given relative to the location of the yaml file.

Relative path or file locations must always start with "./"

```yaml
example_config:
    rel_script_path: './script_folder/my_awesome_script.py'
    abs_script_path_win: 'C:/script_folder/my_awesome_script.py'
    abs_script_path_unix: '/home/usr/script_folder/my_awesome_script.py'
```

4.8 Make a proxy auxiliary trace

Proxy auxiliary is capable of creating a trace file, where all received messages at connector level are written. This feature is useful when proxy auxiliary is associated with a connector who doesn't have any trace capability (in contrast to cc_pcan_can or cc_rtt_segger for example).

Everything is handled at configuration level and especially at yaml file :

```yaml
proxy_aux:
  connectors:
    # communication channel alias
    com: <channel-alias>
  config:
    # Auxiliaries alias list bound to proxy auxiliary
    aux_list : [<aux alias 1>, <aux alias 2>, <aux alias 3>]
    # activate trace at proxy level, sniff everything received at
    # connector level and write it in .log file.
    activate_trace : True
    # by default the trace is placed where pykiso is launched
    # otherwise user should specify his own path
    # (absolute and relative)
    trace_dir: ./suite_proxy
    # by default the trace file's name is :
    # YY-MM-DD_hh-mm-ss_proxy_logging.log
    # otherwise user should specify his own name
    trace_name: can_trace
  type: pykiso.lib.auxiliaries.proxy_auxiliary:ProxyAuxiliary
```

CHAPTER

FIVE

INCLUDED CONNECTORS

pykiso comes with some ready to use implementations of different connectors.

5.1 collection of ready to use connectors

module connectors

5.2 Flashers

5.2.1 JLink Flasher

module flash_jlink

synopsis a Flasher adapter of the pylink-square library

class pykiso.lib.connectors.flash_jlink.**JLinkFlasher**(*binary=None*, *lib=None*, *serial_number=None*, *chip_name='STM32L562QE'*, *speed=9600*, *verbose=False*, *power_on=False*, *start_addr=0*, *xml_path=None*, ***kwargs*)

A Flasher adapter of the pylink-square library.

Constructor.

Parameters

- **binary** (Union[str, Path, None]) – path to the binary firmware file
- **lib** (Union[str, Path, None]) – path to the location of the JLink.so/JLink.DLL, usually automatically determined
- **serial_number** (Optional[int]) – optional debugger's S/N (required if many connected) (see pylink-square documentation)
- **chip_name** (str) – see pylink-square documentation
- **speed** (int) – see pylink-square documentation
- **verbose** (bool) – see pylink-square documentation
- **power_on** (bool) – see pylink-square documentation
- **start_addr** (int) – see pylink-square documentation
- **xml_path** (Optional[str]) – device configuration (see pylink-square documentation)

close()
Close flasher and free resources.

Return type None

flash()
¨Perform firmware delivery.

Return type None

open()
Initialize the flasher.

Return type None

5.2.2 Lauterbach Flasher

module flash_lauterbach

synopsis used to flash through lauterbach probe.

class pykiso.lib.connectors.flash_lauterbach.**LauterbachFlasher**(*t32_exc_path=None*, *t32_config=None*, *t32_script_path=None*, *t32_api_path=None*, *port=None*, *node='localhost'*, *packlen='1024'*, *device=1*, ***kwargs*)

Connector used to flash through one and only one Lauterbach probe using Trace32 as remote API.

Initialize attributes with configuration data.

Parameters

- **t32_exc_path** (Optional[str]) – full path of Trace32 app to execute
- **t32_config** (Optional[str]) – full path of Trace32 configuration file
- **t32_script_path** (Optional[str]) – full path to .cmm flash script to execute
- **t32_api_path** (Optional[str]) – full path of remote api
- **port** (Optional[str]) – port number used for UDP communication
- **node** (str) – node name (default localhost)
- **packlen** (str) – data pack length for UDP communication (default 1024)
- **device** (int) – configure device number given by Trace32 (default 1)

close()
Close UDP socket and shut down Trace32 App.

Return type None

flash()
Flash software using configured .cmm script.

The Flash command leads to the following sub-tasks execution :

- Send to Trace32 CD.DO internal command (execute script)
- Wait until script is finished
- Get script execution verdict

Raises `Exception` – if Trace32 error occurred during flash.

Return type `None`

`open()`
Open UDP socket between ITF and Trace32 loaded app.

The open command leads to the following sub-tasks execution:

- Open a Trace32 app
- Load remote API using ctypes
- Configure UPD channel (Port/buffer size…)
- Open UDP connection
- Make a ping request

Return type `None`

`class pykiso.lib.connectors.flash_lauterbach.MessageLineState(value)`
Use to determine Message reading command.

`class pykiso.lib.connectors.flash_lauterbach.ScriptState(value)`
Use to determine script command execution.

5.3 CChannels

5.3.1 Virtual Communication Channel for tests

module cc_example

`class pykiso.lib.connectors.cc_example.CCExample(name=None, **kwargs)`
Only use for development purpose.

This channel simply handle basic TestApp response mechanism.

Initialize attributes.

Parameters `name` (`Optional[str]`) – name of the communication channel

5.3.2 Loopback CChannel

module cc_raw_loopback

synopsis Loopback CChannel for testing purposes.

`class pykiso.lib.connectors.cc_raw_loopback.CCLoopback(**kwargs)`
Loopback CChannel for testing purposes.

Whatever gets sent via cc_send will land in a FIFO and can be received via cc_receive.

constructor

5.3.3 Communication Channel Via Uart

module cc_uart

synopsis Uart communication channel

class pykiso.lib.connectors.cc_uart.**CCUart**(*serialPort*, *baudrate=9600*, ***kwargs*)
UART implementation of the coordination channel.

constructor

exception pykiso.lib.connectors.cc_uart.**IncompleteCCMsgError**(*value*)

5.3.4 Communication Channel Via Udp

module cc_udp

synopsis Udp communication channel

class pykiso.lib.connectors.cc_udp.**CCUdp**(*dest_ip*, *dest_port*, ***kwargs*)
UDP implementation of the coordination channel.

Initialize attributes.

Parameters

- **dest_ip** (str) – destination ip address
- **dest_port** (int) – destination port

5.3.5 Communication Channel via UDP server

module cc_udp_server

synopsis basic UDP server

Warning: if multiple clients are connected to this server, ensure that each client receives all necessary responses before receiving messages again. Otherwise the responses may be sent to the wrong client

class pykiso.lib.connectors.cc_udp_server.**CCUdpServer**(*dest_ip*, *dest_port*, ***kwargs*)
Connector channel used to set up an UDP server.

Initialize attributes.

Parameters

- **dest_ip** (str) – destination port
- **dest_port** (int) – destination port

5.3.6 Communication Channel Via Usb

module cc_usb

synopsis Usb communication channel

class pykiso.lib.connectors.cc_usb.**CCUsb**(*serial_port*)
constructor

5.3.7 Communication Channel Via lauterbach

module cc_fdx_lauterbach

synopsis CChannel implementation for lauterbach(FDX)

class pykiso.lib.connectors.cc_fdx_lauterbach.**CCFdxLauterbach**(*t32_exc_path=None*, *t32_config=None*, *t32_main_script_path=None*, *t32_reset_script_path=None*, *t32_fdx_clr_buf_script_path=None*, *t32_in_test_reset_script_path=None*, *t32_api_path=None*, *port=None*, *node='localhost'*, *packlen='1024'*, *device=1*, ***kwargs*)

Lauterbach connector using the FDX protocol.

Constructor: initialize attributes with configuration data.

Parameters

- **t32_exc_path** (Optional[str]) – full path of Trace32 app to execute
- **t32_config** (Optional[str]) – full path of Trace32 configuration file
- **t32_main_script_path** (Optional[str]) – full path to the main cmm script to execute
- **t32_reset_script_path** (Optional[str]) – full path to the reset cmm script to execute
- **t32_fdx_clr_buf_script_path** (Optional[str]) – full path to the FDX reset cmm script to execute
- **t32_in_test_reset_script_path** (Optional[str]) – full path to the board reset cmm script to execute
- **t32_api_path** (Optional[str]) – full path of remote api
- **port** (Optional[str]) – port number used for UDP communication
- **node** (str) – node name (default localhost)
- **packlen** (str) – data pack length for UDP communication (default 1024)
- **device** (int) – configure device number given by Trace32 (default 1)

load_script(*script_path*)
Load a cmm script.

Parameters **script_path** (str) – cmm file path

Returns error status

reset_board()
Executes the board reset.

Return type None

start()
Override clicking on "go" in the Trace32 application.

The channel must have been successfully opened (Trace32 application opened and script loaded).

Return type None

class pykiso.lib.connectors.cc_fdx_lauterbach.**PracticeState**(*value*)
Available state for any scripts loaded into TRACE32.

5.3.8 Communication Channel Via segger j-link

module cc_rtt_segger

synopsis channel used to enable RTT communication using Segger J-Link debugger. Additionally, RTT logs can be captured by setting the rtt_log_path parameter on the specified channel.

class pykiso.lib.connectors.cc_rtt_segger.**CCRttSegger**(*serial_number=None*, *chip_name='STM32L562QE'*, *speed=4000*, *block_address=537131008*, *verbose=False*, *tx_buffer_idx=3*, *rx_buffer_idx=0*, *rtt_log_path=None*, *rtt_log_buffer_idx=0*, *connection_timeout=5*, ***kwargs*)
Channel using RTT to communicate through Segger J-Link debugger.

Initialize attributes.

Parameters

- **serial_number** (Optional[int]) – optional segger debugger serial number (required if many connected)
- **chip_name** (str) – microcontoller name (STM....)
- **speed** (int) – communication speed in Hz
- **block_address** (int) – start address to start RTT communication
- **tx_buffer_idx** (int) – buffer index used for transmission
- **rx_buffer_idx** (int) – buffer index used for reception
- **verbose** (bool) – boolean indicating if J-Link connection should be verbose in logging
- **rtt_log_path** (Optional[str]) – path to the folder where the RTT log file should be stored
- **rtt_log_buffer_idx** (int) – buffer index used for RTT logging
- **connection_timeout** (int) – available time (in seconds) to open the connection

receive_log()
Receive RTT log messages from the corresponding RTT buffer.

Return type None

5.3.9 Proxy Channel

module cc_proxy

synopsis CChannel implementation for multi-auxiliary usage.

CCProxy channel was created, in order to enable the connection of multiple auxiliaries on one and only one CChannel. This CChannel has to be used with a so called proxy auxiliary.

class pykiso.lib.connectors.cc_proxy.**CCProxy**(***kwargs*)
Proxy CChannel for multi auxiliary usage.

Initialize attributes.

5.3.10 Communication Channel using VISA protocol

module cc_visa

synopsis VISA communication channel to communicate to instruments using SCPI protocol.

class pykiso.lib.connectors.cc_visa.**VISAChannel**(***kwargs*)
VISA Interface for devices communicating with SCPI

Initialize channel settings.

query(*query_command*)
Send a query request to the instrument

Parameters **query_command** (str) – query command to send

Return type str

Returns Response message, None if the request expired with a timeout.

class pykiso.lib.connectors.cc_visa.**VISASerial**(*serial_port*, *baud_rate=9600*, ***kwargs*)
Connector used to communicate with an instrument via Serial.

Initialize channel attributes.

Parameters

- **serial_port** (int) – COM port to use to connect to the instrument
- **baud_rate** – baud rate used to communicate with the instrument

class pykiso.lib.connectors.cc_visa.**VISATcpip**(*ip_address*, *protocol='INSTR'*, ***kwargs*)
Connector used to communicate with an instrument via TCPIP

Initialize channel attributes.

Parameters

- **ip_address** (str) – target instrument's ip address
- **protocol** – communication protocol to use

5.3.11 Communication Channel via socket

module cc_socket

synopsis connector for communication via socket

class pykiso.lib.connectors.cc_tcp_ip.**CCTcpip**(*dest_ip*, *dest_port*, *max_msg_size=256*, ***kwargs*)
Connector channel used to communicate via socket

Initialize channel settings.

Parameters

- **dest_ip** (str) – destination ip address
- **dest_port** (int) – destination port
- **max_msg_size** (int) – the maximum amount of data to be received at once

CHAPTER

SIX

INCLUDED AUXILIARIES

pykiso comes with some ready to use implementations of different auxiliaries.

6.1 collection of ready to use auxiliaries

module auxiliaries

6.2 CommunicationAuxiliary

module communication_auxiliary

synopsis Auxiliary used to send raw bytes via a connector instead of pykiso.Messages

class pykiso.lib.auxiliaries.communication_auxiliary.**CommunicationAuxiliary**(*com*, ***kwargs*)

Auxiliary used to send raw bytes via a connector instead of pykiso.Messages.

Constructor.

Parameters **com** (*CChannel*) – CChannel that supports raw communication

receive_message(*blocking=True*, *timeout_in_s=None*)

Receive a raw message.

Parameters

- **blocking** (bool) – wait for message till timeout elapses?
- **timeout_in_s** (Optional[float]) – maximum time in second to wait for a response

Return type bytes

Returns raw message

send_message(*raw_msg*)

Send a raw message (bytes) via the communication channel.

Parameters **raw_msg** (bytes) – message to send

Return type bool

Returns True if command was executed otherwise False

6.3 Example Auxiliary

module example_test_auxiliary

synopsis Example auxiliary that simulates a normal test run without ever doing anything.

Warning: Still under test

class pykiso.lib.auxiliaries.example_test_auxiliary.**ExampleAuxiliary**(*name=None*, *com=None*, *flash=None*, ***kwargs*)

Example of an auxiliary implementation.

Constructor.

Parameters

- **name** – Alias of the auxiliary instance
- **com** – Communication connector
- **flash** – flash connector

6.4 Virtual DUT simulation package

module simulated_auxiliary

synopsis provide a simple interface to simulate a device under test

This auxiliary can be used as a simulated version of a device under test.

The intention is to set up a pair of CChannels like a pipe, for example a *CCUdpServer* and a *CCUdp* bound to the same address. One side of this pipe is then connected to this virtual auxiliary, the other one to a *real* auxiliary.

The `SimulatedAuxiliary` will then receive messages from the real auxiliary just like a proper TestApp on a DUT would and answer them according to a predefined playbook.

Each predefined playbooks are linked with real auxiliary received messages, using test case and test suite ids (see *simulation*). A so called playbook, is a basic list of different *Message* instances where the content is adapted to the current context under test (simulate a communication lost, a test case run failure…). (see *scenario*). In order to increase playbook configuration flexibility, predefined and reusable responses are located into *response_templates*.

pykiso.lib.auxiliaries.simulated_auxiliary.simulated_auxiliary	Simulated Auxiliary
pykiso.lib.auxiliaries.simulated_auxiliary.simulation	Simulation
pykiso.lib.auxiliaries.simulated_auxiliary.scenario	Scenario
pykiso.lib.auxiliaries.simulated_auxiliary.response_templates	ResponseTemplates

6.4.1 Simulated Auxiliary

module simulated_auxiliary

synopsis auxiliary used to simulate a virtual Device Under Test(DUT)

class pykiso.lib.auxiliaries.simulated_auxiliary.simulated_auxiliary.**SimulatedAuxiliary**(*com=None*, ***kwargs*)

Custom auxiliary use to simulate a virtual DUT.

Initialize attributes.

Parameters **com** – configured channel

6.4.2 Simulation

module simulation

synopsis map virtual DUT behavior with test case/suite id

Warning: Still under test

class pykiso.lib.auxiliaries.simulated_auxiliary.simulation.**Simulation**

Simulate a virtual DUT, by playing pre-defined scenario depending on test case and test suite id.

Initialize attributes and mapping.

get_scenario(*test_suite_id*, *test_case_id*)

Return the selected scenario mapped with the received test case and test suite id.

Parameters

- **test_suite_id** (int) – current test suite id
- **test_case_id** (int) – current test case id

Return type *Scenario*

Returns scenario instance containing all steps

handle_default_response()

Return a scenario to handle DUT default behavior.

Return type *Scenario*

Returns scenario instance containing all steps

handle_ping_pong()

Return a scenario to handle init ping pong exchange.

Return type *Scenario*

Returns scenario instance containing all steps

6.4.3 Scenario

module scenario

synopsis base object used to create pre-defined virtual DUT scenario.

> **Warning:** Still under test

class pykiso.lib.auxiliaries.simulated_auxiliary.scenario.**Scenario**(*initlist=None*)
Container used to create pre-defined virtual DUT scenario.

class pykiso.lib.auxiliaries.simulated_auxiliary.scenario.**TestScenario**
Encapsulate all possible test's scenarios.

class **VirtualTestCase**
Used to gather all virtual DUT test case scenarios based on their fixture level (setup, run, teardown).

class **Run**
Used to gather all possible scenarios linked to a test case run execution.

classmethod **handle_failed_report_run**()
Return a scenario to handle a complete test case with failed report at run phase.
Return type *Scenario*
Returns Scenario instance containing all steps

classmethod **handle_failed_report_run_with_log**()
Return a scenario to handle a complete test case with failed log and report at run phase.
Return type *Scenario*
Returns Scenario instance containing all steps

classmethod **handle_lost_communication_during_run_ack**()
Return a scenario to handle a complete test case with lost of communication during ACK to run Command.
Return type *Scenario*
Returns Scenario instance containing all steps

classmethod **handle_lost_communication_during_run_report**()
Return a scenario to handle a complete test case with lost of communication during report to run Command.
Return type *Scenario*
Returns Scenario instance containing all steps

classmethod **handle_not_implemented_report_run**()
Return a scenario to handle a complete test case with not implemented report at run phase.
Return type *Scenario*
Returns Scenario instance containing all steps

classmethod **handle_successful_report_run_with_log**()
Return a scenario to handle a complete test case with successful log and report at run phase.
Return type *Scenario*
Returns Scenario instance containing all steps

class **Setup**
Used to gather all possible scenarios linked to a test case setup execution.

classmethod **handle_failed_report_setup**()
Return a scenario to handle a complete test case with failed report at setup phase.
Return type *Scenario*

Returns Scenario instance containing all steps

classmethod handle_lost_communication_during_setup_ack()
Return a scenario to handle a complete test case with lost of communication during ACK to setup Command.
Return type *Scenario*
Returns Scenario instance containing all steps

classmethod handle_lost_communication_during_setup_report()
Return a scenario to handle a complete test case with lost of communication during report to setup Command.
Return type *Scenario*
Returns Scenario instance containing all steps

classmethod handle_not_implemented_report_setup()
Return a scenario to handle a complete test case with not implemented report at setup phase.
Return type *Scenario*
Returns Scenario instance containing all steps

class Teardown
Used to gather all possible scenarios linked to a test case teardown execution.

classmethod handle_failed_report_teardown()
Return a scenario to handle a complete test case with failed report at teardown phase.
Return type *Scenario*
Returns Scenario instance containing all steps

classmethod handle_lost_communication_during_teardown_ack()
Return a scenario to handle a complete test case with lost of communication during ACK to teardown Command.
Return type *Scenario*
Returns Scenario instance containing all steps

classmethod handle_lost_communication_during_teardown_report()
Return a scenario to handle a complete test case with lost of communication during report to teardown Command.
Return type *Scenario*
Returns Scenario instance containing all steps

classmethod handle_not_implemented_report_teardown()
Return a scenario to handle a complete test case with not implemented report at teardown phase.
Return type *Scenario*
Returns Scenario instance containing all steps

class VirtualTestSuite
Used to gather all virtual DUT test suite scenarios based on their fixture level (setup, teardown).

class Setup
Used to gather all possible scenarios linked to a test suite setup execution.

classmethod handle_failed_report_setup()
Return a scenario to handle a test suite setup with report failed.
Return type *Scenario*
Returns Scenario instance containing all steps

classmethod handle_lost_communication_during_setup_ack()
Return a scenario to handle a lost of communication during ACK to setup command.
Return type *Scenario*
Returns Scenario instance containing all steps

classmethod handle_lost_communication_during_setup_report()
Return a scenario to handle a lost of communication during report to setup command.
Return type *Scenario*
Returns Scenario instance containing all steps

classmethod handle_not_implemented_report_setup()
Return a scenario to handle a test suite setup with report not implemented.
Return type *Scenario*
Returns Scenario instance containing all steps

class Teardown
Used to gather all possible scenarios linked to a test suite teardown execution.

classmethod handle_failed_report_teardown()
Return a scenario to handle a test suite teardown with report failed.
Return type *Scenario*
Returns Scenario instance containing all steps

classmethod handle_lost_communication_during_teardown_ack()
Return a scenario to handle a lost of communication during ACK to teardown command.
Return type *Scenario*
Returns Scenario instance containing all steps

classmethod handle_lost_communication_during_teardown_report()
Return a scenario to handle a lost of communication during report to teardown command.
Return type *Scenario*
Returns Scenario instance containing all steps

classmethod handle_not_implemented_report_teardown()
Return a scenario to handle a test suite teardown with report not implemented.
Return type *Scenario*
Returns Scenario instance containing all steps

classmethod handle_successful()
Return a scenario to handle a complete successful test case exchange(TEST CASE setup->run->teardown).

Return type *Scenario*

Returns Scenario instance containing all steps

6.4.4 ResponseTemplates

module response_templates

synopsis Used to create a set of predefined messages

Warning: Still under test

class pykiso.lib.auxiliaries.simulated_auxiliary.response_templates.**ResponseTemplates**
Used to create a set of predefined messages (ACK, NACK, REPORT ...).

classmethod ack(*msg*)
Return an acknowledgment message.

Parameters **msg** (*Message*) – current received message

Return type List[*Message*]

Returns list of Message

classmethod ack_with_logs_and_report_nok(*msg*)
Return an acknowledge message and log messages and report message with verdict failed + tlv part with failure reason.

param msg current received message

return list of Message

Return type List[*Message*]

classmethod ack_with_logs_and_report_ok(*msg*)
Return an acknowledge message and log messages and report message with verdict pass.

param msg current received message

return list of Message

Return type List[*Message*]

classmethod ack_with_report_nok(*msg*)
Return an acknowledgment message and a report message with verdict failed + tlv part with failure reason.

Parameters **msg** (*Message*) – current received message

Return type List[*Message*]

Returns list of Message

classmethod ack_with_report_not_implemented(*msg*)
Return an acknowledge message and a report message with verdict test not implemented.

Parameters **msg** (*Message*) – current received message

Return type List[*Message*]

Returns list of Message

classmethod ack_with_report_ok(*msg*)
Return an acknowledgment message and a report message with verdict pass.

Parameters **msg** (*Message*) – current received message

Return type List[*Message*]

Returns list of Message

classmethod default(*msg*)
handle default response, if not test case/suite run just return ACK message otherwise ACK + REPORT.

Parameters **msg** (*Message*) – current received message

Return type *Message*

Returns list of Message

classmethod get_random_reason()
Return tlv dictionary containing a random reason from pre-defined reason list.

Parameters **msg** – current received message

Return type dict

Returns tlv dictionary with failure reason

classmethod nack_with_reason(*msg*)
Return a NACK message with a tlv part containing the failure reason.

Parameters **msg** (*Message*) – current received message

Return type List[*Message*]

Returns list of Message

6.5 Instrument Control Auxiliary

module instrument_control

synopsis provide a simple interface to control instruments using SCPI protocol.

The functionalities provided in this package may be used directly inside ITF tests using the corresponding auxiliary, but also using a CLI.

Warning:

This auxiliary can only be used with the cc_visa or cc_tcp_ip connector.
It is not intended to be used with a proxy connector.
One instrument is bound to one auxiliary even if the instrument has multiple channels.

pykiso.lib.auxiliaries.instrument_control_auxiliary.instrument_control_auxiliary	Instrument Control Auxiliary
pykiso.lib.auxiliaries.instrument_control_auxiliary.instrument_control_cli	Instrument Control CLI
pykiso.lib.auxiliaries.instrument_control_auxiliary.lib_scpi_commands	Library of SCPI commands
pykiso.lib.auxiliaries.instrument_control_auxiliary.lib_instruments	Library of instruments communicating via VISA

6.5.1 Instrument Control Auxiliary

module instrument_control_auxiliary

synopsis Auxiliary used to communicate via a VISA connector using the SCPI protocol.

class pykiso.lib.auxiliaries.instrument_control_auxiliary.instrument_control_auxiliary.**InstrumentContro**

Auxiliary used to communicate via a VISA connector using the SCPI protocol.

Constructor.

Parameters

- **com** (*CChannel*) – VISAChannel that supports VISA communication
- **instrument** – name of the instrument currently in use (will be used to adapt the SCPI commands)
- **write_termination** – write termination character
- **output_channel** (Optional[int]) – output channel to use on the instrument currently in use (if more than one)

handle_query(*query_command*)

Send a query request to the instrument. Uses the 'query' method of the channel if available, uses 'cc_send' and 'cc_receive' otherwise.

Parameters **query_command** (str) – query command to send

Return type str

Returns Response message, None if the request expired with a timeout.

handle_read()

Handle read command by calling associated connector cc_receive.

Return type str

Returns received response from instrument otherwise empty string

handle_write(*write_command*, *validation=None*)

Send a write request to the instrument and then returns if the value was successfully written. A query is sent immediately after the writing and the answer is compared to the expected one.

Parameters

- **write_command** (str) – write command to send
- **validation** (Optional[Tuple[str, Union[str, List[str]]]]) – tuple of the form (validation command (str), expected output (str or list of str))

Return type str

Returns status message depending on the command validation: SUCCESS, FAILURE or NO_VALIDATION

query(*query_command*)

Send a query request to the instrument. Uses the 'query' method of the channel if available, uses 'cc_send' and 'cc_receive' otherwise.

Parameters **query_command** (str) – query command to send

Return type Union[bytes, str]

Returns Response message, None if the request expired with a timeout.

read()

Send a read request to the instrument.

Return type Union[str, bool]

Returns received response from instrument otherwise empty string

resume()
Not used.

Return type None

run_command(*cmd_message*, *cmd_data=None*, *blocking=True*, *timeout_in_s=0*)
Put a command to execute in thread queue in.

Parameters

- **cmd_message** (str) – command request to the auxiliary
- **cmd_data** (Optional[str]) – payload to send over associated connector.
- **blocking** (bool) – If you want the command request to be blocking or not
- **timeout_in_s** (int) – Number of time (in s) you want to wait for an answer

Return type Union[str, bool]

Returns received response from connected instrument or False if an error occurred.

stop()
Stop the auxiliary thread

Return type None

suspend()
Not used.

Return type None

write(*write_command*, *validation=None*)
Send a write request to the instrument.

Parameters

- **write_command** (str) – command to send
- **validation** (Optional[Tuple[str, Union[str, List[str]]]]) – contain validation criteria apply on the response

Return type str

6.5.2 Instrument Control CLI

module instrument_control_cli

synopsis Command Line Interface used to communicate with an instrument using the SCPI protocol.

class pykiso.lib.auxiliaries.instrument_control_auxiliary.instrument_control_cli.**ExitCode**(*value*)
List of possible exit codes

class pykiso.lib.auxiliaries.instrument_control_auxiliary.instrument_control_cli.**Interface**(*value*)
List of available interfaces

pykiso.lib.auxiliaries.instrument_control_auxiliary.instrument_control_cli.**initialize_logging**(*log_level*)
Initialize the logging by setting the general log level

Parameters **log_level** (str) – any of DEBUG, INFO, WARNING, ERROR

Return type getLogger

Returns configured Logger

pykiso.lib.auxiliaries.instrument_control_auxiliary.instrument_control_cli.**parse_user_command**(*user_cmd*)
Parses the command from user input in interactive mode

Parameters **user_cmd** (str) – command provided by the user in interactive mode

Return type dict

Returns a single-item dictionary containing the parsed command as key the the corresponding payload as value

pykiso.lib.auxiliaries.instrument_control_auxiliary.instrument_control_cli.**perform_actions**(*instr_aux*, *actions*)
Performs the desired actions from the CLI arguments

Parameters

- **instr_aux** (*InstrumentControlAuxiliary*) – instrument on which to perform the actions
- **actions** (dict) – dictionary containing the parsed argument and the corresponding value.

Return type None

pykiso.lib.auxiliaries.instrument_control_auxiliary.instrument_control_cli.**setup_interface**(*interface*, *baud_rate=Non*
ip_address=Nor
port=None, *protocol=None*, *name=None*)
Set up the instrument auxiliary with the appropriate interface settings. The ip address must be provided in the case of TCPIP interfaces, as must the serial port for VISA_SERIAL interface. The baud rate and the output channel to use are optional.

Parameters

- **interface** (str) – interface to use
- **baud_rate** (Optional[int]) – baud rate to use
- **ip_address** (Optional[str]) – ip address of the remote instrument (used for remote control only)
- **port** (Optional[int]) – the port of the device to connect to. This is either a serial port for a VISA_SERIAL interface or an IP port in case of an TCPIP interfaces.
- **protocol** (Optional[str]) – The protocol to use for VISA_TCPIP interfaces.
- **name** (Optional[str]) – instrument name used to adapt the SCPI commands to be sent to the instrument

Return type *InstrumentControlAuxiliary*

Returns The created instrument auxiliary.

6.5.3 Library of SCPI commands

module lib_scpi_commands

synopsis Library of helper functions used to send requests to instruments with SCPI protocol. This library can be used with any VISA instance having a write and a query method.

class pykiso.lib.auxiliaries.instrument_control_auxiliary.lib_scpi_commands.**LibSCPI**(*visa_object*, *instrument=''*)

Class containing common SCPI commands for write and query requests.

Constructor.

Parameters

- **visa_object** – any visa object having a write and a query method
- **instrument** (str) – name of the instrument in use. If registered, the commands adapted to this instrument's capabilities are used instead of the default ones.

disable_output()
Disable output on the currently selected output channel of an instrument.

Return type str

Returns the writing operation's status code

enable_output()
Enable output on the currently selected output channel of an instrument.

Return type str

Returns the writing operation's status code

get_all_errors()
Get all errors of an instrument.

return: list of off errors

get_command(*cmd_tag*, *cmd_type*, *cmd_validation=None*)
Return the pre-defined command.

Parameters

- **cmd_tag** (str) – command tag corresponding to the command to execute
- **cmd_type** (str) – either 'write' or 'query'
- **cmd_validation** (Optional[tuple]) – expected output after validation (only used in write commands)

Return type Tuple

Returns the associated command plus a tuple containing the associated query and the expected response (if cmd_validation is not none) otherwise None

get_current_limit_high()
Returns the current upper limit (in V) of an instrument.

Return type str

Returns the query's response message

get_current_limit_low()
Returns the current lower limit (in V) of an instrument.

Return type str

Returns the query's response message

get_identification()
Get the identification information of an instrument.

Returns the instrument's identification information

get_nominal_current()
Query the nominal current of an instrument on the selected channel (in A).

Return type str

Returns the nominal current

get_nominal_power()
Query the nominal power of an instrument on the selected channel (in W).

Return type str

Returns the nominal power

get_nominal_voltage()
Query the nominal voltage of an instrument on the selected channel (in V).

Return type str

Returns the nominal voltage

get_output_channel()
Get the currently selected output channel of an instrument.

Return type str

Returns the currently selected output channel

get_output_state()
Get the output status (ON or OFF, enabled or disabled) of the currently selected channel of an instrument.

Return type str

Returns the output state (ON or OFF)

get_power_limit_high()
Returns the power upper limit (in W) of an instrument.

Return type str

Returns the query's response message

get_remote_control_state()
Get the remote control mode (ON or OFF) of an instrument.

Returns the remote control state

get_status_byte()
Get the status byte of an instrument.

Returns the instrument's status byte

get_target_current()
Get the desired output current (in A) of an instrument.

Return type str

Returns the target current

get_target_power()
Get the desired output power (in W) of an instrument.

Return type str

Returns the target power

get_target_voltage()
Get the desired output voltage (in V) of an instrument.

Return type str

Returns the target voltage

get_voltage_limit_high()
Returns the voltage upper limit (in V) of an instrument.

Return type str

Returns the query's response message

get_voltage_limit_low()
Returns the voltage lower limit (in V) of an instrument.

Return type str

Returns the query's response message

measure_current()
Return the measured output current of an instrument (in A).

Return type str

Returns the measured current

measure_power()
Return the measured output power of an instrument (in W).

Return type str

Returns the measured power

measure_voltage()
Return the measured output voltage of an instrument (in V).

Return type str

Returns the measured voltage

reset()
Reset an instrument.

Returns NO_VALIDATION status code

self_test()
Performs a self-test of an instrument.

Returns the query's response message

set_current_limit_high(*limit_value*)
Set the current upper limit (in A) of an instrument.

Parameters **limit_value** (float) – limit value to be set on the instrument

Return type str

Returns the writing operation's status code

set_current_limit_low(*limit_value*)
Set the current lower limit (in A) of an instrument.

Parameters **limit_value** (float) – limit value to be set on the instrument

Return type str

Returns the writing operation's status code

set_output_channel(*channel*)
Set the output channel of an instrument.

Parameters **channel** (int) – the output channel to select on the instrument

Return type str

Returns the writing operation's status code

set_power_limit_high(*limit_value*)
Set the power upper limit (in W) of an instrument.

Parameters **limit_value** (float) – limit value to be set on the instrument

Return type str

Returns the writing operation's status code

set_remote_control_off()
Disable the remote control of an instrument. The instrument will respond to query and read commands only.

Returns the writing operation's status code

set_remote_control_on()
Enables the remote control of an instrument. The instrument will respond to all SCPI commands.

Returns the writing operation's status code

set_target_current(*value*)
Set the desired output current (in A) of an instrument.

Parameters **value** (float) – value to be set on the instrument

Return type str

Returns the writing operation's status code

set_target_power(*value*)
Set the desired output power (in W) of an instrument.

Parameters **value** (float) – value to be set on the instrument

Return type str

Returns the writing operation's status code

set_target_voltage(*value*)
Set the desired output voltage (in V) of an instrument.

Parameters **value** (float) – value to be set on the instrument

Return type str

Returns the writing operation's status code

set_voltage_limit_high(*limit_value*)
Set the voltage upper limit (in V) of an instrument.

Parameters **limit_value** (float) – limit value to be set on the instrument

Return type str

Returns the writing operation's status code

set_voltage_limit_low(*limit_value*)
Set the voltage lower limit (in V) of an instrument.

Parameters **limit_value** (float) – limit value to be set on the instrument

Return type str

Returns the writing operation's status code

6.5.4 Library of instruments communicating via VISA

module lib_instruments

synopsis Dictionaries containing the appropriate SCPI commands for some instruments.

6.6 Device Under Test Auxiliary

module DUTAuxiliary

synopsis The Device Under Test auxiliary allow to flash and run test on the target using the connector provided.

class pykiso.lib.auxiliaries.dut_auxiliary.**DUTAuxiliary**(*name=None*, *com=None*, *flash=None*, ***kwargs*)
Device Under Test(DUT) auxiliary implementation.

Constructor.

Parameters

- **name** (Optional[str]) – Alias of the auxiliary instance
- **com** (Optional[*CChannel*]) – Communication connector
- **flash** (Optional[*Flasher*]) – flash connector

resume()
Resume DutAuxiliary's run.

Set _is_suspend flag to False in order to re-activate flash sequence in case of e.g. a futur abort command.

Return type None

suspend()
Suspend DutAuxiliary's run.

Set _is_suspend flag to True to avoid a re-flash sequence.

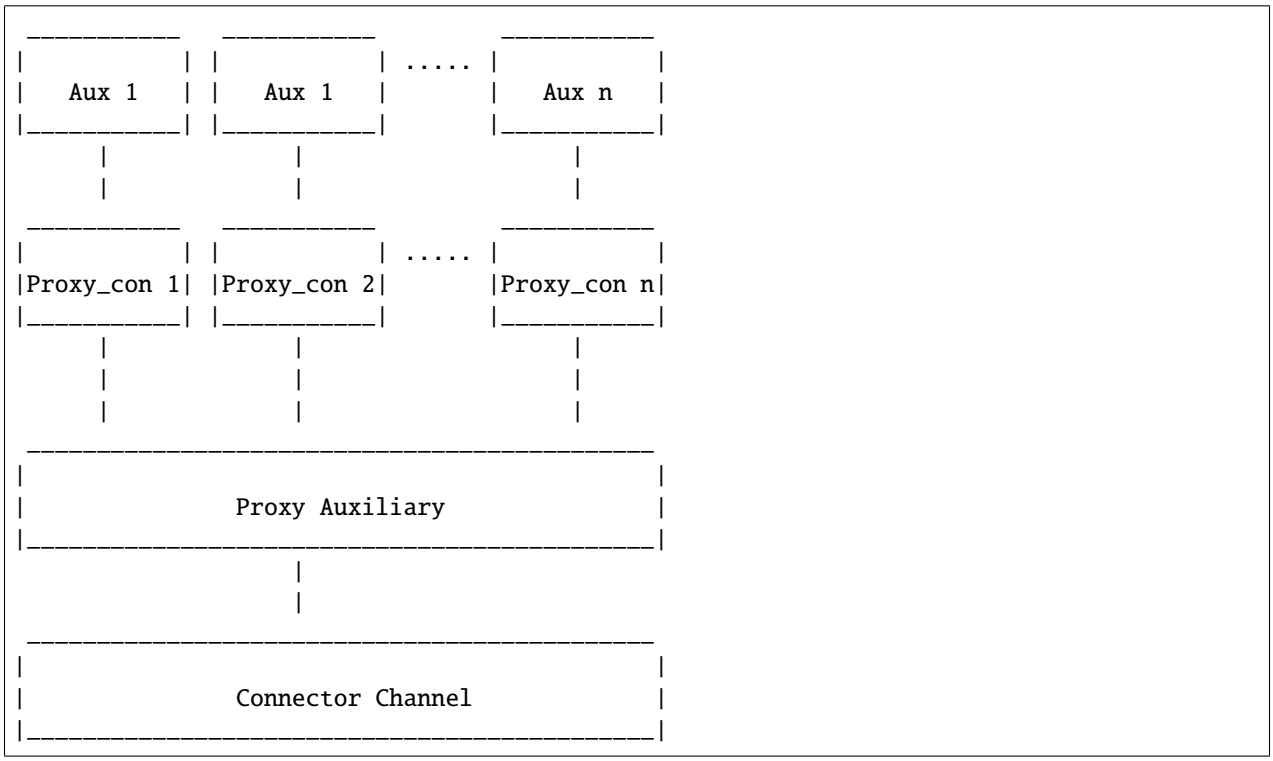
Return type None

6.7 Proxy Auxiliary

module proxy_auxiliary

synopsis auxiliary use to connect multiple auxiliaries on a unique connector.

This auxiliary simply spread all commands and received messages to all connected auxiliaries. This auxiliary is only usable through proxy connector.



class pykiso.lib.auxiliaries.proxy_auxiliary.**ProxyAuxiliary**(*com*, *aux_list*, *activate_trace=False*, *trace_dir=None*, *trace_name=None*, ***kwargs*)

Proxy auxiliary for multi auxiliaries communication handling.

Initialize attributes.

Parameters

- **com** (*CChannel*) – Communication connector
- **aux_list** (List[str]) – list of auxiliary's alias

get_proxy_con(*aux_list*)

Retrieve all connector associated to all given existing Auxiliaries.

If auxiliary alias exists but auxiliary instance was not created yet, create it immediately using ConfigRegistry _aux_cache.

Parameters **aux_list** (List[str]) – list of auxiliary's alias

Return type Tuple[*AuxiliaryInterface*]

Returns tuple containing all connectors associated to all given auxiliaries

run()
Run function of the auxiliary thread

Return type None

CHAPTER

SEVEN

ITF AND ROBOT FRAMEWORK

Integration Test Framework auxiliary<->connector mechanism is usable with Robot framework. In order to achieve it, extra plugins have been developed :

- RobotLoader : handle the import magic mechanism
- RobotComAux : keyword declaration for existing CommunicationAuxiliary

Note: See Robot framework regarding details about Robot keywords, cli...

7.1 How to

To bind ITF with Robot framework, the RobotLoader library has to be used in order to correctly create all auxiliaries and connectors (using the "usual" yaml configuration style). This step is mandatory, and could be done using the "Library" keyword and RobotLoader install/uninstall function. For example, inside a test suite using "Suite Setup" and "Suite Teardown":

```
*** Settings ***
Documentation   How to handle auxiliaries and connectors creation using Robot framework

Library    pykiso.lib.robot_framework.loader.RobotLoader    robot_com_aux.yaml    WITH␣
↪NAME    Loader

Suite Setup       Loader.install
Suite Teardown    Loader.uninstall
```

7.2 Ready to Use Auxiliaries

7.2.1 Communication Auxiliary

This plugin only contains two keywords "Send message" and "Receive message". The first one simply sends raw bytes using the associated connector and the second one returns one received message (raw form).

See below a complete example of the Robot Communication Auxiliary plugin:

```
*** Settings ***
Documentation   Robot framework Demo for communication auxiliary implementation
```

(continues on next page)

(continued from previous page)

```
Library    pykiso.lib.robot_framework.communication_auxiliary.CommunicationAuxiliary  ␣
↪WITH NAME    ComAux

*** Keywords ***

send raw message
    [Arguments]  ${raw_msg}  ${aux}
    ${is_executed}=    ComAux.Send message    ${raw_msg}    ${aux}
    [return]  ${is_executed}

get raw message
    [Arguments]  ${aux}  ${blocking}  ${timeout}
    ${msg}  ${source}=    ComAux.Receive message    ${aux}    ${blocking}    ${timeout}
    [return]  ${msg}  ${source}

*** Test Cases ***

Test send raw bytes using keywords
    [Documentation]    Simply send raw bytes over configured channel
    ...                using defined keywords

    ${state}  send raw message    \x01\x02\x03    aux1

    Log    ${state}

    Should Be Equal    ${state}    ${TRUE}

    ${msg}  ${source}  get raw message    aux1    ${TRUE}    0.5

    Log    ${msg}

Test send raw bytes
    [Documentation]    Simply send raw bytes over configured channel
    ...                using communication auxiliary methods directly

    ${state} =  Send message    \x04\x05\x06    aux2

    Log    ${state}

    Should Be Equal    ${state}    ${TRUE}

    ${msg}  ${source} =  Receive message    aux2    ${FALSE}    0.5

    Log    ${msg}
```

7.2.2 DUT Auxiliary

This plugin can be used to control the ITF TestApp on the DUT.

See below an example of the Robot DUT Auxiliary plugin:

```
*** Settings ***
Documentation   Test demo with RobotFramework and ITF TestApp

Library    pykiso.lib.robot_framework.dut_auxiliary.DUTAuxiliary    WITH NAME    DutAux

Suite Setup       Setup Aux

*** Keywords ***
Setup Aux
    @{auxiliaries} =    Create List  aux1  aux2
    Set Suite Variable  @{suite_auxiliaries}  @{auxiliaries}

*** Variables ***

*** Test Cases ***

Test TEST_SUITE_SETUP
    [Documentation]   Setup test suite on DUT
    Test App    TEST_SUITE_SETUP  1  1  ${suite_auxiliaries}

Test TEST_SECTION_RUN
    [Documentation]   Run test section on DUT
    Test App    TEST_SECTION_RUN  1  1  ${suite_auxiliaries}

Test TEST_CASE_SETUP
    [Documentation]   Setup test case on DUT
    Test App    TEST_CASE_SETUP  1  1  ${suite_auxiliaries}

Test TEST_CASE_RUN
    [Documentation]   Run test case on DUT
    Test App    TEST_CASE_RUN  1  1  ${suite_auxiliaries}

Test TEST_CASE_TEARDOWN
    [Documentation]   Teardown test case on DUT
    Test App    TEST_CASE_TEARDOWN  1  1  ${suite_auxiliaries}

Test TEST_SUITE_TEARDOWN
    [Documentation]   Teardown test suite on DUT
    Test App    TEST_SUITE_TEARDOWN  1  1  ${suite_auxiliaries}
```

7.2.3 Proxy Auxiliary

This robot plugin only contains two keywords : Suspend and Resume.

See below example :

```
*** Settings ***
Documentation   Robot framework Demo for proxy auxiliary implementation

Library    pykiso.lib.robot_framework.proxy_auxiliary.ProxyAuxiliary    WITH NAME    ␣
↪ProxyAux


*** Test Cases ***

Stop auxiliary run
    [Documentation]    Simply stop the current running auxiliary

    Suspend    ProxyAux

Resume auxiliary run
    [Documentation]    Simply resume the current running auxiliary

    Resume    ProxyAux
```

7.2.4 Instrument Control Auxiliary

As the "ITF" instrument control auxiliary, the robot version integrate exactly the same user's interface.

Note: All return types between "ITF" and "Robot" auxiliary's version stay identical!

Please find below a complete correlation table:

ITF method	robot equivalent	Parameter 1	Parameter 2	Parameter 3
write	Write	command	aux alias	validation
read	Read	aux alias		
query	Query	command	aux alias	
get_identification	Get identification	aux alias		
get_status_byte	Get status byte	aux alias		
get_all_errors	Get all errors	aux alias		
reset	Reset	aux alias		
self_test	Self test	aux alias		
get_remote_control_state	Get remote control state	aux alias		
set_remote_control_on	Set remote control on	aux alias		
set_remote_control_off	Set remote control off	aux alias		
get_output_channel	Get output channel	aux alias		
set_output_channel	Set output channel	channel	aux alias	
get_output_state	Get output state	aux alias		
enable_output	Enable output	aux alias		
disable_output	Disable output	aux alias		

continues on next page

Table 1 – continued from previous page

ITF method	robot equivalent	Parameter 1	Parameter 2	Parameter 3
get_nominal_voltage	Get nominal voltage	aux alias		
get_nominal_current	Get nominal current	aux alias		
get_nominal_power	Get nominal power	aux alias		
measure_voltage	Measure voltage	aux alias		
measure_current	Measure current	aux alias		
measure_power	Measure power	aux alias		
get_target_voltage	Get target voltage	aux alias		
get_target_current	Get target current	aux alias		
get_target_power	Get target power	aux alias		
set_target_voltage	Set target voltage	voltage	aux alias	
set_target_current	Set target current	current	aux alias	
set_target_power	Set target power	power	aux alias	
get_voltage_limit_low	Get voltage limit low	aux alias		
get_voltage_limit_high	Get voltage limit high	aux alias		
get_current_limit_low	Get current limit low	aux alias		
get_current_limit_high	Get current limit high	aux alias		
get_power_limit_high	Get power limit high	aux alias		
set_voltage_limit_low	Set voltage limit low	voltage limit	aux alias	
set_voltage_limit_high	Set voltage limit high	voltage limit	aux alias	
set_current_limit_low	Set current limit low	current limit	aux alias	
set_current_limit_high	Set current limit high	current limit	aux alias	
set_power_limit_high	Set power limit high	power limit	aux alias	

To run the available example:

```
cd examples
robot robot_test_suite/test_instrument
```

Note: A script demo with all available keywords is under examples/robot_test_suite/test_instrument and yaml see robot_inst_aux.yaml!

7.3 Library Documentation

7.3.1 Dynamic Loader plugin

module loader

synopsis implementation of existing magic import mechanism from ITF for Robot framework usage.

class pykiso.lib.robot_framework.loader.**RobotLoader**(*config_file*)

Robot framework plugin for ITF magic import mechanism.

Initialize attributes.

:param config_file : yaml configuration file path

install()

Provide, create and import auxiliaires/connectors present within yaml configuration file.

Raises re-raise the caught exception (Exception level)

Return type None

uninstall()
Uninstall all created instances of auxiliaires/connectors.

Raises re-raise the caught exception (Exception level)

Return type None

7.3.2 Auxiliary interface

module aux_interface

synopsis Simply stored common methods for auxiliary's when ITF is used with Robot framework.

class pykiso.lib.robot_framework.aux_interface.**RobotAuxInterface**(*aux_type*)
Common interface for all Robot auxiliary.

Initialize attributes.

Parameters **aux_type** (*AuxiliaryInterface*) – auxiliary's class

7.3.3 Communication auxiliary plugin

module communication_auxiliary

synopsis implementation of existing CommunicationAuxiliary for Robot framework usage.

class pykiso.lib.robot_framework.communication_auxiliary.**CommunicationAuxiliary**
Robot framework plugin for CommunicationAuxiliary.

Initialize attributes.

receive_message(*aux_alias*, *blocking=True*, *timeout_in_s=None*)
Return a raw received message from the queue.

Parameters

- **aux_alias** (str) – auxiliary's alias
- **blocking** (bool) – wait for message till timeout elapses?
- **timeout_in_s** (Optional[float]) – maximum time in second to wait for a response

Return type Union[list, Tuple[list, int]]

Returns raw message and source (return type could be different depending on the associated channel)

send_message(*raw_msg*, *aux_alias*)
Send a raw message via the communication channel.

Parameters

- **aux_alias** (str) – auxiliary's alias
- **raw_msg** (bytes) – message to send

Return type bool

Returns state representing the send message command completion

7.3.4 Testapp binding

module dut_auxiliary

synopsis implementation of existing DUTAuxiliary for Robot framework usage.

class pykiso.lib.robot_framework.dut_auxiliary.**DUTAuxiliary**
Robot library to control the TestApp on the DUT

Initialize attributes.

test_app_run(*command_type*, *test_suite_id*, *test_case_id*, *aux_list*, *timeout_cmd=5*, *timeout_resp=5*)
Handle default communication mechanism between test manager and device under test.

Parameters

- **command_type** (str) – message command sub-type (TEST_SECTION_SETUP , TEST_SECTION_RUN, ...)
- **test_suite_id** (int) – select test suite id on dut
- **test_case_id** (int) – select test case id on dut
- **aux_list** (List[str]) – List of selected auxiliary
- **timeout_cmd** (int) – timeout in seconds for auxiliary run_command
- **timeout_resp** (int) – timeout in seconds for auxiliary wait_and_get_report

Return type None

class pykiso.lib.robot_framework.dut_auxiliary.**TestEntity**(*test_suite_id*, *test_case_id*, *aux_list*)
Dummy Class to use handle_basic_interaction from test_message_handler.

Initialize generic test-case

Parameters

- **test_suite_id** (int) – test suite identification number
- **test_case_id** (int) – test case identification number
- **aux_list** (List[*AuxiliaryInterface*]) – list of used aux_list

cleanup_and_skip(*aux*, *info_to_print*)
Cleanup auxiliary and log reasons.

Parameters

- **aux** (*AuxiliaryInterface*) – corresponding auxiliary to abort
- **info_to_print** (str) – A message you want to print while cleaning up the test

7.3.5 Proxy auxiliary plugin

module proxy_auxiliary

synopsis implementation of existing ProxyAuxiliary for Robot framework usage.

class pykiso.lib.robot_framework.proxy_auxiliary.**ProxyAuxiliary**
Robot framework plugin for ProxyAuxiliary.

Initialize attributes.

resume(*aux_alias*)
Resume given auxiliary's run.

Parameters **aux_alias** (str) – auxiliary's alias

Return type None

suspend(*aux_alias*)
Suspend given auxiliary's run.

Parameters **aux_alias** (str) – auxiliary's alias

Return type None

7.3.6 Instrument control auxiliary plugin

module instrument_control_auxiliary

synopsis implementation of existing InstrumentControlAuxiliary for Robot framework usage.

class pykiso.lib.robot_framework.instrument_control_auxiliary.**InstrumentControlAuxiliary**
Robot framework plugin for InstrumentControlAuxiliary.

Initialize attributes.

disable_output(*aux_alias*)
Disable output on the currently selected output channel of an instrument.

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the writing operation's status code

enable_output(*aux_alias*)
Enable output on the currently selected output channel of an instrument.

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the writing operation's status code

get_all_errors(*aux_alias*)
Get all errors of an instrument.

Parameters **aux_alias** (str) – auxiliary's alias

return: list of off errors

Return type str

get_current_limit_high(*aux_alias*)
Returns the current upper limit (in V) of an instrument.

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the query's response message

get_current_limit_low(*aux_alias*)
Returns the current lower limit (in V) of an instrument.

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the query's response message

get_identification(*aux_alias*)
Get the identification information of an instrument.

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the instrument's identification information

get_nominal_current(*aux_alias*)
Query the nominal current of an instrument on the selected channel (in A)

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the nominal current

get_nominal_power(*aux_alias*)
Query the nominal power of an instrument on the selected channel (in W).

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the nominal power

get_nominal_voltage(*aux_alias*)
Query the nominal voltage of an instrument on the selected channel (in V).

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the nominal voltage

get_output_channel(*aux_alias*)
Get the currently selected output channel of an instrument.

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the currently selected output channel

get_output_state(*aux_alias*)
Get the output status (ON or OFF, enabled or disabled) of the currently selected channel of an instrument.

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the output state (ON or OFF)

get_power_limit_high(*aux_alias*)
Returns the power upper limit (in W) of an instrument.

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the query's response message

get_remote_control_state(*aux_alias*)
Get the remote control mode (ON or OFF) of an instrument.

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the remote control state

get_status_byte(*aux_alias*)
Get the status byte of an instrument.

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the intrument's status byte

get_target_current(*aux_alias*)
Get the desired output current (in A) of an instrument.

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the target current

get_target_power(*aux_alias*)
Get the desired output power (in W) of an instrument.

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the target power

get_target_voltage(*aux_alias*)
Get the desired output voltage (in V) of an instrument.

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the target voltage

get_voltage_limit_high(*aux_alias*)
Returns the voltage upper limit (in V) of an instrument.

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the query's response message

get_voltage_limit_low(*aux_alias*)
Returns the voltage lower limit (in V) of an instrument.

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the query's response message

measure_current(*aux_alias*)
Return the measured output current of an instrument (in A).

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the measured current

measure_power(*aux_alias*)
Return the measured output power of an instrument (in W).

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the measured power

measure_voltage(*aux_alias*)
Return the measured output voltage of an instrument (in V).

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the measured voltage

query(*query_command*, *aux_alias*)
Send a query request to the instrument.

Parameters

- **query_command** (str) – query command to send
- **aux_alias** (str) – auxiliary's alias

Return type str

Returns Response message, None if the request expired with a timeout.

read(*aux_alias*)
Send a read request to the instrument.

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns Response message, None if the request expired with a timeout.

reset(*aux_alias*)
Reset an instrument.

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns NO_VALIDATION status code

self_test(*aux_alias*)
Performs a self-test of an instrument.

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the query's response message

set_current_limit_high(*limit_value*, *aux_alias*)
Set the current upper limit (in A) of an instrument.

Parameters

- **limit_value** (float) – limit value to be set on the instrument
- **aux_alias** (str) – auxiliary's alias

Return type str

Returns the writing operation's status code

set_current_limit_low(*limit_value*, *aux_alias*)
Set the current lower limit (in A) of an instrument.

Parameters

- **limit_value** (float) – limit value to be set on the instrument
- **aux_alias** (str) – auxiliary's alias

Return type str

Returns the writing operation's status code

set_output_channel(*channel*, *aux_alias*)
Set the output channel of an instrument.

Parameters

- **channel** (int) – the output channel to select on the instrument
- **aux_alias** (str) – auxiliary's alias

Return type str

Returns the writing operation's status code

set_power_limit_high(*limit_value*, *aux_alias*)
Set the power upper limit (in W) of an instrument.

Parameters

- **limit_value** (float) – limit value to be set on the instrument
- **aux_alias** (str) – auxiliary's alias

Return type str

Returns the writing operation's status code

set_remote_control_off(*aux_alias*)
Disable the remote control of an instrument. The instrument will respond to query and read commands only.

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the writing operation's status code

set_remote_control_on(*aux_alias*)
Enables the remote control of an instrument. The instrument will respond to all SCPI commands.

Parameters **aux_alias** (str) – auxiliary's alias

Return type str

Returns the writing operation's status code

set_target_current(*value*, *aux_alias*)
Set the desired output current (in A) of an instrument.

Parameters

- **value** (float) – value to be set on the instrument
- **aux_alias** (str) – auxiliary's alias

Return type str

Returns the writing operation's status code

set_target_power(*value*, *aux_alias*)
Set the desired output power (in W) of an instrument.

Parameters

- **value** (float) – value to be set on the instrument
- **aux_alias** (str) – auxiliary's alias

Return type str

Returns the writing operation's status code

set_target_voltage(*value*, *aux_alias*)
Set the desired output voltage (in V) of an instrument.

Parameters

- **value** (float) – value to be set on the instrument
- **aux_alias** (str) – auxiliary's alias

Return type str

Returns the writing operation's status code

set_voltage_limit_high(*limit_value*, *aux_alias*)
Set the voltage upper limit (in V) of an instrument.

Parameters

- **limit_value** (float) – limit value to be set on the instrument
- **aux_alias** (str) – auxiliary's alias

Return type str

Returns the writing operation's status code

set_voltage_limit_low(*limit_value*, *aux_alias*)
Set the voltage lower limit (in V) of an instrument.

Parameters

- **limit_value** (float) – limit value to be set on the instrument
- **aux_alias** (str) – auxiliary's alias

Return type str

Returns the writing operation's status code

write(*write_command*, *aux_alias*, *validation=None*)
Send a write request to the instrument and then returns if the value was successfully written. A query is sent immediately after the writing and the answer is compared to the expected one.

Parameters

- **write_command** (str) – write command to send
- **aux_alias** (str) – auxiliary's alias
- **validation** (Optional[tuple]) – tuple of the form (validation command (str), expected output (str))

Return type str

Returns status message depending on the command validation: SUCCESS, FAILURE or NO_VALIDATION

CHAPTER

EIGHT

API DOCUMENTATION

8.1 Test Cases

8.1.1 Generic Test

module test_case

synopsis Basic extensible implementation of a TestCase.

Note: TODO later on will inherit from a metaclass to get the id parameters

class pykiso.test_coordinator.test_case.**BasicTest**(*test_suite_id*, *test_case_id*, *aux_list*, *setup_timeout*, *run_timeout*, *teardown_timeout*, *test_ids*, *args*, *kwargs*)

Base for test-cases.

Initialize generic test-case.

Parameters

- **test_suite_id** (int) – test suite identification number
- **test_case_id** (int) – test case identification number
- **aux_list** (Optional[List[*AuxiliaryInterface*]]) – list of used auxiliaries
- **setup_timeout** (Optional[int]) – maximum time (in seconds) used to wait for a report during setup execution
- **run_timeout** (Optional[int]) – maximum time (in seconds) used to wait for a report during test_run execution
- **teardown_timeout** (Optional[int]) – the maximum time (in seconds) used to wait for a report during teardown execution
- **test_ids** (Optional[dict]) – jama references to get the coverage eg: {"Component1": ["Req1", "Req2"], "Component2": ["Req3"]}

cleanup_and_skip(*aux*, *info_to_print*)

Cleanup auxiliary and log reasons.

Parameters

- **aux** (*AuxiliaryInterface*) – corresponding auxiliary to abort
- **info_to_print** (str) – A message you want to print while cleaning up the test

Return type None

msg_handler
alias of *pykiso.test_coordinator.test_message_handler.TestCaseMsgHandler*

setUp()
Hook method for constructing the test fixture.

Return type None

classmethod setUpClass()
A class method called before tests in an individual class are run.

This implementation is only mandatory to enable logging in junit report. The logging configuration has to be call inside test runner run, otherwise stdout is never caught.

Return type None

tearDown()
Hook method for deconstructing the test fixture after testing it.

Return type None

test_run()
Hook method from unittest in order to execute test case.

Return type None

pykiso.test_coordinator.test_case.**define_test_parameters**(*suite_id=0*, *case_id=0*, *aux_list=None*, *setup_timeout=None*, *run_timeout=None*, *teardown_timeout=None*, *test_ids=None*)
Decorator to fill out test parameters of the BasicTest automatically.

8.2 Connectors

8.2.1 Interface Definition for Connectors, CChannels and Flasher

module connector

synopsis Interface for a channel

class pykiso.connector.**CChannel**(***kwargs*)
Abstract class for coordination channel.

constructor

abstract _cc_close()
Close the channel.

abstract _cc_open()
Open the channel.

abstract _cc_receive(*timeout*, *raw=False*)
How to receive something from the channel.

Parameters

- **timeout** (float) – Time to wait in second for a message to be received
- **raw** (bool) – send raw message without further work (default: False)

Return type Union[*Message*, bytes, str]

Returns message.Message() - If one received / None - If not

abstract _cc_send(*msg*, *raw=False*)
Sends the message on the channel.

Parameters

- **msg** (Union[*Message*, bytes, str]) – Message to send out
- **raw** (bool) – send raw message without further work (default: False)

Return type None

cc_receive(*timeout=0.1*, *raw=False*)
Read a thread-safe message on the channel and send an acknowledgement.

Parameters

- **timeout** (float) – time in second to wait for reading a message
- **raw** (bool) – should the message be returned raw or should it be interpreted as a pykiso.Message?

Returns Message if successful, None else

Raises **ConnectionRefusedError** – when lock acquire failed

cc_send(*msg*, *raw=False*, ***kwargs*)
Send a thread-safe message on the channel and wait for an acknowledgement.

Parameters **msg** (Union[*Message*, bytes, str]) – message to send

Raises **ConnectionRefusedError** – when lock acquire failed

close()
Close a thread-safe channel.

Return type None

open()
Open a thread-safe channel.

Raises **ConnectionRefusedError** – when lock acquire failed

Return type None

class pykiso.connector.**Connector**(*name=None*)
Abstract interface for all connectors to inherit from.

Defines hooks for opening and closing the connector and also defines a contextmanager interface.

Constructor.

Parameters **name** (Optional[str]) – alias for the connector, used for repr and logging.

abstract close()
Close the connector, freeing resources.

abstract open()
Initialise the Connector.

class pykiso.connector.**Flasher**(*binary=None*, ***kwargs*)
Interface for devices that can flash firmware on our targets.

Constructor.

Parameters **binary** (Union[str, Path, None]) – binary firmware file

Raises

- **ValueError** – if binary doesn't exist or is not a file
- **TypeError** – if given binary is None

abstract flash()
Flash firmware on the target.

8.3 Auxiliary Interface

8.3.1 Auxiliary Interface Definition

module auxiliary

synopsis Implementation of basic auxiliary functionality and definition of abstract methods for override

class pykiso.auxiliary.**AuxiliaryInterface**(*name=None*, *is_proxy_capable=False*, *is_pausable=False*, *activate_log=None*)
Defines the Interface of Auxiliaries.

Auxiliaries get configured by the Test Coordinator, get instantiated by the TestCases and in turn use Connectors.

Auxiliary thread initialization.

Parameters

- **name** (Optional[str]) – alias of the auxiliary instance
- **is_proxy_capable** (bool) – notify if the current auxiliary could be (or not) associated to a proxy-auxiliary.
- **is_pausable** (bool) – notify if the current auxiliary could be (or not) paused
- **activate_log** (Optional[List[str]]) – loggers to deactivate

abort_command(*blocking=True*, *timeout_in_s=25*)
Force test to abort.

Parameters

- **blocking** (bool) – If you want the command request to be blocking or not
- **timeout_in_s** (float) – Number of time (in s) you want to wait for an answer

Return type bool

Returns True - Abort was a success / False - if not

create_instance()
Create an auxiliary instance and ensure the communication to it.

Return type bool

Returns message.Message() - Contain received message

delete_instance()
Delete an auxiliary instance and its communication to it.

Return type bool

Returns message.Message() - Contain received message

static initialize_loggers(*loggers*)
Deactivate all external loggers except the specified ones.

Parameters **loggers** (Optional[List[str]]) – list of logger names to keep activated

lock_it(*timeout_in_s*)
Lock to ensure exclusivity.

Parameters **timeout_in_s** (*integer*) – How many second you want to wait for the lock

Return type bool

Returns True - Lock done / False - Lock failed

resume()
Resume current auxiliary's run.

Return type None

run()
Run function of the auxiliary thread.

Return type None

run_command(*cmd_message*, *cmd_data=None*, *blocking=True*, *timeout_in_s=0*)
Send a test request.

Parameters

- **cmd_message** (Union[*Message*, bytes, str]) – command request to the auxiliary
- **cmd_data** (Optional[Any]) – data you would like to populate the command with
- **blocking** (bool) – If you want the command request to be blocking or not
- **timeout_in_s** (int) – Number of time (in s) you want to wait for an answer

Return type bool

Returns True - Successfully sent / False - Failed by sending / None

stop()
Force the thread to stop itself.

Return type None

suspend()
Supend current auxiliary's run.

Return type None

unlock_it()
Unlock exclusivity

Return type None

wait_and_get_report(*blocking=False*, *timeout_in_s=0*)
Wait for the report of the previous sent test request.

Parameters

- **blocking** (bool) – True: wait for timeout to expire, False: return immediately
- **timeout_in_s** (int) – if blocking, wait the defined time in seconds

Return type Union[*Message*, bytes, str]

Returns a message.Message() - Message received / None - nothing received

8.4 Message Protocol

8.4.1 pykiso Control Message Protocol

module message

synopsis Message that will be send though the different agents

class pykiso.message.**Message**(*msg_type=0*, *sub_type=0*, *error_code=0*, *test_suite=0*, *test_case=0*, *tlv_dict=None*)

A message who fit testApp protocol.

The created message is a tlv style message with the following format: TYPE: msg_type | message_token | sub_type | errorCode |

Create a generic message.

Parameters

- **msg_type** (*MessageType*) – Message type
- **sub_type** (*Message<MessageType>Type*) – Message sub-type
- **error_code** (*integer*) – Error value
- **test_section** (*integer*) – Section value
- **test_suite** (*integer*) – Suite value
- **test_case** (*integer*) – Test value
- **tlv_dict** (*dict*) – Dictionary containing tlvs elements in the form {'type':'value', ...}

check_if_ack_message_is_matching(*ack_message*)

Check if the ack message was for this sent message.

Parameters **ack_message** (*Message*) – received acknowledge message

Return type bool

Returns True if message type and token are valid otherwise False

generate_ack_message(*ack_type*)

Generate acknowledgement to send out.

Parameters **ack_type** (int) – ack or nack

Return type Optional[*Message*]

Returns filled acknowledge message otherwise None

classmethod get_crc(*serialized_msg*, *crc_byte_size=2*)

Get the CRC checksum for a bytes message.

Parameters

- **serialized_msg** (bytes) – message used for the crc calculation
- **crc_byte_size** (int) – number of bytes dedicated for the crc

Return type int

Returns CRC checksum

get_message_sub_type()

Return actual message subtype.

Return type int

get_message_tlv_dict()
Return actual message type/length/value dictionary.

Return type dict

get_message_token()
Return actual message token.

Return type int

get_message_type()
Return actual message type.

Return type Union[int, *MessageType*]

classmethod parse_packet(*raw_packet*)
Factory function to create a Message object from raw data.

Parameters **raw_packet** (bytes) – array of a received message

Return type *Message*

Returns itself

serialize()
Serialize message into raw packet.

Format: | msg_type (1b) | msg_token (1b) | sub_type (1b) | error_code (1b) |

test_section (1b) | test_suite (1b) | test_case (1b) | payload_length (1b) |
tlv_type (1b) | tlv_size (1b) | ... | crc_checksum (2b)

Return type bytes

Returns bytes representing the Message object

class pykiso.message.**MessageAckType**(*value*)
List of possible received messages.

class pykiso.message.**MessageCommandType**(*value*)
List of commands allowed.

class pykiso.message.**MessageLogType**(*value*)
List of possible received log messages.

class pykiso.message.**MessageReportType**(*value*)
List of possible received messages.

class pykiso.message.**MessageType**(*value*)
List of messages allowed.

class pykiso.message.**TlvKnownTags**(*value*)
List of known / supported tags.

8.5 Import Magic

8.5.1 Auxiliary Interface Definition

module dynamic_loader

synopsis Import magic that enables aliased auxiliary loading in TestCases

class pykiso.test_setup.dynamic_loader.**DynamicImportLinker**
Public Interface of Import Magic.

initialises the Loaders, Finders and Caches, implements interfaces to install the magic and register the auxiliaries and connectors.

Initialize attributes.

install()
Install the import hooks with the system.

provide_auxiliary(*name*, *module*, *aux_cons=None*, ***config_params*)
Provide a auxiliary.

Parameters

- **name** (str) – the auxiliary alias
- **module** (str) – either 'python-file-path:Class' or 'module:Class' of the class we want to provide
- **aux_cons** – list of connectors this auxiliary has

provide_connector(*name*, *module*, ***config_params*)
Provide a connector.

Parameters

- **name** (str) – the connector alias
- **module** (str) – either 'python-file-path:Class' or 'module:Class' of the class we want to provide

uninstall()
Deregister the import hooks, close all running threads, delete all instances.

8.5.2 Config Registry

module config_registry

synopsis register auxiliaries and connectors to provide them for import.

class pykiso.test_setup.config_registry.**ConfigRegistry**
Register auxiliaries with connectors to provide systemwide import statements.

classmethod delete_aux_con()
deregister the import hooks, close all running threads, delete all instances.

Return type None

classmethod get_all_auxes()
Return all auxiliaires instances and alias

Return type dict

Returns dictionary with alias as keys and instances as values

classmethod get_auxes_alias()
return all created auxiliaries alias.

Return type list

Returns list containing all auxiliaries alias

classmethod get_auxes_by_type(*aux_type*)
Return all auxiliaries who match a specific type.

Parameters **aux_type** (Any) – auxiliary class type (DUTAuxiliary, CommunicationAuxiliary...)

Return type dict

Returns dictionary with alias as keys and instances as values

classmethod register_aux_con(*config*)
Create import hooks. Register auxiliaries and connectors.

Parameters **config** (dict) – dictionary containing yaml configuration content

Return type None

8.6 Test Suites

8.6.1 Test Suite

module test_suite

synopsis Create a generic test-suite based on the connected modules

class pykiso.test_coordinator.test_suite.**BaseTestSuite**(*test_suite_id*, *test_case_id*, *aux_list*, *setup_timeout*, *run_timeout*, *teardown_timeout*, *test_ids*, *args*, *kwargs*)
Initialize generic test-case.

Parameters

- **test_suite_id** (int) – test suite identification number
- **test_case_id** (int) – test case identification number
- **aux_list** (Optional[List[*AuxiliaryInterface*]]) – list of used auxiliaries
- **setup_timeout** (Optional[int]) – maximum time (in seconds) used to wait for a report during setup execution
- **run_timeout** (Optional[int]) – maximum time (in seconds) used to wait for a report during test_run execution
- **teardown_timeout** (Optional[int]) – the maximum time (in seconds) used to wait for a report during teardown execution
- **test_ids** (Optional[dict]) – jama references to get the coverage eg: {"Component1": ["Req1", "Req2"], "Component2 ["Req3"]}

base_function(*current_fixture*, *step_name*, *test_command*, *timeout_resp*)
Base function used for test suite setup and teardown.

Parameters

- **current_fixture** (Callable) – fixture instance teardown or setup
- **step_name** (str) – name of the current step
- **test_command** (*Message*) – A message you want to print while cleaning up the test
- **timeout_resp** (int) – maximum amount of time in seconds to wait for a response

cleanup_and_skip(*aux*, *info_to_print*)
Cleanup auxiliary and log reasons.

Parameters

- **aux** (*AuxiliaryInterface*) – corresponding auxiliary to abort
- **info_to_print** (str) – A message you want to print while cleaning up the test

class pykiso.test_coordinator.test_suite.**BasicTestSuite**(*modules_to_add_dir*, *test_filter_pattern*, *test_suite_id*, *args*, *kwargs*)
Inherit from the unittest framework test-suite but build it for our integration tests.

Initialize our custom unittest-test-suite.

class pykiso.test_coordinator.test_suite.**BasicTestSuiteSetup**(*test_suite_id*, *test_case_id*, *aux_list*, *setup_timeout*, *run_timeout*, *teardown_timeout*, *test_ids*, *args*, *kwargs*)
Inherit from unittest testCase and represent setup fixture.

Initialize generic test-case.

Parameters

- **test_suite_id** (int) – test suite identification number
- **test_case_id** (int) – test case identification number
- **aux_list** (Optional[List[*AuxiliaryInterface*]]) – list of used auxiliaries
- **setup_timeout** (Optional[int]) – maximum time (in seconds) used to wait for a report during setup execution
- **run_timeout** (Optional[int]) – maximum time (in seconds) used to wait for a report during test_run execution
- **teardown_timeout** (Optional[int]) – the maximum time (in seconds) used to wait for a report during teardown execution
- **test_ids** (Optional[dict]) – jama references to get the coverage eg: {"Component1": ["Req1", "Req2"], "Component2": ["Req3"]}

test_suite_setUp()
Test method for constructing the actual test suite.

class pykiso.test_coordinator.test_suite.**BasicTestSuiteTeardown**(*test_suite_id*, *test_case_id*, *aux_list*, *setup_timeout*, *run_timeout*, *teardown_timeout*, *test_ids*, *args*, *kwargs*)
Inherit from unittest testCase and represent teardown fixture.

Initialize generic test-case.

Parameters

- **test_suite_id** (int) – test suite identification number

- **test_case_id** (int) – test case identification number
- **aux_list** (Optional[List[*AuxiliaryInterface*]]) – list of used auxiliaries
- **setup_timeout** (Optional[int]) – maximum time (in seconds) used to wait for a report during setup execution
- **run_timeout** (Optional[int]) – maximum time (in seconds) used to wait for a report during test_run execution
- **teardown_timeout** (Optional[int]) – the maximum time (in seconds) used to wait for a report during teardown execution
- **test_ids** (Optional[dict]) – jama references to get the coverage eg: {"Component1": ["Req1", "Req2"], "Component2": ["Req3"]}

test_suite_tearDown()
Test method for deconstructing the actual test suite after testing it.

class pykiso.test_coordinator.test_suite.**TestSuiteMsgHandler**
Encapsulate all test suite communication mechanisms.

> **Warning:** This class is speculative code and not currently used. it should eventually replace the handle_basic_interaction call in the BaseTestSuite

classmethod setup(*cls*, *test_entity*, *timeout_cmd*, *timeout_resp*)
Handle communication mechanism during test suite setup. Based on actual implementation, this context manager calls handle_basic_interaction.

Parameters

- **test_entity** (Callable) – test instance in use (BaseTestSuite, BasicTest,...)
- **timeout_cmd** (int) – timeout in second apply on auxiliary run_command
- **timeout_resp** (int) – timeout in second apply on auxiliary wait_and_get_report

Returns namedtuple containing current auxiliary, reported message, logging method to use, and pre-defined log message.

classmethod teardown(*cls*, *test_entity*, *timeout_cmd*, *timeout_resp*)
Handle communication mechanism during test suite teardown. Based on actual implementation, this context manager call handle_basic_interaction.

Parameters

- **test_entity** (Callable) – test instance in use (BaseTestSuite, BasicTest,...)
- **timeout_cmd** (int) – timeout in second apply on auxiliary run_command
- **timeout_resp** (int) – timeout in second apply on auxiliary wait_and_get_report

Returns namedtuple containing current auxiliary, reported message, logging method to use, and pre-defined log message.

8.7 Test Execution

8.7.1 Test Execution

module test_execution

synopsis Execute a test environment based on the supplied configuration.

Note:

1. Glob a list of test-suite folders
2. Generate a list of test-suites with a list of test-cases
3. Loop per suite
4. Gather result

---

class pykiso.test_coordinator.test_execution.**ExitCode**(*value*)
List of possible exit codes

pykiso.test_coordinator.test_execution.**create_test_suite**(*test_suite_dict*)
create a test suite based on the config dict

Parameters **test_suite_dict** (Dict) – dict created from config with keys 'suite_dir', 'test_filter_pattern', 'test_suite_id'

Return type *BasicTestSuite*

pykiso.test_coordinator.test_execution.**execute**(*config*, *report_type='text'*)
create test environment base on config

Parameters

- **config** (Dict) – dict from converted YAML config file
- **report_type** (str) – str to set the type of report wanted, i.e. test or junit

pykiso.test_coordinator.test_execution.**failure_and_error_handling**(*result*)
provide necessary information to Jenkins if an error occur during tests execution

Parameters **result** – a unittest.TestResult object

Returns an ExitCode object

8.8 Test-Message Handling

8.8.1 Handle common communication with device under test

By default, the integration test framework handles internal messaging and control flow using a message format defined in pykiso.Message. pykiso.test_message_handler defines the default messaging protocol from a behavioral point of view.

The general procedure is described in handle_basic_interaction context manager, but specific _MsgHandler_ classes are provided with *TestCaseMsgHandler* and *TestSuiteMsgHandler* to provide shorthands for the specialised communication from pykiso.test_case.BasicTest and pykiso.test_suite.BasicTestSuite.

module test_message_handler

synopsis default communication between TestManagement and DUT.

class pykiso.test_coordinator.test_message_handler.**TestCaseMsgHandler**

Encapsulate all test case communication mechanism.

classmethod **run**(*cls*, *test_entity*, *timeout_cmd*, *timeout_resp*)

Handle communication mechanism during test case run. Based on actual implementation, this context manager call handle_basic_interaction.

Parameters

- **test_entity** (Callable) – test instance in use (BaseTestSuite, BasicTest,...)
- **timeout_cmd** (int) – timeout in second apply on auxiliary run_command
- **timeout_resp** (int) – timeout in second apply on auxiliary wait_and_get_report

Returns namedtuple containing current auxiliary, reported message, logging method to use, and pre-defined log message.

classmethod **setup**(*cls*, *test_entity*, *timeout_cmd*, *timeout_resp*)

Handle communication mechanism during test case setup. Based on actual implementation, this context manager call handle_basic_interaction.

Parameters

- **test_entity** (Callable) – test instance in use (BaseTestSuite, BasicTest,...)
- **timeout_cmd** (int) – timeout in second apply on auxiliary run_command
- **timeout_resp** (int) – timeout in second apply on auxiliary wait_and_get_report

Returns namedtuple containing current auxiliary, reported message, logging method to use, and pre-defined log message.

classmethod **teardown**(*cls*, *test_entity*, *timeout_cmd*, *timeout_resp*)

Handle communication mechanism during test case teardown. Based on actual implementation, this context manager call handle_basic_interaction.

Parameters

- **test_entity** (Callable) – test instance in use (BaseTestSuite, BasicTest,...)
- **timeout_cmd** (int) – timeout in second apply on auxiliary run_command
- **timeout_resp** (int) – timeout in second apply on auxiliary wait_and_get_report

Returns namedtuple containing current auxiliary, reported message, logging method to use, and pre-defined log message.

class pykiso.test_coordinator.test_message_handler.**TestSuiteMsgHandler**

Encapsulate all test suite communication mechanisms.

> **Warning:** This class is speculative code and not currently used. it should eventually replace the handle_basic_interaction call in the BaseTestSuite

classmethod **setup**(*cls*, *test_entity*, *timeout_cmd*, *timeout_resp*)

Handle communication mechanism during test suite setup. Based on actual implementation, this context manager calls handle_basic_interaction.

Parameters

- **test_entity** (Callable) – test instance in use (BaseTestSuite, BasicTest,...)

- **timeout_cmd** (int) – timeout in second apply on auxiliary run_command
- **timeout_resp** (int) – timeout in second apply on auxiliary wait_and_get_report

Returns namedtuple containing current auxiliary, reported message, logging method to use, and pre-defined log message.

classmethod teardown(*cls*, *test_entity*, *timeout_cmd*, *timeout_resp*)

Handle communication mechanism during test suite teardown. Based on actual implementation, this context manager call handle_basic_interaction.

Parameters

- **test_entity** (Callable) – test instance in use (BaseTestSuite, BasicTest,...)
- **timeout_cmd** (int) – timeout in second apply on auxiliary run_command
- **timeout_resp** (int) – timeout in second apply on auxiliary wait_and_get_report

Returns namedtuple containing current auxiliary, reported message, logging method to use, and pre-defined log message.

pykiso.test_coordinator.test_message_handler.**handle_basic_interaction**(*test_entity*, *cmd_sub_type*, *timeout_cmd*, *timeout_resp*)

Handle default communication mechanism between test manager and device under test as follow:

```
TM |   COMMAND ----------->        | DUT
TM |           <---------- ACK    | DUT
TM |           <---------- LOG    | DUT
TM |   ACK     ----------->        | DUT
...
TM |           <---------- LOG    | DUT
TM |   ACK     ----------->        | DUT
TM |           <---------- REPORT | DUT
TM |   ACK     ----------->        | DUT
```

This behaviour is implemented here.

Logs can be sent to TM while waiting for report.

Parameters

- **test_entity** (Callable) – test instance in use (BaseTestSuite, BasicTest,...)
- **cmd_sub_type** (*MessageCommandType*) – message command sub-type (Test case run, setup,....)
- **timeout_cmd** (int) – timeout in seconds for auxiliary run_command
- **timeout_resp** (int) – timeout in seconds for auxiliary wait_and_get_report

Return type List[*report_analysis*]

Returns tuple containing current auxiliary, reported message, logging method to use, and pre-defined log message.

class pykiso.test_coordinator.test_message_handler.**report_analysis**(*current_auxiliary*, *report_message*, *logging_method*, *log_message*)

Create new instance of report_analysis(current_auxiliary, report_message, logging_method, log_message)

property current_auxiliary
Alias for field number 0

property log_message
Alias for field number 3

property logging_method
Alias for field number 2

property report_message
Alias for field number 1

8.9 test xml result

8.9.1 test_xml_result

module test_xml_result

synopsis overwrite xmlrunner.result to be able to add additional data into the xml report.

class pykiso.test_coordinator.test_xml_result.**TestInfo**(*test_result*, *test_method*, *outcome=0*, *err=None*, *subTest=None*, *filename=None*, *lineno=None*, *doc=None*)

This class keeps useful information about the execution of a test method. Used by XmlTestResult

Initialize the TestInfo class and append additional tag that have to be stored for each test

Parameters

- **test_result** (_XMLTestResult) – test result class
- **test_method** – test method (dynamically created eg: test_case.MyTest2-1-2)
- **outcome** (int) – result of the test (SUCCESS, FAILURE, ERROR, SKIP)
- **err** – error cached during test
- **subTest** – optional, refer the test id and the test description
- **filename** (Optional[str]) – name of the file
- **lineno** (Optional[bool]) – store the test line number
- **doc** (Optional[str]) – additional documentation to store

class pykiso.test_coordinator.test_xml_result.**XmlTestResult**(*stream=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>*, *descriptions=True*, *verbosity=1*, *elapsed_times=True*, *properties=None*, *infoclass=<class 'pykiso.test_coordinator.test_xml_result.TestInfo'>*)

Test result class that can express test results in a XML report. Used by XMLTestRunner

Initialize the _XMLTestResult class.

Parameters

- **stream** (TextIOWrapper) – buffered text interface to a buffered raw stream

- **descriptions** (bool) – include description of the test
- **verbosity** (int) – print output into the console
- **elapsed_times** (bool) – include the time spend on the test
- **properties** – junit testsuite properties
- **infoclass** (_TestInfo) – class containing the test information

report_testcase(*xml_testsuite*, *xml_document*)
Appends a testcase section to the XML document.

CHAPTER

NINE

CONTROLLING AN INSTRUMENT

The instrument-control command offers a way to interface with an arbitrary instrument, such as power supplies from different brands. The Standard Commands for Programmable Instruments (SCPI) protocol is used to control the desired instrument. This section aims to describe how to use instrument-control as an auxiliary for integration testing, and also how to interface directly with the instrument using the built-in command line interface (CLI).

9.1 Requirements

A successful pykiso installation as described in this chapter: *Install*

9.2 Integration Test Usage

The auxiliary functionalities can be used during integration tests.

Add Test file: See the dedicated section below: *Implementation of Instrument Tests*

Add Config File In your test configuration file, provide what is necessary to interface with the instrument:

- Chose between the VISASerial and the VISATcpip connector
- If you are using a serial interface, the *serial_port* must be provided in the connector configuration, and the *baud_rate* is optional.
- If you are using a tcpip interface, the *ip_address* must be provided in the connector configuration.
- Chose the InstrumentControlAuxiliary

Note: You cannot use the instrument-control auxiliary with a proxy.

- **The SCPI commands might be different or even not available depending on the instrument that you are using. If you prov**
  parameter and the instrument is recognized, the functions in the *lib_scpi* will automatically be adapted according to your instrument capabilities and specificities.
- If your instrument has more than one output channel, provide the one to use in *output_channel*.

Example of a test configuration file using instrument-control auxiliary:

Examples:

```yaml
# Connection to a local PSI 9000 T power supply from EA Elektro-Automatik GmbH & Co
auxiliaries:
  instr_aux:
    connectors:
      com: VISA
    config:
      instrument: "Elektro-Automatik"
    type: pykiso.lib.auxiliaries.instrument_control_auxiliary:InstrumentControlAuxiliary
connectors:
  VISA:
    config:
      serial_port: 5
    type: pykiso.lib.connectors.cc_visa:VISASerial
test_suite_list:
- suite_dir: test_suite_with_instruments
  test_filter_pattern: 'test*.py'
  test_suite_id: 1
```

```yaml
# Connection to the remote Rohde & Schwartz power supply
auxiliaries:
  instr_aux:
    connectors:
      com: Socket
    config:
      instrument: "Rohde&Schwarz"
      output_channel: 1
    type: pykiso.lib.auxiliaries.instrument_control_auxiliary:InstrumentControlAuxiliary
connectors:
  Socket:
    config:
      dest_ip: 'ENV{POWER_SUPPLY_IP}'
      dest_port: 3000
    type: pykiso.lib.connectors.cc_tcp_ip:CCTcpip
test_suite_list:
- suite_dir: test_suite_with_instruments
  test_filter_pattern: 'test*.py'
  test_suite_id: 1
```

9.2.1 Implementation of Instrument Tests

Using the instrument auxiliary (*instr_aux*) inside integration tests is useful to control the instrument (e.g. a power supply) the device under test is connected to. There are two different ways to interface with an instrument:

1. The first option is to use the *read*, *write*, and *query* commands to directly send SCPI commands to the instrument. If you use this method, refer to your instrument's datasheet to get the appropriate SCPI commands.
2. The other option is to use the built-in functionalities from the library to communicate with the instrument. For that, use the *lib_scpi* attribute of your *instru_aux* auxiliary.

You can then send *read*, *write* and *query* (*write* + *read*) requests to the instrument.

For example: #. To query the identification data of your instrument, you can use *instr_aux.query("*IDN?")* #. To set the voltage target value to 12V, you can use *instr_aux.write("SOUR:VOLT 12.0")*

Some helper commands have already been implemented to simplify the testing. For example, using helpers: #. To query the identification data of your instrument: *instr_aux.helpers.get_identification()*. #. To set the voltage target value to 12V: *instr_aux.helpers.set_target_voltage(12.0)*

Notice that the SCPI command can be different depending on the instrument. For some instrument, some features are also unav
Some instruments are already registered. If you specify the name of the instrument that you are using in the YAML file, the helpers function will select and use the SCPI commands that are appropriate or tell you if the command is not available.

When setting a parameter on the instrument, it is possible to use a validation procedure to make sure that the parameter was su
The validation procedure consists in sending a query immediately after sending the write command, the answer of the query will then tell if the write command was successful or not. For instance, in order to enable the output on the currently selected channel of the instrument, we can use *instr_aux.write("OUTP ON")*, or, using the validation procedure, *instr_aux.write("OUTP ON", ("OUTP?", "ON"))*. Notice that the validation parameter is a tuple of the form ('query to send to check the writing operation', 'expected answer') When the expected answer is a number, please use the "VALUE{}" tag. For instance, you can use *instr_aux.write("SOUR:VOLT 12.5", ("SOUR:VOLT?", "VALUE{12.5}"))*. That way, it does not matter if the instrument returns *12.50*, *12.500* or *1.25000E1*, the writing operation will be considered successful. Also, if you are not sure what your instrument will respond to the validation, you can compare that output to a list of string, instead on a single string. For example, you can use *instr_aux.write("OUTP ON", ("OUTP?", ["ON", "1"]))*. The *VALUE* should not passed inside a list. This validation procedure is used in all the helper functions (except reset)

The following integration test file will provide some examples:

instrument_test.py:

```
import logging
import time

import pykiso
from pykiso.auxiliaries import instr_aux


@pykiso.define_test_parameters(suite_id=1, case_id=1, aux_list=[instr_aux])
class TestWithPowerSupply(pykiso.BasicTest):
    def setUp(self):
        """Hook method from unittest in order to execute code before test case run."""
        logging.info(
            f"--------------- SETUP: {self.test_suite_id}, {self.test_case_id} ----------
↪-----"
        )

    def test_run(self):
        logging.info(
            f"--------------- RUN: {self.test_suite_id}, {self.test_case_id} ------------
↪---"
        )

        logging.info("---General information about the instrument:")
        # using the auxiliary's 'query' method
        logging.info(f"Info: {instr_aux.query('*IDN?')}")
        # using the commands from the library
        logging.info(f"Status byte: {instr_aux.helpers.get_status_byte()}")
        logging.info(f"Errors: {instr_aux.helpers.get_all_errors()}")
        logging.info(f"Perform a self-test: {instr_aux.helpers.self_test()}")
```

(continues on next page)

(continued from previous page)

```
        # Remote Control
        logging.info("Remote control")
        instr_aux.helpers.set_remote_control_off()
        instr_aux.helpers.set_remote_control_on()

        # Nominal values
        logging.info("---Nominal values:")
        logging.info(f"Nominal voltage: {instr_aux.helpers.get_nominal_voltage()}")
        logging.info(f"Nominal current: {instr_aux.helpers.get_nominal_current()}")
        logging.info(f"Nominal power: {instr_aux.helpers.get_nominal_power()}")

        # Current values
        logging.info("---Measuring current values:")
        logging.info(f"Measured voltage: {instr_aux.helpers.measure_voltage()}")
        logging.info(f"Measured current: {instr_aux.helpers.measure_current()}")
        logging.info(f"Measured power: {instr_aux.helpers.measure_power()}")

        # Limit values
        logging.info("---Limit values:")
        logging.info(f"Voltage limit low: {instr_aux.helpers.get_voltage_limit_low()}")
        logging.info(
            f"Voltage limit high: {instr_aux.helpers.get_voltage_limit_high()}"
        )
        logging.info(f"Current limit low: {instr_aux.helpers.get_current_limit_low()}")
        logging.info(
            f"Current limit high: {instr_aux.helpers.get_current_limit_high()}"
        )
        logging.info(f"Power limit high: {instr_aux.helpers.get_power_limit_high()}")

        # Test scenario
        logging.info("Scenario: apply 36V on the selected channel for 1s")
        dc_voltage = 36.0  # V
        dc_current = 1.0  # A
        logging.info(
            f"Set voltage to {dc_voltage}V: {instr_aux.helpers.set_target_voltage(dc_
↪voltage)}"
        )
        logging.info(
            f"Set voltage to {dc_current}V: {instr_aux.helpers.set_target_current(dc_
↪current)}"
        )
        logging.info(f"Switch on output: {instr_aux.helpers.enable_output()}")
        logging.info("sleeping for 1s")
        time.sleep(0.5)
        logging.info(f"measured voltage: {instr_aux.helpers.measure_voltage()}")
        logging.info(f"measured current: {instr_aux.helpers.measure_current()}")
        time.sleep(0.5)
        logging.info(f"Switch off output: {instr_aux.helpers.disable_output()}")

        logging.info(
            f"Trying to set an impossible value (1000V) {instr_aux.helpers.set_target_
↪voltage(1000)}"
```

(continues on next page)

(continued from previous page)

```
        )

    def tearDown(self):
        """Hook method from unittest in order to execute code after test case run."""
        logging.info(
            f"--------------- TEARDOWN: {self.test_suite_id}, {self.test_case_id} -------
↪--------"
        )
```

9.3 Command Line Usage

The auxiliary functionalities can also be used from a command line interface (CLI). This section provides a basic overview of exemplary use cases processed through the CLI, as well as a general overview of all possible commands.

9.3.1 Connection to the instrument

Every tine that the instrument-control CLI will be called, a connection to the instrument will be opened. Then, some actions and/or measurement will be done, and the connection will finally be closed. As a consequence, you should always give the necessary options to be able to connect to the instrument.

- Chose an interface (*VISA_SERIAL*, *VISA_TCPIP*, or *SOCKET_TCPIP*). Use *-i* or *–interface*. This option is mandatory.
- **Use the *-p/–port*, the *-ip/–ip-address*. Several option are available for the different interfaces:**
  - VISA_TCPIP: you must provide an ip address, the port is optional.
  - VISA_SERIAL: you must indicate the serial port to use.
  - SOCKET_TCPIP: you must have to set the ip address and a port.
- You can add a *-b/–baud-rate* option if you chose a SERIAL interface
- You can add a *–name* option to indicate that you are using a specific instrument. If this instrument is registered, the SCPI command specific to this instrument will be used instead of the default commands. For instance, selecting the output channel is not possible for Elektro-Automatik instruments because they only have one. The Rhode & Schwarz on the other hand does, so the corresponding commands are available.
- You can add a *–log-level* option to indicate the logging verbosity.

9.3.2 Performing measurement and setting values

You can then use other options to perform measurements and set values on your instrument. For that use the following options.

Flag options:

- Get the instrument identification information: *–identification*
- Resets the instrument: *–reset*
- Get the instrument status byte: *–status-byte*
- Get the errors currently stored in the instrument: *–all-errors*
- Performs a self test of the instrument: *–self-test*

- Get the instrument voltage nominal value: *–voltage-nominal*
- Get the instrument current nominal value: *–current-nominal*
- Get the instrument power nominal value: *–power-nominal*
- Measures voltage on the instrument: *–voltage-measure*
- Measures current on the instrument: *–current-measure*
- Measures power on the instrument: *–power-measure*

Options with values (specify a floating value for the parameter that you want to set on the instrument. If you want to get the value currently set on the instrument, write *get* instead of the numeric value)

- Instrument's output channel: *–output-channel*
- Instrument's voltage target value: *–voltage-target*
- Instrument's current target value: *–current-target*
- Instrument's power target value: *–power-target*
- Instrument's voltage lower limit: *–voltage-limit-low*
- Instrument's voltage higher limit: *–voltage-limit-high*
- Instrument's current lower limit: *–current-limit-low*
- Instrument's current higher limit: *–current-limit-high*
- Instrument's power higher limit: *–power-limit-high*

Other options with values:

- Instrument's remote control: *–remote-control*. Use *get* to get the remote control state, *on* to enable and *off* to disable the remote control on the instrument. - Instrument's output mode (output channel enable/disabled): *–output-mode*. Use *get* to get the remote control state, *enable* to enable and *disable* to disable the output of the currently selected channel of the instrument.

You can also send custom write and query commands:

- Send custom query command: *–query*
- Send custom write command: *–write*

9.3.3 Usage Examples

For all following examples, assume that we are connecting to a serial instrument on port COM4.

Requesting basic information from the instrument:

```
instrument-control -i VISA_SERIAL -p 4 --identification
```

Request basic information from the instrument via the SOCKET_TCPIP interface:

```
instrument-control -i SOCKET_TCPIP -ip 10.10.10.10 -p 5025 --identification
```

Reset the device with VISA_TCPIP interface and the address 10.10.10.10:

```
instrument-control -i VISA_TCPIP -ip 10.10.10.10 --reset
```

Also reset the instrument, but use the VISA_SERIAL on port 4 and set the baud rate to 9600:

```
instrument-control -i VISA_SERIAL -p 4 --baud-rate 9600 --reset
```

Get the currently selected output channel from a Rohde & Schwarz device

```
instrument-control -i SOCKET_TCPIP -ip 10.10.10.10 -p 5025 --name "Rohde&Schwarz" --
↪output-channel get
```

Set the output channel from a Rohde & Schwarz device to channel 3

```
instrument-control -i SOCKET_TCPIP -ip 10.10.10.10 -p 5025 --name "Rohde&Schwarz" --
↪output-channel 3
```

Read the target value for the current

```
instrument-control -i VISA_SERIAL -p 4 --current-target
```

Set the current target to 1.0 Ampere

```
instrument-control -i VISA_SERIAL -p 4 --current-target 1.0
```

Enable remote control on the instrument

```
instrument-control -i VISA_SERIAL -p 4 --remote-control ON
```

Set the voltage to 35 Volts and then enable the output:

```
instrument-control -i VISA_SERIAL -p 4 --voltage-target 35.0 --output-mode ENABLE
```

Get the instrument's identification information by sending custom a query command:

```
instrument-control -i VISA_SERIAL -p 4 --query *IDN?
```

Reset the instrument by sending a custom write command:

```
instrument-control -i VISA_SERIAL -p 4 --write *RST
```

Example interacting with a remote instrument:

Measuring the current voltage on channel 3:

```
instrument-control -i SOCKET_TCPIP -ip 10.10.10.10 -p 5025 --output-channel 3 --voltage-
↪measure
```

9.3.4 Interactive mode

The CLI includes an interactive mode. You can use it by adding the *–interactive* flag when you call the instrument-control CLI. Once you are inside this interactive mode, you can send commands one after the other. You may use all the available commands (you can remove the double dash).

Example:

1. Enter interactive mode,
2. get the identification information,
3. query the currently selected output channel,

4. set the output-channel to 3,
5. apply 36V,
6. and then measure the voltage.

```
instrument-control -i VISA_SERIAL -p 4 --identification get --interactive
output-channel
output-channel 3
remote-control on
voltage-target 36
output-mode enable
voltage-measure
exit
```

9.3.5 General Command Overview

```
instrument-control --help
```

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

M

N

O

P

Q

R

S

T

U

V

W

X