pykiso

Release 0.24.0

Sebastian Fischer, Daniel Bühler, Damien Kayser, Uwe Lang, Seba

Aug 22, 2023

CONTENTS:

CHAPTER

ONE

WHAT'S NEW IN PYKISO?

1.1 Version ongoing

1.2 Version 0.24.0

1.2.1 Search size for rtt segger

Added the possibility to search in a given range for the block of the device starting from the specified block address.

1.2.2 Tags in global-config

Added tags to global config.

1.2.3 Overall improvement

For this release, we focused on improving internal functionalities:

- Improve log for uds auxiliary for performance
- Use TrcReader from python-can for trc merging

1.3 Version 0.23.0

1.3.1 New expand and collapse buttons for the StepReport

Added buttons to expand or collapse all or all failed tests.

1.3.2 Pytest integration

Pykiso test suites can now be run without any further change with `pytest`.

For more information please refer to *Pytest Integration*.

1.3.3 Connect to serial devices via pid/vid

Added the possibility to connect to serial devices using pid and vid.

1.3.4 Timestamp in pykiso message dictionary

Added the timestamp to the pykiso message dictionary for CAN dongles.

1.3.5 Add changeable logger class for the test

Add the possibility to change the logger class with the CLI. For more information please refer to *How to change the class of the logger*.

1.3.6 Add shutdown method to CCHannel

Add proper shutdown method for connector instead of instead of using the del method

1.3.7 Connect to serial devices via serial number

Added the possibility to connect to serial devices using the serial number.

1.4 Version 0.22.1

1.4.1 Improved logging for multiple yaml

It is now possible to use a separate log file for each YAML if multiple are provided. A folder can also be passed in which case separate log files will be created with the corresponding YAML name.

1.4.2 Improved Report name

For trace ability the report name will now also include the name of the used config file.

1.4.3 Entirely hidden Proxy Auxiliary

There is now no need at all anymore to manually define a `ProxyAuxiliary` and `CCProxy`s.

If `auto_start` is disabled for all auxiliaries sharing a communication channel, then the `ProxyAuxiliary` won't be started and the shared communication channel will remain closed.

Afterwards, the shared communication channel can be opened and closed by respectively starting one of the auxiliaries and stopping all of the auxiliaries.

For more information, please refer to *Sharing a communication channel between multiple auxiliaries*.

1.5 Version 0.21.2

1.5.1 Remove redundancy of activate_log configuration parameter

In order to activate external loggers, it was previously necessary to specify their names for each defined auxiliary.

This is no longer the case and specifying them in only one auxiliary will be enough for the loggers to stay enabled.

1.5.2 Internal creation of proxy auxiliaries

It is no longer necessary to manually define a `ProxyAuxiliary` with `CCProxy` instances yourself. If you simply pass the communication channel to each auxiliary that has to share it, `pykiso` will do the rest for you.

For more information see *Sharing a communication channel between multiple auxiliaries*

1.5.3 Better skipping of test cases based on tags

The test case filtering strategy based on the test tags has been reworked. The resulting skipped test cases now appear explicitly as skiped in the test run output

For more information refer to *Filter the test cases to run with tags*

1.5.4 Ykush Auxiliary

Auxiliary that can be used to power on and off the ports of an Ykush USB Hub.

See *ykush_auxiliary*

1.5.5 Multiple auxiliaries can share a communication channel

see *Sharing a communication channel between multiple auxiliaries*

1.5.6 Remove raw from connector functions

The functions cc_receive and cc_send for every connector no longer take raw as an argument.

1.5.7 Results can be exported on TestRail

see *Export results on TestRail*

1.5.8 Step report

Tests are now foldable items.

1.6 Version 0.20.0

1.6.1 Auxiliary without connector

Makes possible to handle new auxiliary types that do not need a connector

See *Auxiliary without connector*

1.6.2 Process connector

The Process channel provides functionality to start a process and to communicate with it

See *cc_process*

1.6.3 Suite id and Case id are optional for BasicTest

See *Define the test information*

1.7 Version 0.19.3

1.7.1 Double Threaded Auxiliary Interface

Implement a brand new interface using two threads, one for the transmission and one for the reception.

Adapted modules for this release: - Proxy Auxiliary - CCProxy channel - Communication Auxiliary - DUT Auxiliary - Record Auxiliary - Acroname Auxiliary - Instrument Auxiliary - UDS Auxiliary - UDS server Auxiliary

There is not API changes, therefor, as user, your tests should not be affected.

1.7.2 Kiso log levels

To let users decide the level of information they want to see in their logs, new log levels have been defined. When launched normally only the logs in the tests and the errors will be active. The option -v (–verbose) should be used to display the internal logs of the framework.

See *Test verbosity*

1.7.3 Filter for Test cases

It is now possible to select test classes or even test cases with a unix filename pattern. This pattern can be passed with the -p flag or inside the yaml file.

For more info see *Run single tests*

1.7.4 Agnostic tag call

Instead of having only the 2 tags “variant” and “branch_level” to select tests, users can now set any tagname.

See: *Define the test information* for more details.

1.7.5 Step report (better reports for system-tester)

It tracks each assertion containing a message to create a comprehensive test-report.

See: *System-test (step) report* for more details.

1.7.6 Agnostic CCSocketCan

Incompatibilities with the agnostic proxy are now resolved. You should be able to use it again.

1.7.7 Tester Present Sender

Add a context manager, tester present sender, that send cyclic tester present frames to keep UDS session alive more than 5 seconds

It can also be started and stopped by using the following methods:
start_tester_present_sender() and *stop_tester_present_sender()*.

See *UDS tester present sender* See *Using UDS protocol*

1.7.8 RTT connector log folder creation

RTT connector now creates a log folder if it does not exist instead of throwing an error.

1.7.9 Communication Auxiliary

To save on memory, the communication auxiliary does not collect received messages automatically anymore. The functionality is now available with the context manager `collect_messages`.

See `kiso-testing/examples/templates/suite_com/test_com.py`.

The collected messages by the Communication auxiliary can still be cleared with the API method :py:meth`~pykiso.lib.auxiliaries.communication_auxiliary.CommunicationAuxiliary.clear_buffer`

See *communication_auxiliary*

1.7.10 Configurable waiting for send_uds_raw

To avoid extra waiting time during long/heavy UDS data exchange(flashing) expose the parameter tpWaitTime from kiso-testing-python-uds for uds auxilary send_uds_raw method

See *Using UDS protocol*

1.7.11 Lightweight UDS auxiliary configuration

The add of an .ini file to configured the UDS auxiliary and it variant (server) is no more mandatory, every parameter is now reachable in the .yaml file.

See `kiso-testing/examples/uds.yaml`.

In addition, if the tp_layer and uds_layer parameters are not given at yaml level a default configuration is applied.

See *Using UDS protocol*

1.7.12 New serial connector

Added cc_serial for serial communication.

1.7.13 Tool for test suites tags analysis

See *Show and export test suite tags*

1.8 Version 0.18.0

1.8.1 Remote approach for test

Remove all that is remote specific from BasicTestXXXX. The remote approach is now handle by RemoteTestXXXX.

1.8.2 Pykiso To Pytest

See *Pykiso to Pytest*

1.8.3 UDS Server Auxiliary

A UDS auxiliary acting as a server/ECU is now implemented. It is based on user-defined callbacks that send a UDS response when the defined request is received.

See *UDS protocol handling as a server*

1.9 Version 0.17.0

1.9.1 Access Framework's Configuration

All parameters given at CLI and yaml level are available for each test cases/suites. This allows you to access configuration parameters from the CLI and the yaml config. See *Access framework's configuration*

1.9.2 "Tag" Are The New "Variant"

The "tag " argument in the pykiso decorator define_test_parameters has been changed to "tag". See *Define the test information*

1.9.3 New Python Package Management

Poetry will now be used to manage pykiso. For more details see the official Poetry web site.

1.10 Version 0.16.0

1.10.1 Global Config

Configuration paramertes can now be passed from the cli or the yaml file into your test.

See *Access framework's configuration*

1.10.2 Fail Fast

With the new –fail-fast flag which can be passed through the pykiso cli, tests will be stopped on the first error or failure.

At the same time the behavior of the auxiliary create instance was changed. When the auxiliary instance creation are now failing, they will raise an exception. Combined with the new flag the test execution will be stopped when something go wrong.

1.10.3 Include Sub-YAMLs

Frequently used configuration parts can be stored in a separate YAML file.

See *Include sub-YAMLs*

CHAPTER

TWO

GETTING STARTED

2.1 User Guide

2.1.1 Requirements

- Python 3.7+
- pip

2.1.2 Install

```
pip install pykiso # Core framework
pip install pykiso[plugins] # To install all plugins
pip install pykiso[can] # To enable you to use CAN related plugins
pip install pykiso[debugger] # To enable you to use JLINK and else related plugins
pip install pykiso[instrument] # To enable you to use instrument control plugins

pip install pykiso[all] # To enable you to install everything we have to offer
```

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

```
git clone https://github.com/eclipse/kiso-testing.git
cd kiso-testing
poetry install --all-extras
poetry shell
```

2.1.3 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.

  TAG Filters: any additional option to be passed to the test as tag through
  the pykiso call. Multiple values must be separated with a comma.
```

(continues on next page)

(continued from previous page)

```
  For example: pykiso -c your_config.yaml --branch-level dev,master --variant
  delta

Options:
  -c, --test-configuration-file FILE
                                  path to the test configuration file (in YAML
                                  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
  --step-report PATH              generate the step report at the specified
                                  path
  --failfast                      stop the test run on the first error or
                                  failure
  -v, --verbose                   activate the internal framework logs
  -p, --pattern TEXT              test filter pattern, e.g. 'test_suite_1.py'
                                  or 'test_*.py'. Or even more granularly
                                  'test_suite_1.py::TestClass::test_name'
  --logger TEXT                   use the specified logger class in pykiso
  --version                       Show the version and exit.
  -h, --help                      Show this message and exit.
```

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

Demo using example config

```
pykiso -c ./examples/dummy.yaml --log-level=DEBUG -l killme.log
```

2.1.4 Basic configuration

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: pykiso.lib.auxiliaries.dut_auxiliary:DUTAuxiliary
  aux2:
    connectors:
        com:   chan2
    type: pykiso.lib.auxiliaries.dut_auxiliary:DUTAuxiliary
  aux3:
    connectors:
```

(continues on next page)

(continued from previous page)

```
        com:   chan4
        flash: chan3
    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: null
    type: pykiso.lib.connectors.cc_flasher_example:FlasherExample
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

requirements:
  - pykiso : '>=0.10.1'
  - robotframework : 3.2.2
  - pyyaml: any
```

Connectors

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

```
    connectors:
        com:   chan4
        flash: chan3
    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
```

(continues on next page)

(continued from previous page)

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

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: pykiso.lib.auxiliaries.dut_auxiliary:DUTAuxiliary
  aux2:
    connectors:
        com:   chan2
    type: pykiso.lib.auxiliaries.dut_auxiliary:DUTAuxiliary
  aux3:
```

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
```

Test Suites

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

```
  chan4:
    type: ext_lib/cc_example.py:CCExample
  chan3:
    config: null
    type: pykiso.lib.connectors.cc_flasher_example:FlasherExample
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*.

For fast test development, the *test_filter_pattern* can be overwritten from the command line in order to e.g. execute a single test file inside the *suite_dir* using the CLI argument *-p* or *–pattern*:

```
pykiso -c dummy.yaml
```

To learn more, please take a look at *How to make the most of the config file*.

2.1.5 Basic test writing

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 *How to make the most of the config 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).

Note: User can run several test using several times flag -c. If a folder path is specified, a log for each yaml file will be stored. If otherwise a filename is provided, all log information will be in one logfile.

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

`suite_id` and `case_id` are used to coordinate and define a clear test execution order. It is now optional for `pykiso.BasicTest` but still mandatory in case of `pykiso.RemoteTest` as the TestApp protocol relies on these identifiers.

If none of these IDs are defined, the default value of 0 will be used and the alphabetical order will be applied on each .py module and each contained test class.

In other words, If we have the following test suite folder organisation:

test_suite_folder
├— a.py module with classes B/C/A (declare in this order)
├— b.py module with classes Z/G (declare in this order)
├— c.py module with classes S/T + Suite Setup/Teardown

The framework will execute the tests in the following order:

```
test_suite_setUp (c.SuiteSetup-0-0)
test_run (a.A-0-0)
test_run (a.B-0-0)
test_run (a.C-0-0)
test_run (b.G-0-0)
test_run (b.Z-0-0)
test_run (c.S-0-0)
test_run (c.T-0-0)
test_suite_tearDown (c.SuiteTearDown-0-0)
```

In order to make use of the SetUp/TearDown test-suite feature, users have to define a class inheriting from *BasicTestSuiteSetup* or *BasicTestSuiteTeardown*. For each of these classes, the following methods `test_suite_setUp` or `test_suite_tearDown` must be overridden with the behaviour you want to have.

Note:

Because the python unittest module is used in the background, all methods
starting with “def test_” are executed automatically

Note: If a test in SuiteSetup raises an exception, all tests which belong to the same suite_id will be skipped.

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:
```

(continues on next page)

(continued from previous page)

```
-> suite_id : set to 1
-> case_id : Parameter case_id is not mandatory for setup.
-> aux_list : used aux1 and aux2 is used
"""
@pykiso.define_test_parameters(suite_id=1, aux_list=[aux1, aux2])
class SuiteSetup(pykiso.BasicTestSuiteSetup):
    def test_suite_setUp():
        logging.info("I HAVE RUN THE TEST SUITE SETUP!")
        if aux1.not_properly_configured():
            aux1.configure()
        aux2.configure()
        callback_registering()

"""
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
"""
@pykiso.define_test_parameters(suite_id=1, aux_list=[aux1, aux2])
class SuiteTearDown(pykiso.BasicTestSuiteTeardown):
    def test_suite_tearDown():
        logging.info("I HAVE RUN THE TEST SUITE TEARDOWN!")
        callback_unregistering()

"""
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
"""
@pykiso.define_test_parameters(
        suite_id=1,
        case_id=1,
        aux_list=[aux1, aux2]
)
class MyTest(pykiso.BasicTest):
    pass
```

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>.
"""
```

(continues on next page)

(continued from previous page)

```
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])
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
```

How are the tests called

Let us imagine we have 2 test-cases which are part of a test-suite.

```
import pykiso
from pykiso.auxiliaries import aux1, aux2

@pykiso.define_test_parameters(suite_id=1, aux_list=[aux1, aux2])
class SuiteSetup(pykiso.BasicTestSuiteSetup):
    pass

@pykiso.define_test_parameters(suite_id=1, aux_list=[aux1, aux2])
class SuiteTearDown(pykiso.BasicTestSuiteTeardown):
    pass

@pykiso.define_test_parameters(suite_id=1, case_id=1, aux_list=[aux1])
class TestCase1(pykiso.BasicTest):
    def setUp(self):
```

(continues on next page)

(continued from previous page)

```
        pass
    def test_run_1(self):
        pass
    def test_run_2(self):
        pass
    def tearDown(self):
        pass

@pykiso.define_test_parameters(suite_id=1, case_id=1, aux_list=[aux1])
class TestCase2(pykiso.BasicTest):
    def setUp(self):
        pass
    def test_run_1(self):
        pass
    def test_run_2(self):
        pass
    def tearDown(self):
        pass
```

The pykiso will call the elements in the following order:

```
TestSuiteSetup().test_suite_setUp

    TestCase1.setUpClass

        TestCase1().setUp
        TestCase1().test_run
        TestCase1().tearDown

        TestCase1().setUp
        TestCase1().test_run_2
        TestCase1().tearDown

    TestCase1.tearDownClass

    TestCase2.setUpClass

        TestCase2().setUp
        TestCase2().test_run
        TestCase2().tearDown

        TestCase2().setUp
        TestCase2().test_run_2
        TestCase2().tearDown

    TestCase2.tearDownClass

TestSuiteTeardown().test_suite_tearDown
```

To learn more, please take a look at *How to make the most of the tests*.

2.2 Contribution Guide

2.2.1 What should I do before I get started?

You need to go through few steps to get the ball rolling. But no worries, it is pretty straightforward.

2.2.2 Accounts

First of all, you need accounts for:

- Github account, some of you might already have one. If not, you can go to github and register a free user account in 2 minutes.

Note: If you are working for a company and the work you are going to contribute is in the name of the company, please register your account using company email address.

- Eclipse account, since Kiso-testing is an Eclipse project (full name: Eclipse Kiso-testing), you need an Eclipse account. Go to https://accounts.eclipse.org/user/register to register one for free.

After a successful registration, you need to hook up your github account with Eclipse account. Login in Eclipse foundation website and go to 'Edit My Profile' where you can bind your github account information.

2.2.3 ECA signing

ECA stands for 'Eclipse Contributor Agreement', which is a prerequisite to become a contributor. No paper work needed, go to https://www.eclipse.org/legal/ECA.php , read it carefully and follow its instruction to sign.

2.2.4 DCO signing

DCO stand for "Developer's Certificate of Origin", which you will encounter as part of ECA signing process. It is highly recommended that you read it, while you as a developer might overlook the legal consequences if the way you contribute does not follow certain rules and regulations.

2.2.5 License

License is one of the few first things people would think of, when they use or develop an open source project. Eclipse Kiso is and will be developed under the EPL v2.0 license from Eclipse foundation. Of course this exclude 3rd party source code.

EPL v2.0 is available under https://www.eclipse.org/legal/epl-2.0/ . You need read it carefully before using Kiso-testing or developing on Kiso-testing and make sure that you understand your rights and obligations.

Any contributions to Kiso-testing project code base needs to be licensed under EPL v2.0.

2.2.6 How to setup my environment?

Requirements

- Python 3.7+
- poetry (used to get the rest of the requirements)

Install

```
git clone https://github.com/eclipse/kiso-testing.git
cd kiso-testing
poetry install --all-extras
poetry shell
```

Pre-Commit

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

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

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
```

Demo using example config

```
invoke run
```

Running the Tests

```
invoke test
```

or

```
pytest
```

Building the Docs

```
invoke docs
```

2.2.7 What should I do before committing ?

2.2.8 PEP8-Compliancy

In order to maintain clear and user-friendly project, make sure that your changes respect PEP8 standards. PEP8 is a guide that provides Python coding conventions (naming, indentation,...). Official document : https://peps.python.org/pep-0008/

To make sure your changes are PEP8 Compliant, different tools exist to help you here:

- **linter (applicable on IDE) :**
  show some warning directly on IDE.
- **pre-commit hook :**
  hook scripts that lint the added code using flake8 and format it using black and isort.

2.2.9 Typography

Most of the comments made during PR-Reviewing are about typography/misspelling mistakes. An easy way to avoid these is by running codespell on your written code.

2.2.10 Function type hinting

In kiso-testing, every implemented function must have annotations for its parameters and return types (type hints). This results in increased readability and therefore in easier comprehension of the code for any reader.

```
def some_fun(some_dict_param: dict, some_string_param: str) -> list:
```

Note: As not every types are available in the builtins, or as it is important to precise inner type you might import some from collections module or typing

```
from typing import List
from collections import namedtuple

def some_fun(
    some_int_list_param: List[int], some_imported_type_param: namedtuple
) -> list:
```

2.2.11 Unit Testing

To ensure the correct behaviour of your code, add unit tests for every function you implemented.A convenient and pythonic way to do this is this is given by pytest Code coverage is measured with codecov. It simply checks if the code coverage is not going lower than it was before changes.

2.2.12 Examples Adaptation

To ensure proper integration of your changes into the existing features, and demonstrate their usages, adapt the examples of modified module, and run it locally.

2.2.13 Update Documentation

Regarding documentation there are four main purposes that have to be fulfilled before committing :

- **Documentation regarding the changes :**
  Make sure that the documentation allow easy understanding of the new feature(s).This step mainly concern docstring (module, class, and function) as shown below, but you could also have to change .rst documentation if your changes concern general working principle of the ITF (e.g. cli) To ensure proper formatting of the documentation, run `invoke docs` in the poetry environment.

```
"""
name_of_the_module
******************

:module: name_of_the_module
:synopsis: short description of the module.

Extended description of the module's functionnality,
how it works, etc

.. currentmodule:: name_of_the_module
"""
```

```
class ClassName:
"""Short description of class"""
```

```
def fun(param1, param2):
"""Short description of fun.

More extended description of the function
if needed.

:param param1: short description of param1
:param param2: short description of param2
:raise exception1: short description of raised exception

:return: short description of the return parameter
"""
```

Note: Make also sure to do the type hinting for exceptions

- **What's new section:**
  Add your changes into the what's new section, so user can stay updated of the brand new features.
- **Changelog: (automatically updated)**
  Your commit needs to follow the [conventional commits](https://www.conventionalcommits.org/en/v1.0.0/) pattern. Changelog is updated automatically with the commit message.
- Documentation has to build properly.

CHAPTER

THREE

PYKISO DESIGN

3.1 Introduction

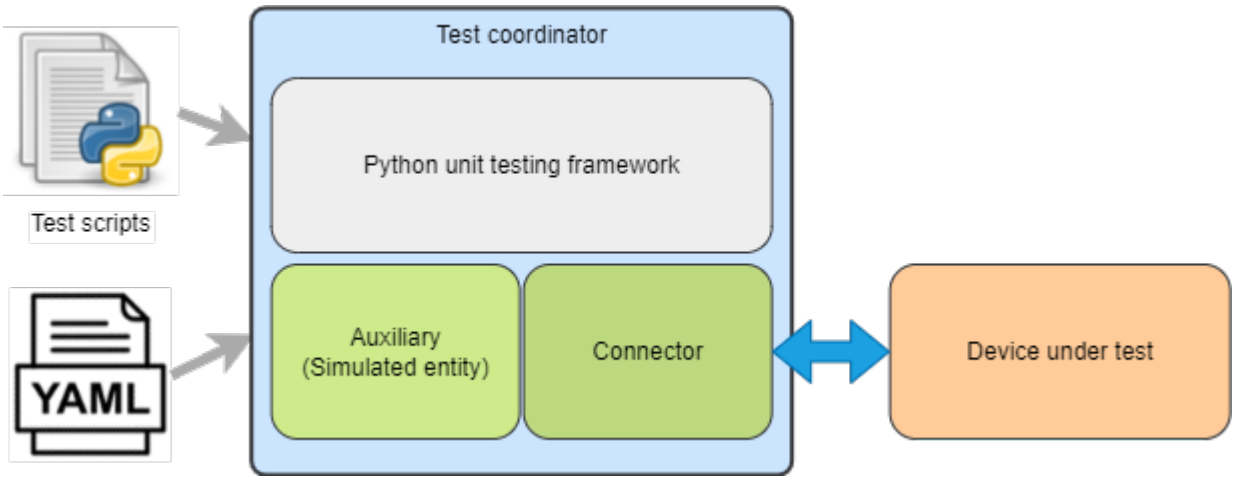
The *pykiso* Integration Test Framework (ITF) build in a modular and configurable way, which aims to enable the user, to write and run tests on hardware. Pykiso is build to orchestrate the entities (e.g. device under test) and services (e.g. flash the target) which are involved in the test. The Testing Framework can be used for both white-box and black-box testing as well as in the integration and system testing.

3.2 Quality Goals

The Framework tries to achieve the following quality goals:

Quality Goal (with prio)	Scenarios
Portability	The Framework shall run on linux, windows, macOS
	The Framework shall run on a raspberryPI or a regular laptop
Modularity	The Framework shall allow me to implement complex logic and to run it over any communication port
	The Framework shall allow me to add any communication port
	The Framework shall allow me to use private modules within my tests if it respects its APIs
	The Framework shall allow me to define my own test approach
Correctness	The Framework shall verify that its inputs (test-setup) are correct before performing any test
	The Framework shall execute the provided tests always in the same order
Usability	The Framework shall feel familiar for embedded developers
	The Framework shall feel familiar for system tester
	The Framework shall generate test reports that are human and machine readable
Performance (new)	The Framework shall use only the right/reasonable amount of resources to run (real-time timings)

3.3 Design Overview



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

As illustrated in Figure 1, the *pykiso* ITF is composition of a *test-coordinator*, *auxiliaries* and their corresponding *connectors*.

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.

3.3.1 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
- for remote tests (see *Remote Test*) flash and run and synchronize the tests on the auxiliaries
- gather the reports and publish the results

3.3.2 Auxiliary

The **auxiliary** provides to the **test-coordinator** an interface to interact with the physical or digital device under test. It can be seen as a helping hand for the test-coordinator to communicate with the device under test, designated by the term *auxiliary*. It is composed by 2 blocks:

- instance creation / deletion
- connectors to facilitate interaction and communication with the device (e.g. messaging with *UART*)

If for example an auxiliary needs to interact with cloud services, it could have:

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

Create an Auxiliary

Detailed information can be found here *How to create an auxiliary*.

Existing Auxiliarys

acroname_auxiliary - control an acroname usb hub.

communication_auxiliary - used to for raw byte communication.

dut_auxiliary - allows to flash and run test on the target device.

instrument_control_auxiliary - interface to arbitrary instrument (e.g. power supplies).

mp_proxy_auxiliary - multiprocessing proxy auxiliary

proxy_auxiliary - connect multiple auxiliaries to one unique connector.

record_auxiliary - logging from connector.

simulated_auxiliary - simulated device under test.

Using UDS protocol - uds requests.

UDS protocol handling as a server - simulated uds ecu.

ykush_auxiliary - control Yepkit USB hub.

Detailed information about included auxiliarys can be found here *Existing Auxiliaries*.

3.3.3 Connector

The communication between the *test-coordinator* and the *device under test* via the *auxiliary* is enabled by the corresponding connector. The connector can be used either as a *communication channel* or a *flasher*.

Communication Channel

The Communication Channel - also known as **cchannel** - is the medium which enables the communication with the device under test. Example include *UART*, *UDP*, *USB*, *REST*,... The communication protocol itself can be auxiliary specific.

Create a Connector

Detailed information can be found here *How to create a connector*.

3.3.4 Dynamic Import Linking

The *pykiso* Framework was developed with modularity and reusability in mind. To avoid close coupling between testcases and auxiliaries as well as between auxiliaries and connectors, the linking between those components is defined in a config file (see *How to make the most of the config 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.

3.4 Usage

Please see *How to make the most of the config file* to have a deep-dive on how the pykiso configuration work.

Please see *How to make the most of the tests* to have a deep-dive on how pykiso tests work.

CHAPTER

FOUR

ADVANCED USAGE

4.1 How to make the most of the config file

4.1.1 Requirements specification

[optional] - Any package specified will be checked.

Use cases:

- A new feature is introduced and used in the test
- Breaking change introduced with a new release
- Specific package used in a test that does not belong to pykiso

```
#_____________________________ Requirements section ____________________________
# FEATURE: Check the environment before running the tests
#
# The version can be:
#   - specified alone (minimum version accepted)
#   - conditioned using <, <=, >, >=, == or !=
#   - no specified using 'any' (accept any version)
#
# /!\: If the check fail, the tests will not start and the mismatch
#      displayed
#_____________________________________________________________________________
requirements:
  - pykiso: '>=0.10.1, <1.0.0'
  - robotframework: 3.2.2
  - pyyaml: any
```

4.1.2 Real-World Configuration File

```
auxiliaries:
  DUT:
    connectors:
        com: uart
    config: null
    type: pykiso.lib.auxiliaries.example_test_auxiliary:ExampleAuxiliary
connectors:
  uart:
```

(continues on next page)

(continued from previous page)

```
    config:
      serialPort: COM3
    type: pykiso.lib.connectors.cc_uart:CCUart
test_suite_list:
- suite_dir: test_suite_1
  test_filter_pattern: '*.py'
  test_suite_id: 1
```

4.1.3 Activation 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 activates 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.

4.1.4 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.

```
- suite_dir: ENV{TEST_SUITE_1}
  test_filter_pattern: '*.py'
  test_suite_id: 1
- suite_dir: ENV{TEST_SUITE_2=./test_suite_2}
```

It is also possible to set a default value in case the environment variable is not found. The following format should be used: `ENV{my-env-var=my_default_value}`.

In the following example, an environment variable called TEST_SUITE_2 would contain the path to the test_suite_2 directory. If the variable is not set, the default value will be taken instead.

```
    config: null
```

4.1.5 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**.

According to the YAML specification, values enclosed in single quotes are enforced as strings, and **will not be parsed**.

```
example_config:
    # this relative path will not be made absolute
    rel_script_path_unresolved: './script_folder/my_awesome_script.py'
    # this one will
    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
```

Warning: Relative path or file locations must always start with `./`. If not, it will still be resolved but unexpected behaviour can result from it.

4.1.6 Include sub-YAMLs

Frequently used configuration parts can be stored in a separate YAML file. To include this configuration file in the main one, the path to the sub-configuration file has to be provided, preceded with the *!include* tag.

Relative paths in the sub-YAML file are then resolved **relative to the sub-YAML's location**.

```
#_____________________________ Connectors section _____________________________
# FEATURE : yaml in yaml
#
# In order to call a yaml file inside a other yaml file, the special
# tag !include has to be used. The path could be in a relative or
# absolute form.
#_______________________________________________________________________________
connectors:
  <chan>: !include ./channel_config/my_channel_config.yaml
```

4.1.7 Delay an auxiliary start-up

All threaded auxiliaries are capable to delay their start-up (not starting at import level). This means, from user point of view, it's possible to start it on demand and especially where it's really needed.

Warning: in an explicitely defined proxy setup (for shared communication channels) be sure to always start the proxy auxiliary last. Otherwise, an error will occur due to the specific *ProxyAuxiliary* import rules.

To avoid such corner cases, consider switching to an *implicit definition of shared communication channels*.

In order to achieved that, a parameter was added at the auxiliary configuration level.

```
auxiliaries:
  proxy_aux:
    connectors:
        com: can_channel
    config:
      aux_list: [aux1, aux2]
      activate_trace: True
      trace_dir: ./suite_proxy
      trace_name: can_trace
      # if False create the auxiliary instance but don't start it, an
      # additional call of start method has to be performed.
      # By default, auto_start flag is set to True and "normal" ITF aux
      # creation mechanism is used.
      auto_start: False
    type: pykiso.lib.auxiliaries.proxy_auxiliary:ProxyAuxiliary
  aux1:
    connectors:
      com: proxy_com1
    config:
      auto_start: False
    type: pykiso.lib.auxiliaries.communication_auxiliary:CommunicationAuxiliary
  aux2:
    connectors:
      com: proxy_com2
    config:
      auto_start: False
    type: pykiso.lib.auxiliaries.communication_auxiliary:CommunicationAuxiliary
```

In user's script simply call the related auxiliary start method:

```
    used for sending and the other one for the reception
    """

    def setUp(self):
        """If a fixture is not use just override it like below."""
        logging.info(
            f"--------------- SETUP: {self.test_suite_id}, {self.test_case_id} ----------
↪-----"
```

4.1.8 Sharing a communication channel between multiple auxiliaries

Internal patching of the configuration

In order to attach a single communication channel to multiple defined auxiliaries, it is sufficient to use the same channel name for all auxiliaries.

Under the hood, `pykiso` will automatically create a *ProxyAuxiliary* that will have exclusive access to the communication channel. A *CCProxy* will be plugged between each declared auxiliary and the created *ProxyAuxiliary*. This allows to keep a minimalistic *CChannel* implementation.

Access to attributes and methods of the defined communication channel that has been attached to the created *ProxyAuxiliary* is still possible, but keep in mind that if one auxiliary modifies one the communication channel's

attributes, every other auxiliary sharing this communication channel will be affected by this change.

An illustration of the resulting internal setup can be found at *proxy_auxiliary*.

In other words, if you define the following YAML configuration file:

```yaml
auxiliaries:
  aux1:
    connectors:
      com: chan1
    config: null
    type: pykiso.lib.auxiliaries.dut_auxiliary:DUTAuxiliary
  aux2:
    connectors:
      com: chan1
    type: pykiso.lib.auxiliaries.communication_auxiliary:CommunicationAuxiliary

connectors:
  chan1:
    config: null
    type: pykiso.lib.connectors.cc_example:CCExample
```

Then, pykiso will internally modify this configuration to become:

```yaml
auxiliaries:
  proxy_aux:
    connectors:
      com: chan1
    config:
      aux_list: [aux1, aux2]
    type: pykiso.lib.auxiliaries.proxy_auxiliary:ProxyAuxiliary
  aux1:
    connectors:
      com: cc_proxy_aux1
    config: null
    type: pykiso.lib.auxiliaries.dut_auxiliary:DUTAuxiliary
  aux2:
    connectors:
      com: chan2
    type: pykiso.lib.auxiliaries.communication_auxiliary:CommunicationAuxiliary
  connectors:
    chan1:
      config: null
      type: pykiso.lib.connectors.cc_example:CCExample
    cc_proxy_aux1:
      config: null
      type: pykiso.lib.connectors.cc_proxy:CCProxy
    cc_proxy_aux2:
      config: null
      type: pykiso.lib.connectors.cc_proxy:CCProxy
```

Note: Keep in mind that an auxiliary connected through a *ProxyAuxiliary* will receive the other auxiliaries' messages.

Delayed startup

If the shared communication channel needs to be opened at a later point of a test, it is also possible to apply the *delayed auxiliary startup* to each auxiliary that is connected to this communication channel.

The communication channel will then be opened as soon as one of the auxiliaries holding this channel is started.

Taking as reference the example configuration file above, the delayed startup can be achieved by simply setting the `auto_start` flag to `False` for each auxiliary:

```
auxiliaries:
  aux1:
    connectors:
      com: chan1
    config:
      auto_start: False
    type: pykiso.lib.auxiliaries.dut_auxiliary:DUTAuxiliary
  aux2:
    connectors:
      com: chan1
    config:
      auto_start: False
    type: pykiso.lib.auxiliaries.communication_auxiliary:CommunicationAuxiliary

connectors:
  chan1:
    config: null
    type: pykiso.lib.connectors.cc_example:CCExample
```

Opening and closing a shared communication channel

Within the test case where a delayed startup is wished, the shared communication channel can then be opened and closed without any need to interact with the internally created *ProxyAuxiliary*.

As a simple rule of thumb, a shared communication channel:

- is opened when the first connected auxiliary is started
- is closed when the last connected auxiliary is stopped.

This can be illustrated in the following test example, based on the configuration shown above:

```
import pykiso
from pykiso.auxiliaries import aux1, aux2

@pykiso.define_test_parameters(suite_id=1, case_id=1)
class MyTest1(pykiso.BasicTest):
    """This test case definition will override the setUp, test_run and tearDown method."
↪"

    def setUp(self):
        """
        Execute code before running the test case.
        No auxiliary is currently running as their startup was delayed.
        """
```

(continues on next page)

(continued from previous page)

```
        # do something without the imported auxiliaries

    def test_run(self):
        """Test case that stops and starts the auxiliaries."""

        aux1.start()   # chan1 attached to aux1 and aux2 is opened

        self.assertTrue(aux1.is_instance, "aux1 was not successfully started!")
        self.assertFalse(aux2.is_instance, "aux2 is unexpectedly running!")

        aux2.start()   # chan1 attached to aux1 and aux2 stays open

        self.assertTrue(aux2.is_instance, "aux2 was not successfully started!")

        aux1.stop()    # chan1 attached to aux1 and aux2 stays open as aux2 is still␣
↪running

        self.assertFalse(aux1.is_instance, "aux1 is unexpectedly running after having␣
↪been stopped!")

        aux2.stop()    # chan1 attached to aux1 and aux2 is now closed as aux1 and aux2␣
↪are both stopped

        self.assertFalse(aux1.is_instance, "aux1 is unexpectedly running!")
        self.assertFalse(aux1.is_instance, "aux2 is unexpectedly running!")
```

4.1.9 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 communication channel that doesn't have any tracing capability (in contrast to `CCPcanCan` or `CCRttSegger` for example).

Everything is handled at configuration level within the YAML test configuration file:

```
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
```

(continues on next page)

(continued from previous page)

```
    trace_name: can_trace
  type: pykiso.lib.auxiliaries.proxy_auxiliary:ProxyAuxiliary
```

Note: This feature is only available with an explicit proxy definition as shown *above*.

4.2 How to make the most of the tests

4.2.1 Define the test information (in addition)

Assign test requirements to test cases

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 linked to the test that need to be defined as follow:

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

These test IDs will then be added to the generated XML or HTML report in order to be linked to general test requirements.

Filter the test cases to run with tags

In order to run only a subset of tests, an additional reference can be added to the test_run decorator:

- `tag`: optional dictionary of tag name-value pairs defined as:

```
{"variant": ["variant2", "variant1"], "branch_level": ["daily"]}
```

Note that these tag names are only examples. `Pykiso` accepts any tag name, as long as their name does not collide with default command line option names.

Based on the tags you provide as CLI options, only the tests fulfilling the following conditions will be run:

- tests that don't define any tag
- tests that define all provided tag names with at least one matching tag value for each of the tag names.

The only limitation is to always specify you tags as options in the CLI, i.e. as pairs of tag name and tag value. Multiple tag values for a single tag name simply need to be comma-separated (without whitespace).

Note: The cli will only allow you to use the character `-` instead of `_` for tags. If you call for example `--branch-level` in the cli, you can use following corresponding tags in your test case: `branch-level`, `branch_level` or `branchlevel`.

Find below an example of such a CLI invocation:

```
pykiso -c configuration_file --variant var1,var2 --branch-level daily,nightly
```

Table 1: Execution table for test case tags and cli tag arguments

CLI Tags	Test case Tags	Executed
none	any	
`--branch-level nightly`	`"branch-level": ["daily", "nightly"]`	
`--branch-level nightly,daily`	`"branch_level": ["daily"]`	
`--branch-level nightly,daily`	`"branch_level": ["daily", "nightly"]`	
`--branch-level other`	`"branch_level": ["daily", "nightly"]`	
`--branch-level daily --variant var1`	`"branch_level": ["daily", "nightly"]`	
`--branch-level daily`	`"variant": ["var1"]`	
`--variant var1`	`"branch_level": ["daily", "nightly"], "variant": ["var1"]`	
`--variant var2`	`"branch_level": ["daily", "nightly"], "variant": ["var1"]`	
`--branch-level nightly`	`"branch_level": ["daily", "nightly"], "variant": ["var1"]`	
`--branch-level daily --variant var1`	`"branch_level": ["daily", "nightly"], "variant": ["var1"]`	
`--branch-level daily --variant var42`	`"branch_level": ["daily", "nightly"], "variant": ["var1"]`	

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
"""
@pykiso.define_test_parameters(suite_id=1, aux_list=[aux1, aux2])
class SuiteSetup(pykiso.BasicTestSuiteSetup):
    def test_suite_setUp():
        logging.info("I HAVE RUN THE TEST SUITE SETUP!")
        if aux1.not_properly_configured():
            aux1.configure()
        aux2.configure()
        callback_registering()

"""
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
"""
@pykiso.define_test_parameters(suite_id=1, aux_list=[aux1, aux2])
class SuiteTearDown(pykiso.BasicTestSuiteTeardown):
    def test_suite_tearDown():
        logging.info("I HAVE RUN THE TEST SUITE TEARDOWN!")
        callback_unregistering()
```

(continues on next page)

(continued from previous page)

```
"""
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
-> test_ids: [optional] store the requirements into the report
-> tag: [optional] dictionary containing lists of variants and/or test levels when only␣
↪a subset of tests needs to be executed
"""
@pykiso.define_test_parameters(
        suite_id=1,
        case_id=1,
        aux_list=[aux1, aux2],
        test_ids={"Component1": ["Req1", "Req2"]},
        tag={"variant": ["variant2", "variant1"], "branch_level": ["daily", "nightly"]},
)
class MyTest(pykiso.BasicTest):
    pass
```

4.2.2 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 *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.

4.2.3 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 2 steps/methods `setUp()` and `tearDown()` 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** (in that case use RemoteTest instead of BasicTest)

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:
            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
```

(continues on next page)

(continued from previous page)

```
                aux.run_command(testcase_setup_basic_message, blocking=True, timeout_in_
↪s=10)
```

4.2.4 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
        # Check if the report for the reboot was received.
        if (
            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:
```

(continues on next page)

(continued from previous page)

```
                    # 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
```

4.2.5 Implementation of Advanced Tests - Continue a test case on a failed assertion

The Python unittest framework has a built-in solution in order to isolate assertions from the rest of the test: the subTest.

```
@pykiso.define_test_parameters(suite_id=1, case_id=1)
class MyTest1(pykiso.BasicTest):
    """Simple test case example that shows a possible usage of subTest."""

    def test_run(self):
        """
        In this test case we want to perform assertions that won't interrupt
        the execution when failing by using subTest.
        """
        with self.subTest("Verify that 1 == 2"):
            self.assertEqual(1, 2)

        logging.info("The rest of the test case will still be executed")
        self.assertTrue(False)
```

4.2.6 Implementation of Advanced Tests - Repeat testCases

pykiso.**retry_test_case**(*max_try=2*, *rerun_setup=False*, *rerun_teardown=False*, *stability_test=False*)

Decorator: retry mechanism for testCase.

The aim is to cover the 2 following cases:

- Unstable test : get the test pass within the {max_try} attempt
- Stability test : run {max_try} time the test expecting no error

The **retry_test_case** comes with the possibility to re-run the setUp and tearDown methods automatically.

Parameters

- **max_try** (int) – maximum number of try to get the test pass.
- **rerun_setup** (bool) – call the "setUp" method of the test.
- **rerun_teardown** (bool) – call the "tearDown" method of the test.

- **stability_test** (bool) – run {max_try} time the test and raise an exception if an error occurs.

Returns
None, a testCase is not supposed to return anything.

Raises
Exception – if stability_test, the exception that occurred during the execution; if not stability_test, the exception that occurred at the last try.

test_suite_1.py:

```
# define an external iterator that can be used for retry_test_case demo
side_effect = cycle([False, False, True])

@pykiso.define_test_parameters()
class MyTest1(pykiso.BasicTest):
    """This test case definition will override the setUp, test_run and tearDown method."
↪"

    @pykiso.retry_test_case(max_try=3)
    def setUp(self):
        """Hook method from unittest in order to execute code before test case run.

        In this case the default setUp method is implemented, allowing us to apply the
        retry_test_case's decorator.
        """
        pass

    @pykiso.retry_test_case(max_try=5, rerun_setup=True, rerun_teardown=False)
    def test_run(self):
        """In this case the default test_run method is overridden and
        instead of calling test_run from BasicTest class the following
        code is called.

        Here, the test pass at the 3rd attempt out of 5. The setup and
        tearDown methods are called for each attempt.
        """
        self.assertTrue(next(side_effect))
        logging.info(f"I HAVE RUN 0.1.1 for variant {self.variant}!")

    @pykiso.retry_test_case(max_try=3, stability_test=True)
    def tearDown(self):
        """Hook method from unittest in order to execute code after the test case ran.

        In this case the default tearDown method is implemented, allowing us to apply the
        retry_test_case's decorator.

        The retry_test_case has stability test activated, so the tearDown method will
        be run 3 times.
        """
        super().tearDown()
```

4.2.7 Test verbosity

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

To let the user decide which information they want to see in their logs, new log levels have been defined. When launched normally, only the logs from the tests and the framework errors will be active. The option -v (–verbose) should be used to display the internal logs of the framework:

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

or

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

Three internal log levels are available: INTERNAL_INFO, INTERNAL_DEBUG, INTERNAL_WARNING. They will then be activated depending on the value of the–log-level option. Error logs level will always be logged, internal or not.

The summary of the activated logs depending of the value of the –log-level and –verbose options can be found in the following table:

	verbose == True	verbose == False
log-level == DEBUG	DEBUG, INTERNAL_DEBUG, INFO, INTERNAL_INFO, WARNING, INTERNAL_WARNING, ERROR	DEBUG, INFO, WARNING, ERROR
log-level == INFO	INFO, INTERNAL_INFO, WARNING, INTERNAL_WARNING, ERROR	INFO, WARNING, ERROR
log-level == WARNING	WARNING, INTERNAL_WARNING, ERROR	WARNING, ERROR
log-level == ERROR	ERROR	ERROR

4.2.8 Run single tests

Test case can be selected with the -p or –pattern flag. Here is an example to just override the test file:

```
pykiso -c dummy.yaml -p test_suite_1.py
```

It is also possible to select single or multiple test cases by extending the pattern. Test classes and single test methods can be selected. The pattern can consist 3 elements separated by a "::". Each element is a unix file name pattern.

The elements are file_name::test_class_name::test_method_name

Here some examples:

```
#select a single test
pykiso -c dummy.yaml -p test_suite_1.py::TestClass::test_run1

#select all test methods which begins with test_
pykiso -c dummy.yaml -p test_suite_1.py::TestClass::test_*

#select all test classes which starts with Test and run method test_run1
pykiso -c dummy.yaml -p test_suite_1.py::Test*::test_run1

#use file pattern from yaml file and select all test classes and run method test_run1
pykiso -c dummy.yaml -p ::*::test_run1
```

4.3 Access framework's configuration

All parameters given at CLI and yaml level are available for each test cases/suites. For a convenient usage, all configuration information are class based represented. This means each parameter is accessible like a "normal" instance attribute (dot-access) and the attribute's name is simply the one given in the yaml or the CLI level.

Each parameter stored in GlobalConfig's yaml and CLI attributes is read-only.

> **Warning:** assign a new value will automatically raise an AttributeError

Let's admit we have the following yaml configuration file:

```yaml
auxiliaries:
  aux1:
    connectors:
        com: chan1
    config: null
    type: pykiso.lib.auxiliaries.dut_auxiliary:DUTAuxiliary
  aux2:
    connectors:
        com:   chan2
        flash: chan3
    type: pykiso.lib.auxiliaries.dut_auxiliary:DUTAuxiliary
  aux3:
    connectors:
        com:   chan4
    type: pykiso.lib.auxiliaries.dut_auxiliary:DUTAuxiliary
connectors:
  chan1:
    config:
      param_1: "value 1"
      param_2: 2000
    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: ~
    type: pykiso.lib.connectors.cc_flasher_example:FlasherExample
test_suite_list:
- suite_dir: conf_access
  test_filter_pattern: '*.py'
  test_suite_id: 1
```

And we passed the following arguments at the command line interface:

```
pykiso -c examples/conf_access.yaml --variant variant1 --variant daily --log-level INFO --verbose
```

To access all those parameters contain in both sources (cli and yaml) :

```
##########################################################################
# Copyright (c) 2010-2022 Robert Bosch GmbH
# This program and the accompanying materials are made available under the
# terms of the Eclipse Public License 2.0 which is available at
# http://www.eclipse.org/legal/epl-2.0.
#
# SPDX-License-Identifier: EPL-2.0
##########################################################################

"""
Configuration access example
****************************

:module: test_access

:synopsis: just a basic example on how to access configuration
    information fron test case level
"""

import logging

import pykiso
from pykiso.auxiliaries import aux1, aux2, aux3
from pykiso.global_config import GlobalConfig

# get all parameters given at yaml configuration level
yaml_config = GlobalConfig().yaml
# store all auxiliaries configuration
aux_config = yaml_config.auxiliaries
# store all connectors configuration
con_config = yaml_config.connectors
# get all parameters given at cli level
cli_config = GlobalConfig().cli


@pykiso.define_test_parameters(
    suite_id=1,
    case_id=1,
    aux_list=[aux1, aux2],
    setup_timeout=1,
    teardown_timeout=1,
    tag={"variant": ["variant2", "variant1"], "branch_level": ["daily", "nightly"]},
)
class MyTest1(pykiso.BasicTest):
    """Simply Test case use to show configuraton parameters access."""

    def setUp(self):
        """Just print all given cli parameters."""
        logging.info(
            f"--------------- SETUP: {self.test_suite_id}, {self.test_case_id} ----------
↪-----"
        )
        logging.info("*** print all parameters given at cli level ***")
```

(continues on next page)

(continued from previous page)

```
        logging.info(f"loaded configuration file: {cli_config.test_configuration_file}")
        logging.info(f"logging text file path: {cli_config.log_path}")
        logging.info(f"log level: {cli_config.log_level}")
        logging.info(f"report type: {cli_config.report_type}")
        logging.info(f"variant filter: {cli_config.variant}")
        logging.info(f"branch level: {cli_config.branch_level}")
        logging.info(f"pattern: {cli_config.pattern}")

    def test_run(self):
        """Just verify some configuration values."""
        logging.info(
            f"--------------- RUN: {self.test_suite_id}, {self.test_case_id} ------------
↪---"
        )
        if yaml_config.auxiliaries.aux2.connectors.flash == "chan3":
            logging.info("Auxiliary aux2 has a flasher")

        # just make a simple assertion in order to raise an error and
        # stop test execution if a specific value is not given to chan1
        # connector
        self.assertEqual(yaml_config.connectors.chan1.config.param_1, "value 1")

    def tearDown(self):
        """Just print aux2 configuration and it related connector too."""
        logging.info(
            f"---------------TEARDOWN: {self.test_suite_id}, {self.test_case_id} --------
↪-------"
        )
        logging.info("*** print aux2 configuration ***")
        logging.info(f"auxiliary aux2 flasher: {aux_config.aux2.connectors.flash}")
        logging.info(f"auxiliary aux2 connector: {aux_config.aux2.connectors.com}")

        logging.info("*** print associated connector configuration ***")
        logging.info(
            f"channel chan1 param 1: {yaml_config.connectors.chan1.config.param_1}"
        )
        logging.info(
            f"channel chan1 param 2: {yaml_config.connectors.chan1.config.param_2}"
        )
        logging.info(f"channel chan1 type: {yaml_config.connectors.chan1.type}")
        logging.info(f"connector chan2 type: {yaml_config.connectors.chan2.type}")
```

Note: the GlobalConfig class is a singleton, so one and only one instance is created during the whole execution time

4.4 Dynamic Configuration

In some situation, it can be useful to change the behaviour of the auxiliary in-use dynamically. For example, switching for a brand new channel or simply change an attribute value.

Thanks to the common auxiliary interface, users can easily change their auxiliary configuration by simply stop it (call of delete_instance public method) ,access it different public attributes, and then just restarts the auxiliary (call of the public method `delete_instance`)

> **Warning:** if you are using the original auxiliary instance don't forget to switch back to its initial configuration for the next test cases.

Find below a complete example where during the test, the current pcan connector is replaced by a simple CCLoopback:

```
import logging
import time

import pykiso

# as usual import your auxiliairies
from pykiso.auxiliaries import aux1, aux2, proxy_aux
from pykiso.lib.connectors.cc_raw_loopback import CCLoopback


@pykiso.define_test_parameters(
    suite_id=2,
    case_id=3,
    aux_list=[aux1, aux2],
)
class TestCaseOverride(pykiso.BasicTest):
    """In this test case we will simply use 2 communication auxiliaries
    bounded with a proxy one. The first communication auxiliary will be
    used for sending and the other one for the reception
    """

    def setUp(self):
        """If a fixture is not use just override it like below."""
        logging.info(
            f"--------------- SETUP: {self.test_suite_id}, {self.test_case_id} ----------
↪-----"
        )

        # start auxiliary one and two because I need it
        aux1.start()
        aux2.start()
        # start the proxy auxiliary in order to open the connector
        proxy_aux.start()

    def test_run(self):
        """Just send some raw bytes using aux1 and log first 100
        received messages using aux2.
        """
```

(continues on next page)

(continued from previous page)

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

        logging.info(f">> Send and receive message using the connected pcan <<")

        logging.info(f"send 300 messages using aux1/aux2")
        # just send some requests
        self._send_messages(300)
        # log the first 100 received messages, with the aux2
        self._receive_message(100)

        logging.info(
            f">> Change proxy_aux channel dynamically and continue to send/receive <<"
        )

        logging.info(f"Stop current running auxiliaries")
        self._stop_auxes()

        # save current channel used by the proxy
        self.pcan_channel = proxy_aux.channel
        # change proxy attached channel to CCLoopback
        proxy_aux.channel = CCLoopback()

        logging.info(f"Restart all auxiliaries")
        self._start_auxes()

        logging.info(f">> Send and receive message using the connected CCLoopback <<")

        logging.info(f"send 30 messages using aux1/aux2")
        # just send some requests
        self._send_messages(10)
        # log the first 10 received messages, with the aux2
        self._receive_message(10)

        logging.info(
            f">> Switch back to the pcan channel and continue to send/receive <<"
        )

        logging.info(f"Stop current running auxiliaries")
        self._stop_auxes()

        # switch back with pcan connector
        proxy_aux.channel = self.pcan_channel

        logging.info(f"Restart all auxiliaries")
        self._start_auxes()

        logging.info(f">> Send and receive message using the connected initial pcan <<")

        logging.info(f"send 30 messages using aux1/aux2")
```

(continues on next page)

(continued from previous page)

```
        # just send some requests
        self._send_messages(10)
        # log the first 10 received messages, with the aux2
        self._receive_message(10)

    def _stop_auxes(self) -> None:
        """Stop all auxiliaries currently in use."""
        # always stop the proxy auxiliary at the end
        aux1.delete_instance()
        aux2.delete_instance()
        proxy_aux.delete_instance()

    def _start_auxes(self) -> None:
        """Start all configured auxiliaries."""
        # always start the proxy auxiliary at the end
        aux1.create_instance()
        aux2.create_instance()
        proxy_aux.create_instance()

    def _send_messages(self, nb_msg: int) -> None:
        """Send n messages a defined number of times.

        :param nb_msg: number of messages pack to send
        """
        for _ in range(nb_msg):
            # send random messages using aux1
            aux1.send_message(b"\x01\x02\x03")
            aux2.send_message(b"\x04\x05\x06")
            aux1.send_message(b"\x07\x08\x09")

    def _receive_message(self, nb_msg: int) -> None:
        """Get messages from the reception queue.

        :param nb_msg: number of messages to dequeue
        """
        for _ in range(nb_msg):
            logging.info(f"received message: {aux2.receive_message()}")

    def tearDown(self):
        """If a fixture is not use just override it like below."""
        logging.info(
            f"--------------- TEARDOWN: {self.test_suite_id}, {self.test_case_id} -------
↪--------"
        )
```

Warning: this feature allows to change the complete auxiliary configuration, so depending on which parameters are changed the auxiliary execution could lead to unexpected behaviors.

4.5 Remote Test

With the remote test approach, the idea is to execute tests on the targeted hardware to enable the developer to practice test-driven-development directly on the target.

4.5.1 Test Coordinator

In the case of remote tests usage, the **test-coordinator** will still perform the same task but will also:

- verify if the tests can be performed
- flash and run and synchronize the tests on the *device under test*

4.5.2 Auxiliary

For the remote test approach, auxiliaries should be 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*)

One example of implementation of such an auxiliary is the *device under test* auxiliary used with the TestApp. In this specific case we have:

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

4.5.3 Connector

Communication Channel

In case of the *device under test*, we have a specific communication protocol. Please see the next paragraph.

Flasher

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

4.5.4 Message Protocol

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|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 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

4.5.5 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 flasher connector, which has to be configured with the correct binary file. The flasher connector is in turn called from an appropriate auxiliary (usually in its setup phase).

4.5.6 Implementation of Remote Tests

For remote tests, RemoteTestCase / RemoteTestSuite should be used instead of BasicTestCase / BasicTestSuite, 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.

Find below a full example for a test suite/case declaration in case the Message Protocol / TestApp is used:

```
"""
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.RemoteTestSuiteSetup):
      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.RemoteTestSuiteTeardown):
      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
-> tag: [optional] dictionary containing lists of variants and/or test levels when only␣
↪a subset of tests needs to be executed
"""
  @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"]},
          tag={"variant": ["variant2", "variant1"], "branch_level": ["daily", "nightly"]}
↪,
  )
  class MyTest(pykiso.RemoteTest):
```

(continues on next page)

(continued from previous page)

```
        pass
```

4.5.7 Config File for remote tests

For details see *How to make the most of the config file*.

Find below an example of config for used for remote testing (is that case using *device under test* auxiliary)

```yaml
auxiliaries:
  fdx_aux:
    connectors:
      com: fdx_channel
      flash: flash_lauterbach
    config: null
    type: pykiso.lib.auxiliaries.dut_auxiliary:DUTAuxiliary
connectors:
  #  Every path can be set as absolute or relative to the yaml file
  #  Todo: Fix the cmm dependencies in order to not be compulsory to run the tests from␣
↪the cmm's folder
  fdx_channel:
    config:
      t32_exc_path: 'Path/to/Trace32.exe'
      t32_config: '../path/to/config.t32'
      t32_main_script_path: '../path/to/TAPP_Demo.cmm'
      t32_reset_script_path: '../path/to/reset.cmm'
      t32_api_path: 'path/to/trace32.dll'
      port: '20000'
      node: 'localhost'
      packlen: '1024'
      device: 1
    type: pykiso.lib.connectors.cc_fdx_lauterbach:CCFdxLauterbach
  flash_lauterbach:
    config:
      t32_exc_path: 'Path/to/Trace32.exe'
      t32_config: '../path/to/config.t32'
      t32_main_script_path: '../path/to/TAPP_Demo.cmm'
      t32_reset_script_path: '../path/to/reset.cmm'
      t32_api_path: 'path/to/trace32.dll'
      port: '20000'
      node: 'localhost'
      packlen: '1024'
      device: 1
    type: pykiso.lib.connectors.flash_lauterbach:LauterbachFlasher

test_suite_list:
- suite_dir: test_suite_fdx_lauterbach
  test_filter_pattern: 'test*.py'
  test_suite_id: 1
```

4.6 System-test (step) report

The step report aims to provide a more comprehensive test-report (adapted for system testers) by tracking each assertion that contains a message. It follows the following structure:

- test name
- test description
- date of execution
- elapsed time
- information gathered during test
- assertion detail: - value of the data_in - variable name of the data_in - expected value - message
- the report is presented as an HTML page

4.6.1 Usage Examples

```
def setUp(self):
    # Data to test
    device_on = True
    voltage = 3.8

    # Simple assert
    self.assertTrue(device_on, msg="Check my device is ready")

    with self.subTest("Non critical checks"):
        # This check will fail but the test continues
        self.assertFalse(device_on, msg="Some check")

    # Add a new table in the report
    self.step_report.current_table = "Another table"

    # Assert with custom message
    # Assert msg overwritten when step_report_message not null
    self.step_report.message = "Custom message"
    self.assertAlmostEqual(voltage, 4, delta=1, msg="Check voltage device")

    # Additional data to include in the step-report
    self.step_report.header["Version_device"] = "2022-1234"
```

“self.step_report.header” allows you to store data data during test

4.6.2 How to generate

```
pykiso -c my_config.yaml --step-report my_report.html
```

4.6.3 Limitations

The step report generator might fail if you put parentheses or "assert" strings in strings or comments in the assert statement.

4.6.4 HTML result example

ITF Test Report for: StepReportTest-1-1 -> [
Fail
]

Test Description:

In this case the default test_run method is overridden and
instead of calling test_run from RemoteTest class the following
code is called.
Here, the test pass at the 3rd attempt out of 5. The setup and
tearDown methods are called for each attempt.

Date, Time, Software versions:

Start Time: 04/11/22 12:09:45
End Time: 04/11/22 12:09:45
Elapsed Time: 0.01
ITF version: 0.19.3
Version_device: 2022-1234

setUp

STEP	MESSAGE	VAR_NAME	EXPECTED_RESULT	ACTUAL_RESULT
1	Check my device is ready	True	True	True

test_run

STEP	MESSAGE	VAR_NAME	EXPECTED_RESULT	ACTUAL_RESULT
1	Check my device is ready	device_on	True	True
2	Some check	device_on	False	True
3	Custom message	voltage	Almost Equal to 4; with delta=1	3.8
4	Check voltage device	voltage	Almost Equal to 4; with delta=1	3.8

ITF Test Report for: TableTest-1-2 -> [
Fail
]

Test Description:

In this test we specify custom tables to group the steps.

Date, Time, Software versions:

Start Time: 04/11/22 12:09:45
End Time: 04/11/22 12:09:45
Elapsed Time: 0.01
ITF version: 0.19.3
Version_device: 2022-1234

First table

STEP	MESSAGE	VAR_NAME	EXPECTED_RESULT	ACTUAL_RESULT
1	Check my device is ready	device_on	True	True

Another table

STEP	MESSAGE	VAR_NAME	EXPECTED_RESULT	ACTUAL_RESULT
1	Some check	device_on	False	True

Last table

STEP	MESSAGE	VAR_NAME	EXPECTED_RESULT	ACTUAL_RESULT
1	Custom message	voltage	Almost Equal to 4; with delta=1	3.8

4.7 Multiprocessing

4.7.1 Introduction

In addition to the auxiliary's thread based implementation, the multiprocessing approach is possible too. A dedicated multiprocessing auxiliary interface is available and has the same capabilities/methods as the thread based interface.

Note: all examples are under examples/templates/mp_proxy_aux.yaml

4.7.2 Basic Users

For the moment, only the proxy auxiliary and proxy channel have their own multiprocessing version. The usage of those components only require to manipulate the flag newly created flag "processing" at connector configuration level as follow :

```
#_____________________________ Auxiliaries section _____________________________
# The multiprocessing proxy auxiliary has exactly the same interface,
# methods, or features as the thread based one. In addition, exactly
# the same configuration keywords are available.
auxiliaries:
  proxy_aux:
    connectors:
        com: can_channel
    config:
      aux_list : [aux1, aux2]
      activate_trace : True
      trace_dir: ./suite_mp_proxy
      trace_name: can_trace
      activate_log :
        - pykiso.lib.auxiliaries.mp_proxy_auxiliary
    type: pykiso.lib.auxiliaries.mp_proxy_auxiliary:MpProxyAuxiliary
  aux1:
    connectors:
        com: proxy_com1
    type: pykiso.lib.auxiliaries.communication_auxiliary:CommunicationAuxiliary
  aux2:
    connectors:
        com: proxy_com2
    type: pykiso.lib.auxiliaries.communication_auxiliary:CommunicationAuxiliary

#_____________________________ Connectors section _____________________________
connectors:
  proxy_com1:
    config:
      # when using mulitprocessing auxiliary flag processing has to True
      processing : True
    type: pykiso.lib.connectors.cc_mp_proxy:CCMpProxy
  proxy_com2:
    config:
      # when using mulitprocessing auxiliary flag processing has to True
```

(continues on next page)

(continued from previous page)

```
      processing : True
    type: pykiso.lib.connectors.cc_mp_proxy:CCMpProxy
  can_channel:
    config:
      # when using mulitprocessing auxiliary flag processing has to True
      processing : True
      interface : 'pcan'
      channel: 'PCAN_USBBUS1'
      state: 'ACTIVE'
      remote_id : 0x300
    type: pykiso.lib.connectors.cc_pcan_can:CCPCanCan
#_____________________________ Test Suite section _____________________________
test_suite_list:
- suite_dir: suite_proc_proxy
  test_filter_pattern: 'test_*.py'
  test_suite_id: 2
```

4.7.3 Advanced Users

As said before, the approach changes but the interface usage stays the same. Advanced user will not be disorientated, all methods are there. They were just adapted regarding multiprocessing pros and cons:

- lock_it
- unlock_it
- create_instance
- delete_instance
- run_command
- abort_command
- wait_and_get_report
- stop
- resume
- suspend

And inherit from the MpAuxiliaryInterface forces you to implement the following methods (as usual):

- _create_auxiliary_instance
- _delete_auxiliary_instance
- _run_command
- _abort_command
- _receive_message

So nothing really new !!

> **Warning:** note that using multiprocessing auxiliary may lead to an adaptation of your connector implementation or your external libraries.

4.7.4 Limitations

Junit report logging

Logging in junit report is not supported when using multiprocesssing version of the proxy auxiliary. This means no logs from proxy auxiliary and his associated connectors (except proxy channels) will be present in junit report.

logging on stdout

All logs coming from proxy's associated connectors (except proxy channels) won't be displayed on the console.

4.8 How to create an auxiliary

This tutorial aims to explain the working principle of an Auxiliary by providing information on the different Auxiliary interfaces, their purpose and the implementation of new auxiliaries.

To provide hints on the implementation of an Auxiliary, an example that implements a generic auxiliary is provided, that is not usable but shall explain the different concepts and implementation steps.

4.8.1 Different Auxiliary interfaces for different use-cases

Pykiso provides three different auxiliary interfaces, used as basis for any implemented auxiliary. These different interfaces aim to cover every possible usage of an auxiliary:

- The *AuxiliaryInterface* is a Thread-based auxiliary. It is suited for IO-bound tasks where the reception of data cannot be expected.
- The *MpAuxiliaryInterface* is a Process-based auxiliary. It is suited for CPU-bound tasks where the reception of data cannot be expected and its processing can be CPU-intensive. In contrary to the Thread-based auxiliary, this interface is not limited by the GIL and runs on all available CPU cores.
- The *SimpleAuxiliaryInterface* does not implement any kind of concurrent execution. It is suited for host-based applications where the auxiliary initiates every possible action, i.e. the reception of data can always be expected.
- The *DTAuxiliaryInterface* is a double threaded base auxiliary, where a thread is used for the transmission and a second one for the reception. It is suited for IO-bound tasks where the reception of data cannot be expected.

4.8.2 Execution of an Auxiliary

Auxiliary creation and deletion

Any auxiliary is **created** at test setup (before any test case is executed) by calling *create_instance()* and **deleted** at test teardown (after all test cases have been executed) by calling *delete_instance()*.

These methods set the `is_instance attribute` that indicated if the auxiliary is running correctly.

Concurrent auxiliary execution

The execution of concurrent auxiliaries (i.e. inheriting from *AuxiliaryInterface* or *MpAuxiliaryInterface*) is handled by the interfaces' *run()* method.

In the DTAuxiliaryInterface case, everything related to the tansmission is handled by `_transmit_task()` and for the reception by `_reception_task()`

Each command execution is handled in a thread-safe way by getting values from an input queue and returning the command result in an output queue.

Auxiliary run

Each time the execution is entered, the following actions are performed:

1. Verify if a request is available in the input queue

1. If the command message is "create_auxiliary_instance" and the auxiliary is not created yet, call the `_create_auxiliary_instance()` method and put a boolean corresponding to the success of the command processing in the output queue. This command message is put in the queue at test setup.
2. If the command message is "delete_auxiliary_instance" and the auxiliary is created, call the `_delete_auxiliary_instance()` method and put a boolean corresponding the success of the command processing in the output queue. This command message is put in the queue at test teardown.
3. If the command message is a tuple of 3 elements starting with "command", then a custom command has to be executed. This custom command has to be implemented in the `_run_command()` method.
4. If the command message is "abort" and the auxiliary is created, call the `_abort_command()` method and put a boolean corresponding the success of the command processing in the output queue.

2. Verify if a Message is available for reception

1. Call the auxiliarie's `_receive_message()` method
2. If something is returned, put it in the output queue, otherwise repeat this execution cycle.

For an auxiliary based on DTAuxiliaryInterface, the execution is slightly different due to the usage of two threads:

1. Verify if a request is available in the input queue

1. If the command message is "DELETE_AUXILIARY" the transmit task while loop ends
2. If the command message is a tuple of 2 elements starting with your custom command type, and then the data to send. This custom command has to be implemented in the `_run_command()` method.

2. Verify if a Message is available for reception

1. Call the auxiliarie's `_receive_message()` and simpy wait for a message coming from the connector.
2. If something is returned, put it in the output queue, otherwise repeat this execution cycle.

4.8.3 Implement an Auxiliary

Common auxiliary methods

All of the above described Auxiliary interfaces require the same abstract methods to be implemented:

- `_create_auxiliary_instance()`: handle the auxiliary creation. Minimal actions to perform are opening the attached *CChannel*, to which can be added actions such as flashing the device under test, perform security related operations to allow the communication, etc.
- `_delete_auxiliary_instance()`: handle the auxiliary deletion. This method is the counterpart of `_create_auxiliary_instance`, so it needs to be implemented in a way that `_create_auxiliary_instance` can be called again without side effects. In the most basic case, it should at least close the opened *CChannel*.

Concurrent auxiliary methods

In addition to the previously described methods, the concurrent Auxiliary interfaces *AuxiliaryInterface* and *MpAuxiliaryInterface* require the following methods to be implemented:

- `_run_command()`: implement the different commands that should be performed by the Auxiliary. The public API methods of an auxiliary should always call the thread-safe *run_command()* method with arguments corresponding to the command to run, which will in turn call this private method.
- `_abort_command()`: implement the command abortion mechanism. This mechanism **must also be implemented on the target device**. A valid implementation for the TestApp protocol can be found in `pykiso.lib.auxiliaries.dut_auxiliary.DUTAuxiliary._abort_command()`.
- `_receive_message()`: implement the reception of data. This method should at least call the CChannel's *cc_receive()* method. The received data can then be decoded according to a particular protocol, matched against a previously sent request, or trigger any kind of further processing.

For the concurrent Auxiliary interface *DTAuxiliaryInterface*, only one significant difference :

- `_abort_command()`: is not mandatory

Auxiliary implementation example

See below an example implementing the basic functionalities of a Thread Auxiliary:

```
import logging
from pykiso import AuxiliaryInterface, CChannel, Flasher

# this auxiliary is thread-based, so it must inherit AuxiliaryInterface
class MyAuxiliary(AuxiliaryInterface):

    def __init__(self, channel: CChannel, flasher: Flasher, **kwargs):
        """Initialize Auxiliary attributes.

        Any auxiliary must at least be initialised with a CChannel.
        If needed, a Flasher can also be attached.

        Any additional parameter can be added depending on the implementation.

        The additional kwargs contain the auxiliarie's alias and logger
        names to keep activated, all defined in the configuration file.
```

(continues on next page)

(continued from previous page)

```
        """
        super().__init__(**kwargs)
        self.channel = channel
        self.flasher = flasher

    def _create_auxiliary_instance(self):
        """Create the auxiliary instance at test setup.

        This method is also called when running self.resume()

        Simply flash the device under test with the attached Flasher instance
        and open the communication with the attached CChannel instance.
        """
        logging.info("Flash target")
        # used as context manager to close the flashing HW (debugger)
        # after successful flash
        with self.flash as flasher:
            flasher.flash()

        logging.info("Open communication")
        self.channel.open()

    def _delete_auxiliary_instance(self):
        """Delete the auxiliary instance at test teardown.

        This method is also called when running self.suspend()

        Simply end the communication by closing the attached CChannel instance.
        """
        logging.info("Close communication")
        self.channel.close()

    def send(self, to_send):
        """Send data without waiting for any response."""

        # self._run_command(("command", "send", to_send)) will be called internally
        return self.run_command("send", to_send, timeout_in_s=0)

    def send_raw_bytes(self, to_send):
        """Send raw data without waiting for any response."""

        # self._run_command(("command", "send", to_send)) will be called internally
        return self.run_command("send raw", to_send, timeout_in_s=0)

    def send_and_wait_for_response(self, to_send, timeout = 1):
        """Send data and wait for a response during `timeout` seconds."""

        # returns True if the command was successfully executed
        command_sent = self.run_command("send", to_send, timeout_in_s=0)

        if command_sent:
            # method of AuxiliaryCommon that tries to get an element from queue_out
```

(continues on next page)

(continued from previous page)

```
            # queue_out is populated by self._receive_message()
            return self.wait_and_get_report(timeout_in_s=timeout)

    def _run_command(self, cmd_message, cmd_data):
        """Command execution method that is called internally by the
        AuxiliaryInterface Thread.

        Each public API method should call this method with a command message
        and the data corresponding to the command.

        The command message is then matched against every possible implemented
        message and the corresponding action is performed in a thread-safe way.

        In this example, only a "send" command is implemented that will simply
        send the command data over the attached communication channel.
        """
        if cmd_message == "send":
            # in the CChannel implementation raw is set to False by default
            # the data to send is then pre-serialized according to the specified protocol
            return self.channel.send(cmd_data)
        elif cmd_message == "send raw":
            # set raw to True to send raw bytes through the CChannel
            return self.channel.send(cmd_data, raw=True)

    def _abort_command(self):
        """Command abortion method that is called by the AuxiliaryInterface Thread
        when calling `my_aux.abort_command()`.

        Assume that the device under test aborts the running command when receiving
        the data b'abort'.

        For the sake of simplicity, no further check will be performed on the successful
        reception of the data by the DUT (e.g. wait for an acknowledgement).
        """
        command_sent = self.send_raw_bytes(b'abort')
        return command_sent

    def _receive_message(self):
        """Reception method that is called internally by the AuxiliaryInterface Thread.

        Verify if there is 'raw' data to receive for 10ms and return it.
        """
        try:
            received_data = self.channel.cc_receive(timeout=0.01, raw=True)
            if received_data is not None:
                return received_data
        except Exception:
            logging.exception(f"Channel {self.channel} failed to receive data")
```

Regarding a concrete implementation using *DTAuxiliaryInterface* take a look to *CommunicationAuxiliary* source code.

More examples are available under `pykiso.lib.auxiliaries`.

Note: If the created auxiliary should be based on multiprocessing instead of threading, only the base class needs to be changed from `AuxiliaryInterface` to `MpAuxiliaryInterface`. The actual implementation does not need any adaptation.

Auxiliary without connector

If by default all auxiliaries require a connector, in some specific cases, it may complicate the total implementation. Therefor *connector_required* parameter was defined.

Note: Auxiliaries that should fall into this category will need to be discussed case by case.

Warning: Auxiliaries entering this category will raise an error if a connector is indeed assigned to it in the .yaml. Hybrid cases do not exist.

See below an example of an auxiliary without connector:

```
class ExampleAuxiliary(DTAuxiliaryInterface):
  """Example auxiliary without a connector"""

  def __init__(self) -> None:
    """Initialize the auxiliary"""
    super().__init__(connector_required=False)
```

See below for an example of its yaml config file:

```
auxiliaries:
  aux1:
    config: null
    type: pykiso.lib.auxiliaries.example_auxiliary:ExampleAuxiliary

test_suite_list:
- suite_dir: test_suite_1
  test_filter_pattern: '*.py'
  test_suite_id: 1
```

4.9 How to create a connector

On pykiso view, a connector is the communication medium between the auxiliary and the device under test. It can consist of two different blocks:

- a **communication channel**
- a **flasher**

Thus, its goal is to implement the sending and reception of data on the lower level protocol layers (e.g. CAN, UART, UDP).

Both can be used as context managers as they inherit the common interface *Connector*.

Note: Many already implemented connectors are available under `pykiso.lib.connectors`. and can easily be adapted for new implementations.

4.9.1 Communication Channel

In order to facilitate the implementation of a communication channel and to ensure the compatibility with different auxiliaries, pykiso provides a common interface *CChannel*.

This interface enforces the implementation of the following methods:

- *_cc_open()*: open the communication. Does not take any argument.
- *_cc_close()*: close the communication. Does not take any argument.
- *_cc_send()*: send data if the communication is open. Requires one positional argument `msg` .
- *_cc_receive()*: receive data if the communication is open. Requires one positional argument `timeout` .

Class definition and instanciation

To create a new communication channel, the first step is to define its class and constructor.

Let's admin that the following code is added to a file called **my_connector.py**:

```
import pykiso

MyCommunicationChannel(pykiso.CChannel):

    def __init__(self, arg1, arg2, kwarg1 = "default"):
        ...
```

Then, if this CChannel has to be used within a test, the test configuration file will derive from its location and constructor parameters:

```
connectors:
  my_chan:
    # provide the constructor parameters
    config:
      # arg1 and arg2 are mandatory as we defined them as positional arguments
      arg1: value for positional argument arg1
      arg2: value for positional argument arg2
      # kwarg1 is optional as we defined it as a keyword argument with a default value
      kwarg1: different value for keyword argument kwarg1
    # let pykiso know which class we want to instantiate with the provided parameters
    type: path/to/my_connector.py:MyCommunicationChannel
```

Note: If this file is located inside an installable package `my_package`, the type will become `type: my_package.my_connector:MyCommunicationChannel`.

Interface completion

If the code above is left as such, it won't be usable as a connector as the communication channel's abstract methods aren't implemented.

Therefore, all four methods `_cc_open`, `_cc_close`, `_cc_send` and `_cc_receive` need to be implemented.

In order to complete the code above, let's assume that a module *my_connection_module* implements the communication logic.

The connector then becomes:

```
from my_connection_module import Connection
import pykiso

MyCommunicationChannel(pykiso.CChannel):

    def __init__(self, arg1, arg2, kwarg1 = "default"):
        # Connection class could be anything, like serial.Serial or socket.socket
        self.my_connection = Connection(arg1, arg2)

    def _cc_open(self):
        self.my_connection.open()

    def _cc_close(self):
        self.my_connection.close()

    def _cc_send(self, data: bytes):
        self.my_connection.send(data_bytes)

    def _cc_receive(self, timeout) -> Optional[bytes]:
        received_data = self.my_connection.receive(timeout=timeout)
        if received_data:
            return received_data
```

Note: The API used in this example for the fictive *my_connection* module entirely depends on the used module.

Parameters and return values

In order to stay compatible and usable by the attached auxiliary, the created connector has to respect certain rules (in addition to the CChannel base class interface):

- **rule 1** : _cc_receive concrete implementation has to return a dictionary containing at least the key "msg". If more than the data received is return, for example the CAN ID from the emitter, just put it in the return dictionary.

see the example below with the cc_pcan_can connector and the return of the remote_id value :

```
def _cc_receive(
    self, timeout: float = 0.0001
) -> Dict[str, Union[MessageType, int]]:
    """Receive a can message using configured filters.

    :param timeout: timeout applied on reception
```

(continues on next page)

(continued from previous page)

```
    :return: the received data and the source can id
    """
    try:  # Catch bus errors & rcv.data errors when no messages where received
        received_msg = self.bus.recv(timeout=timeout or self.timeout)

        if received_msg is not None:
            frame_id = received_msg.arbitration_id
            payload = received_msg.data
            timestamp = received_msg.timestamp
            log.internal_debug(f"received CAN Message: {frame_id}, {payload}, {timestamp}
↪")
            return {"msg": payload, "remote_id": frame_id}
        else:
            return {"msg": None}
    except can.CanError as can_error:
        log.internal_debug(f"encountered can error: {can_error}")
        return {"msg": None}
    except Exception:
        log.exception(f"encountered error while receiving message via {self}")
        return {"msg": None}
```

- **rule 2** : additional arguments associated to _cc_send concret implementation has to be named arguments and used the **kwargs

see example below with the cc_pcan_can connector and the additional remote_id parameter:

```
def _cc_send(self, msg: MessageType, **kwargs) -> None:
    """Send a CAN message at the configured id.

    If remote_id parameter is not given take configured ones

    :param msg: data to send
    :param kwargs: named arguments

    """
    _data = msg
    remote_id = kwargs.get("remote_id")

    if remote_id is None:
        remote_id = self.remote_id
```

4.9.2 Flasher

pykiso provides a common interface for flashers *Flasher* that aims to be as simple as possible.

It only consists of 3 methods to implement:

- `open()`: open the communication with the flashing hardware if any (for e.g. JTAG flashing) and perform any preliminaly action
- *flash()*: perform all actions to flash the target device
- `close()`: close the communication with the flashing hardware.

Note: To ensure that a Flasher is closed after being opened, it should be used as a context manager (see *Auxiliary implementation example*).

4.10 How to change the class of the logger

4.10.1 Change logger class

You can change the class of the loggers used in pykiso with a custom logger class.

For example you have created a logger class (TestLogger) in the package called package_test in a file called logger_test. To use it you have pass the path to import the class as the value for –logger : package_test.test_logger.TestLogger

```
pykiso -c my_config.yaml --logger package_test.test_logger.TestLogger
```

4.10.2 Class with argument

If you have created a logger class that takes more argument than name and level to be initialized, you can specify them so that all loggers will be initialized with the same value.

See the following example :

```
class TestLogger(logging.Logger):
    def __init__(
        self, name: str, str_arg: str, int_arg: int, level=0
    ) -> None:
        super().__init__(name, level)
        self.str_arg = str_arg
        self.int_arg = int_arg
```

```
pykiso -c my_config.yaml --logger 'package_test.test_logger.TestLogger(str_arg="str_value
↪",int_arg=12)'
```

CHAPTER

FIVE

EXAMPLES

In *pykiso*, auxiliaries and connectors can be contributed. Some of them are more complex to use than others. Examples for some complex modules that need a more extensive documentation can be found here.

5.1 Controlling an acroname USB hub

The acroname auxiliary offers commands to control an acroname usb hub. Ports can be switched individually and their current and voltage can be measured. It is also possible to retrieve and set the current limitation of each port.

5.1.1 Usage Examples

To use the auxiliary in your test scripts the auxiliary must be properly defined in the config yaml. Example:

```
auxiliaries:
    acro_aux:
      config:
        # Serial number to connect to. Example: "0x66F4859B"
        serial_number : null # null = auto detection.
      type: pykiso.lib.auxiliaries.acroname_auxiliary:AcronameAuxiliary
```

Below find a example for the usage in a test script. All available methods are shown there. For convenience units can be selected via a string. For current methods use ‘uA’, ‘mA’ or ‘A’. For voltage methods use ‘uV’, ‘mV’ or ‘V’. If not specified ‘V’ or ‘A’ will be used.

```
##########################################################################
# Copyright (c) 2010-2022 Robert Bosch GmbH
# This program and the accompanying materials are made available under the
# terms of the Eclipse Public License 2.0 which is available at
# http://www.eclipse.org/legal/epl-2.0.
#
# SPDX-License-Identifier: EPL-2.0
##########################################################################

"""
Acroname auxiliary test
***********************

:module: test_acroname
```

(continues on next page)

(continued from previous page)

```
:synopsis: Example test that shows how to control the acroname usb.

.. currentmodule:: test_acroname

"""

import logging
import time

import pykiso
from pykiso.auxiliaries import acro_aux


@pykiso.define_test_parameters(suite_id=1, case_id=1, aux_list=[acro_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("Power off all usb port")

        for port_number in range(4):
            acro_aux.set_port_disable(port_number)

        time.sleep(2)

        logging.info("Power on all usb ports")

        for port_number in range(4):
            acro_aux.set_port_enable(port_number)

        time.sleep(2)

        logging.info("Set current limit on all usb ports to 1000 mA")

        for port_number in range(4):
            acro_aux.set_port_current_limit(port_number, 1000, "mA")

        logging.info("Take measurements on all usb ports")
        for port_number in range(4):
            voltage = acro_aux.get_port_voltage(port_number, "V")
            current = acro_aux.get_port_current(port_number, "mA")
            current_limit = acro_aux.get_port_current_limit(port_number, "mA")
```

(continues on next page)

(continued from previous page)

```
            logging.info(
                f"Measured {voltage:.3f}V {current:.3f}mA "
                f"currentlimit {current_limit:.3f}mA on usb Port {port_number}"
            )

        logging.info("Set current limit on all usb ports to maximum -> 2500 mA")
        for port_number in range(4):
            current_limit = acro_aux.set_port_current_limit(port_number, 2500, "mA")

    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} -------
↪--------"
        )
```

5.2 Controlling an Instrument

The instrument-control auxiliarie 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).

5.2.1 Requirements

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

5.2.2 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 provide an *instrument***
  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
```

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 unavailable. Please refer to your instrument's datasheet for details.
: 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 successfully changed to the desired value.
: 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} ----------
↪-----"
```

(continues on next page)

(continued from previous page)

```
        )

    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()}")

        # 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)}"
```

(continues on next page)

(continued from previous page)

```
        )
        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)}"
        )

    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} -------
↪--------"
        )
```

5.2.3 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.

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.

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*

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
```

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
```

General Command Overview

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

5.3 Passively record a channel

The record auxiliary can be used to utilize the logging mechanism from a connector. For example the realtime trace from the segger jlink can be recorded during a test run. The record auxiliary can also be used to save the log into a chosen file. It is also able to search for some specific message or regular expression (regex) into the current string or into a specified file/folder.

5.3.1 Usage Examples

To use the auxiliary in your test scripts the auxiliary must be properly defined in the config yaml. Example:

```yaml
auxiliaries:
  record_aux:
    connectors:
      com: rtt_channel
    config:
      # When is_active is set, it actively polls the connector. It demands if
      # the used connector needs to be polled actively.
      is_active: False # False because rtt_channel has its own receive thread
    type: pykiso.lib.auxiliaries.record_auxiliary:RecordAuxiliary

connectors:
  rtt_channel:
    config:
      chip_name: "STM12345678"
      speed: 4000
      block_address: 0x12345678
      verbose: True
      tx_buffer_idx: 1
      rx_buffer_idx: 1
      # Path relative to this yaml where the RTT logs should be written to.
      # Creates a file named rtt.log
      rtt_log_path: ./
      # RTT channel from where the RTT logs should be read
      rtt_log_buffer_idx: 0
      # Manage RTT log CPU impact by setting logger speed. eg: 100% CPU load
      # default: 1000 lines/s
      rtt_log_speed: null
    type: pykiso.lib.connectors.cc_rtt_segger:CCRttSegger

test_suite_list:
- suite_dir: test_record
  test_filter_pattern: '*.py'
  test_suite_id: 1
```

```yaml
auxiliaries:
  record_aux:
    connectors:
      com: example_channel
    config:
      com: CChannel
      is_active: True
      timeout: 0
      log_folder_path: "examples/test_record"
    type: pykiso.lib.auxiliaries.record_auxiliary:RecordAuxiliary

connectors:
  example_channel:
    config: null
    type: pykiso.lib.connectors.cc_raw_loopback:CCLoopback
```

(continues on next page)

(continued from previous page)

```
test_suite_list:
- suite_dir: test_record
  test_filter_pattern: test_recorder_example.py
  test_suite_id: 1
```

Below find a example for the usage in a test script. It is only necessary to import record auxiliary.

```
from pykiso.auxiliaries import record_aux
```

Example test script:

```
##########################################################################
# Copyright (c) 2010-2022 Robert Bosch GmbH
# This program and the accompanying materials are made available under the
# terms of the Eclipse Public License 2.0 which is available at
# http://www.eclipse.org/legal/epl-2.0.
#
# SPDX-License-Identifier: EPL-2.0
##########################################################################

"""
Record auxiliary test
*********************

:module: test_record

:synopsis: Example test that shows how to record a connector

.. currentmodule:: test_record

"""

import logging
import time

import pykiso

# !!! IMPORTANT !!!
# To start recording the channel which are specified in the yaml file,
# the record_aux must be first imported here.
# The channel recording will then run automatically in the background.
from pykiso.auxiliaries import record_aux


@pykiso.define_test_parameters(suite_id=1, case_id=1, aux_list=[])
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} ----------
↪----"
```

(continues on next page)

(continued from previous page)

```
        )

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

        logging.info(
            "Sleep 5 Seconds. Record specified channel from .yaml in the background."
        )
        time.sleep(5)

    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} -------
↪----"
        )
```

```
##############################################################################
# Copyright (c) 2010-2022 Robert Bosch GmbH
# This program and the accompanying materials are made available under the
# terms of the Eclipse Public License 2.0 which is available at
# http://www.eclipse.org/legal/epl-2.0.
#
# SPDX-License-Identifier: EPL-2.0
##############################################################################

"""
Record auxiliary test
*********************

:module: test_record

:synopsis: Example test that shows how to record a connector

.. currentmodule:: test_record

"""

import logging
import time

import pykiso
from pykiso.auxiliaries import record_aux

logging = logging.getLogger(__name__)


@pykiso.define_test_parameters(suite_id=1, case_id=1, aux_list=[])
class TestWithPowerSupply(pykiso.BasicTest):
```

(continues on next page)

(continued from previous page)

```
    def generate_new_log(self, msg: bytes):
        return record_aux.channel._cc_send(msg)

    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} ------------
↪----"
        )
        header = record_aux.new_log()
        self.assertEqual(header, "Received data :")

        self.generate_new_log(msg=b"log1")
        time.sleep(1)
        new_log = record_aux.new_log()
        self.assertEqual(new_log, "\nlog1")
        logging.info(new_log)

        self.generate_new_log(msg=b"log2")
        time.sleep(1)
        new_log = record_aux.new_log()
        self.assertEqual(new_log, "\nlog2")
        logging.info(new_log)

        logging.info(record_aux.get_data())

        # search regex
        logging.info(record_aux.search_regex_current_string(regex=r"log\d"))

        # clear data and check
        record_aux.clear_buffer()
        logging.info(record_aux.get_data())

        # create a file where it write recorded data. as log is empty, will not return␣
↪any file
        record_aux.dump_to_file(filename="record_example.txt")

        record_aux.stop_recording()

        logging.info("Sleep 1 Seconds to do something else with the channel.")
        time.sleep(1)

        record_aux.start_recording()
        time.sleep(1)  # Time for the channel to get opened
        header = record_aux.new_log()
```

(continues on next page)

(continued from previous page)

```
        self.assertEqual(header, "Received data :")

        self.generate_new_log(msg=b"log3")
        time.sleep(1)
        new_log = record_aux.new_log()
        self.assertEqual(new_log, "\nlog3")
        logging.info(new_log)

    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} -------
↪----"
        )
```

5.4 Using UDS protocol

UdsAuxiliary class (`pykiso.lib.auxiliaries.udsaux.uds_auxiliary.UdsAuxiliary`) contained in uds_auxiliary.py is the main interface between user and all the behind implemented logic. This class defines usable keywords(methods) for scripters in order to send uds requests to the device under test (raw or configurable)...

5.4.1 Configuration

To configure the UDS auxiliary 3 parameters are mandatory :

- odx_file_path: path to the odx formatted ecu diagnostic definition file.

Note: More information about yaml test configuration creation are available under Test Integration Framework project documentation.

Find below a complete configuration example :

```
auxiliaries:
  uds_aux:
    connectors:
        com: can_channel
    config:
      # you can specify your odx file by using odx_file_path parameter
      # and instead of using send_uds_raw method use the send_uds_config
      # for a more human readable command
      odx_file_path: null
      request_id : 0x123
      response_id : 0x321
      # uds_layer parameter is not mandatory and by default the following
      # values will be applied:
      # transport_protocol -> CAN
      # p2_can_client -> 5
      # p2_can_server -> 1
      uds_layer:
```

(continues on next page)

(continued from previous page)

```
        transport_protocol: 'CAN'
        # p2_can specifies receive time outs in seconds
        p2_can_client: 5
        p2_can_server: 1
      # tp_layer parameter is not mandatory and by default the following
      # values will be applied:
      # addressing_type -> NORMAL
      # n_sa -> 0xFF
      # n_ta -> 0xFF
      # n_ae -> 0xFF
      # m_type -> DIAGNOSTICS
      # discard_neg_resp -> False
      tp_layer:
        addressing_type: 'NORMAL'
        n_sa: 0xFF
        n_ta: 0xFF
        n_ae: 0xFF
        m_type: 'DIAGNOSTICS'
        discard_neg_resp: False
    type: pykiso.lib.auxiliaries.udsaux.uds_auxiliary:UdsAuxiliary
connectors:
  can_channel:
    config:
      interface : 'pcan'
      channel: 'PCAN_USBBUS1'
      state: 'ACTIVE'
    type: pykiso.lib.connectors.cc_pcan_can:CCPCanCan
test_suite_list:
- suite_dir: test_uds
  test_filter_pattern: 'test_raw_uds*.py'
  test_suite_id: 1
```

5.4.2 Send UDS Raw Request

Send UDS request as list of raw bytes.

The method send_uds_raw(`pykiso.lib.auxiliaries.udsaux.UdsAuxiliary.send_uds_raw()`) takes one mandatory parameter msg_to_send and one optional : timeout_in_s The parameter msg_to_send is simply the UDS request payload which is a list of bytes.

The optional parameter timeout_in_s (by default fixed to 5 seconds) simply represent the maximum amount of time in second to wait for a response from the device under test. If this timeout is reached, the uds-auxiliary stop to acquire and log an error.

The method send_uds_raw method returns a `UdsResponse` object, which is a subclass of UserList. UserList allow to keep property of the list, meanwhile attributes can be set, for UdsResponse, defined attributes refer to the positivity of the response, and its NRC if negative.

```
class UdsResponse(UserList):
    NEGATIVE_RESPONSE_SID = 0x7F

    def __init__(self, response_data) -> None:
```

(continues on next page)

(continued from previous page)

```
        super().__init__(response_data)
        self.is_negative = False
        self.nrc = None
        if self.data and self.data[0] == self.NEGATIVE_RESPONSE_SID:
            self.is_negative = True
            self.nrc = NegativeResponseCode(self.data[2])
```

Here is an example:

```
import pykiso
from pykiso.auxiliaries import uds_aux
from collections import UserList

@pykiso.define_test_parameters(suite_id=1, case_id=1, aux_list=[uds_aux])
class ExampleUdsTest(pykiso.BasicTest):
    def setUp(self):
        """Hook method from unittest in order to execute code before test case run.
        """
        pass

    def test_run(self):
        # Set extended session
        diag_session_response = uds_aux.send_uds_raw([0x10, 0x03])
        self.assertEqual(diag_session_response[:2], [0x50, 0x03])
        self.assertEqual(type(diag_session_response), UserList)
        self.assertFalse(diag_session_response.is_negative)

    def tearDown(self):
        """Hook method from unittest in order to execute code after test case run.
        """
        pass
```

5.4.3 Send UDS Config Request

Send UDS request as a configurable data dictionary. This method can be more practical for UDS requests with long payloads and a multitude of parameters. The method send_uds_config(`pykiso.lib.auxiliaries.udsaux.uds_auxiliary.UdsAuxiliary.send_uds_config()`) takes one mandatory parameter msg_to_send and an optional one timeout_in_s. The parameter msg_to_send is the UDS request defined as a configurable dictionary that always respects the below defined template:

Note: this feature is only available if a valid ODX file is given at auxiliary configuration level

```
req = {
    'service': %SERVICE_ID%,
    'data': %DATA%
}
```

SERVICE_ID -> SID (Service Identifier) of the UDS request either defined as a byte or the corresponding enum label:

```
class IsoServices(IntEnum):
    DiagnosticSessionControl = 0x10
    EcuReset = 0x11
    SecurityAccess = 0x27
    CommunicationControl = 0x28
    TesterPresent = 0x3E
    AccessTimingParameter = 0x83
    SecuredDataTransmission = 0x84
    ControlDTCSetting = 0x85
    ResponseOnEvent = 0x86
    LinkControl = 0x87
    ReadDataByIdentifier = 0x22
    ReadMemoryByAddress = 0x23
    ReadScalingDataByIdentifier = 0x24
    ReadDataByPeriodicIdentifier = 0x2A
    DynamicallyDefineDataIdentifier = 0x2C
    WriteDataByIdentifier = 0x2E
    WriteMemoryByAddress = 0x3D
    ClearDiagnosticInformation = 0x14
    ReadDTCInformation = 0x19
    InputOutputControlByIdentifier = 0x2F
    RoutineControl = 0x31
    RequestDownload = 0x34
    RequestUpload = 0x35
    TransferData = 0x36
    RequestTransferExit = 0x37
```

DATA -> dictionary that contains the following keys:

- 'parameter': DID (Data Identifier) of the UDS request. (In most UDS services with DID)
- %param_n%: one or many keys that represent the parameters related to the service, those depend on ODX definition that is tested.

See some examples of UDS requests below:

```
import pykiso
from pykiso.auxiliaries import uds_aux
from uds import IsoServices

@pykiso.define_test_parameters(suite_id=1, case_id=1, aux_list=[uds_aux])
class ExampleUdsTest(pykiso.BasicTest):
    def setUp(self):
        """Hook method from unittest in order to execute code before test case run.
        """
        pass

    def test_run(self):
            extendedSession_req = {
                "service": IsoServices.DiagnosticSessionControl,
                "data": {"parameter": "Extended Diagnostic Session"},
            }
            diag_session_response = uds_aux.send_uds_config(extendedSession_req)
```

(continues on next page)

(continued from previous page)

```
    def tearDown(self):
        """Hook method from unittest in order to execute code after test case run.
        """
        pass
```

The optional parameter timeout_in_s (by default fixed to 6 seconds) simply represents the maximum amount of time in second to wait for a response from the device under test. If this timeout is reached, the uds-auxiliary stops to acquire and log an error.

If the corresponding response is received from entity under test, send_uds_config method returns it also as a preconfigured dictionary. In case of a UDS positive response and no data to be returned, None is returned by the send_uds_config method. In case of a UDS negative response, a dictionary with the key 'NRC' is returned and the NRC value. Optionally, 'NRC_Label' may be returned if it is defined in ODX for the called service, containing the uds negative response description.

5.4.4 UDS Reset functions

Reset might be integrated in different tests.

The methods:

- soft_rest(`pykiso.lib.auxiliaries.udsaux.uds_auxiliary.UdsAuxiliary.soft_reset()`)
- hard_reset(`pykiso.lib.auxiliaries.udsaux.uds_auxiliary.UdsAuxiliary.hard_reset()`)
- force_ecu_reset(`udsaux.uds_auxiliary.UdsAuxiliary.force_ecu_reset()`)

do not take any argument, and regarding the config (with our without odx file) will send either raw message, or uds config (except for the key_off_on methods, but can remain acceptable for odx uds config)

```
Soft reset
uds_aux.soft_reset()
```

5.4.5 UDS check functions

Check functions might be integrated in different tests.

The methods:

- check_raw_response_negative(`pykiso.lib.auxiliaries.udsaux.uds_auxiliary.UdsAuxiliary.check_raw_response_negative()`)
- check_raw_response_positive(`pykiso.lib.auxiliaries.udsaux.uds_auxiliary.UdsAuxiliary.check_raw_response_positive()`)

The methods take as only mandatory argument the received response. The parameter rest is the response as a userlist object.

```
#Check raw response is positive
uds_aux.check_raw_response_positive(resp)

#Check raw response is negative
uds_aux.check_raw_response_negative(resp)
```

5.4.6 UDS read & write data

Read data(udsaux.uds_auxiliary.UdsAuxiliary.read_data()) and write(udsaux.uds_auxiliary.UdsAuxiliary.write_data()) are two helper API that use send_uds_config with specific ISO services (udsaux.uds_utils.UdsAuxiliary.read_data())

```
ReadDataByIdentifier = 0x22

WriteDataByIdentifier = 0x2E
```

Using write_data takes two arguments : parameter, and value. Parameter is simply a string that refer to the name of the data you want to modify, and value is simply the value you want to assign to the chosen parameters API must return None in case of positive response, and dictionary with NRC in it (for further information, check in send_uds_config documentation). Using this API is similar to do this :

```
req = {
    'service': IsoServices.WriteDataByIdentifier,
    'data': {'parameter': 'MyProduct', 'dataRecord': [('SuperProduct', '12345')]}
}

resp = uds_aux.send_uds_config(writeProductCode_req)
return resp
```

In the same way, read_data takes one argument : parameter.

Parameter is a string that contain the name of the data that is to be read. API must return dictionary with either data associated to the read parameter, or NRC.

5.4.7 UDS tester present sender

In order for any diagnostic session to be kept open, a tester presence frame has to be sent every 5 seconds. For this purpose, the tester present sender context manager can be used, it will send the tester present frame at the period given, allowing you to keep the session open for more than 5 seconds.

```
# start sending tester present messages every 3 seconds until the context manager is␣
↪exited
with uds_aux.tester_present_sender(period=3):
    # Perform uds commands here
```

It is also possible to start and stop the tester present sender manually with the methods start_tester_present_sender and stop_tester_present_sender.

```
# start sending tester present messages every 1 seconds until the context manager is␣
↪exited
uds_aux.start_tester_present_sender(period=1)
# Perform uds commands here
uds_aux.stop_tester_present_sender()
```

It is then possible to check if the tester present is active with the attribute is_tester_present

```
if uds_aux.is_tester_present:
    # Perform commands here
```

5.5 UDS protocol handling as a server

The `UdsServerAuxiliary` implements the Unified Diagnostic Services protocol on server side and therefore acts as an Electronic Control Unit communicating with a tester.

It allows the registration of callbacks through the helper class `UdsCallback` or simply by specifying the arguments of `register_callback()`, that are then triggered when the registered UDS request is received, allowing to respond with user-defined UDS messages.

5.5.1 Configuration

To configure the UDS server auxiliary 1 parameter is mandatory :

- `config_ini_path`: path to the UDS parameters configuration file (see format below).

It also accepts three optional parameters:

- **`request_id`: CAN identifier of the UDS responses send by the auxiliary**
  (overrides the one defined in the config.ini file)
- **`response_id`: CAN identifier of the UDS requests received by the auxiliary**
  (overrides the one defined in the config.ini file)
- `odx_file_path`: path to the ECU diagnostic definition file in ODX format

Note: To configure callbacks from a ODX file you need to use a different format for requests (see UdsCallback below).

Find below a complete configuration example :

```
auxiliaries:
    uds_aux:
        connectors:
            com: can_channel
        config:
            odx_file_path: ./path/to/my/file.odx
            # For Vector Box, serial number and interface needs to be updated in config.
↪ini file
            # request and response id need to be configured in config.ini if not␣
↪specified
            # by the request_id and response_id parameters
            config_ini_path: ./test_uds/config.ini
            # override the CAN IDs specified in the config.ini file
            request_id: 0x123
            response_id: 0x321
        type: pykiso.lib.auxiliaries.udsaux.uds_server_auxiliary:UdsServerAuxiliary
connectors:
    can_channel:
        config:
        interface: 'pcan'
        channel: 'PCAN_USBBUS1'
        state: 'ACTIVE'
        type: pykiso.lib.connectors.cc_pcan_can:CCPCanCan
test_suite_list:
-   suite_dir: ./test_uds
```

(continues on next page)

(continued from previous page)

```
    test_filter_pattern: 'test_uds_server.py'
    test_suite_id: 1
```

And for the config.ini file:

```
[can]
interface=peak
canfd=True
baudrate=500000
data_baudrate=2000000
defaultReqId=0xAC
defaultResId=0xDC

[uds]
transportProtocol=CAN
P2_CAN_Client=5
P2_CAN_Server=1

[canTp]
reqId=0xAC
resId=0xDC
addressingType=NORMAL
N_SA=0xFF
N_TA=0xFF
N_AE=0xFF
Mtype=DIAGNOSTICS
discardNegResp=False

[virtual]
interfaceName=virtualInterface

[peak]
device=PCAN_USBBUS1
f_clock_mhz=80
nom_brp=2
nom_tseg1=63
nom_tseg2=16
nom_sjw=16
data_brp=4
data_tseg1=7
data_tseg2=2
data_sjw=2

[vector]
channel=1
appName=MyApp

[socketcan]
channel=can0
```

5.5.2 Configuring UDS callbacks

In order to configure callbacks to be triggered on a received request, the `register_callback()` needs to be called.

The available parameters for defining a callback are the following:

- **`request` (mandatory): the incoming UDS request on which the corresponding callback should be executed.**
  The request can be passed as an integer (e.g. `0x1003` or as a list of integers `[0x10, 0x03]`). If the server has a ODX file specified it is possible to use ODX based callbacks with the following format:

```
odx_request = {
    "service": IsoServices.ReadDataByIdentifier,  # the SID
    "data": {
        "parameter": "sd_name"  # name of the data point in the <SD> element
    }
}
```

- **`response` (optional): the UDS response to send if the registered request is received.**
  Passed format is the same as for the request parameter. If the server has a ODX file specified, you can also use a ODX based format as response:

```
odx_response = {"sd_name": "your_data"}

# for a negative response, the following format is expected:
# the key needs to be the string "negative" (case insensitive)
# nrc is a uds negative response code as int
odx_response = {"Negative": nrc}
```

- **`response_data` (optional): the UDS data to send with the response. If the response is specified**
  the data is simply appended to the response. This parameter can be passed as an integer or as bytes (e.g. `b"DATA"`).
- **`data_length` (optional): the expected length of the data to send within the response, as an integer.**
  This parameter in only taken into account if the `response_data` parameter is specified and applied zero-padding to the response if the data to send is expected to have a fixed length.
- **`callback` (optional): a user-defined callback function to execute. If this parameter is provided,**
  all others optional parameters are discarded. The callback function must admit 2 positional arguments: the request on which the callback function is executed and the `UdsServerAuxiliary` instance that registered the callback.

Note: If the `response` parameter is not specified, the response will be built based on the `request` parameter. For example, a request `0x10020304` will produce the corresponding response `0x50020304`.

In order to define and register callbacks for a test, two ways are made possible:

- **With the helper class `UdsCallback`**
  in order to define the callbacks, and register them later.
- **With the method `register_callback()`**
  in order to define and register a callback at the same time.

Split definition and registration

The `UdsCallback` can be imported from directly from `pykiso.lib.udsaux` and allow an easy definition of callbacks that are common to multiple test cases.

It takes the same parameters as `register_callback()` but allows to define the callbacks in order to register them afterwards.

Pykiso also defined a callback subclass for the UDS data download functional unit that can be directly imported and re-used, or taken as a reference in order to implement other functional UDS units: `UdsDownloadCallback`.

Find below an example:

```
# helper objects to build callbacks can be imported from the pykiso lib
from pykiso.lib.auxiliaries.udsaux import UdsCallback, UdsDownloadCallback

# callbacks to register can then be built and stored in a list in order to be registered␣
↪in tests
UDS_CALLBACKS = [
    # Here the response could be left out
    # It would be automatically built based on the request
    UdsCallback(request=0x3E00, response=0x7E00),

    # The download functional unit is available as a pre-defined callback
    # It only requires the stmin parameter (minimum time between 2 consecutive frames,␣
↪here 10ms)
    # Others (RequestUpload, RequestFileTransfer) can be implemented based on it.
    UdsDownloadCallback(stmin=10),

    # define a callback for incoming read data by identifier request with identifier␣
↪[0x01, 0x02]
    # the response will be built by:
    # - creating the positive response corresponding to the request: 0x620102
    # - appending the passed response data b'DATA': 0x620102_44415451
    # - zero-padding the response data until the expected length is reached: 0x620102_
↪44415451_0000
    UdsCallback(request=0x220102, response_data=b'DATA', data_len=6)
]
```

Admitting that this code is added to a `uds_callback_definition.py` file at the same level as the test case, it can then be registered inside a test as follows:

```
import pykiso
from pykiso.auxiliaries import uds_aux

from uds_callback_definition import UDS_CALLBACKS

@pykiso.define_test_parameters(suite_id=1, case_id=1, aux_list=[uds_aux])
class ExampleUdsServerTest(pykiso.BasicTest):

    def setUp(self):
        """Register callbacks from an external file for the test."""

        for callback in UDS_CALLBACKS:
            uds_aux.register_callback(callback)
```

(continues on next page)

(continued from previous page)

```
    def test_run(self):
        """Actual test."""
        ...

    def tearDown(self):
        """Unregister all callbacks from the external file."""
        for callback in UDS_CALLBACKS:
            uds_aux.register_callback(callback)
```

In-test definition and registration

The method :py:meth:~pykiso.lib.auxiliaries.udsaux.uds_server_auxiliary.UdsServerAuxiliary.register_callback can be used inside a test case to define and register a callback with one line.

It admits the same parameters as :py:class:~pykiso.lib.auxiliaries.udsaux.common.uds_callback.UdsCallback and builds instances of it in the background.

Find below an example showing its usage, along with a custom callback function definition:

```
import typing

import pykiso
from pykiso.auxiliaries import uds_aux

# only used for type-hinting the custom callback
from pykiso.lib.auxiliaries.udsaux import UdsServerAuxiliary

def custom_callback(ecu_reset_request: typing.List[int], aux: UdsServerAuxiliary) ->␣
↪None:
    """Custom callback example for an ECU reset request.

    This simulates a pending response from the server before sending the
    corresponding positive response.

    :param ecu_reset_request: received ECU reset request from the client.
    :param aux: current UdsServerAuxiliary instance used in test.
    """
    for _ in range(4):
        aux.send_response([0x7F, 0x78])
        time.sleep(0.1)
    aux.send_response([0x51, 0x01])


@pykiso.define_test_parameters(suite_id=1, case_id=1, aux_list=[uds_aux])
class ExampleUdsServerTest(pykiso.BasicTest):

    def setUp(self):
        """Register various callbacks for the test."""
        # handle extended diagnostics session request
        # respond to an incoming request [0x10, 0x03] with [0x50, 0x03, 0x12, 0x34]
        uds_aux.register_callback(request=0x1003, response=0x50031234)
```

(continues on next page)

(continued from previous page)

```
        # handle incoming read data by identifier request with identifier [0x01, 0x02]
        # the response will be built by:
        # - creating the positive response corresponding to the request: 0x620102
        # - appending the passed response data b'DATA': 0x620102_44415451
        # - zero-padding the response data until the expected length is reached:␣
↪0x620102_44415451_0000
        uds_aux.register_callback(request=0x220102, response_data=b'DATA', data_length=6)

        # register the custom callback defined above
        uds_aux.register_callback(request=0x1101, callback=custom_callback)

    def test_run(self):
        """Actual test."""
        ...

    def tearDown(self):
        """Unregister all callbacks registered by the auxiliary."""

        for callback in uds_aux.callbacks:
            uds_aux.unregister_callback(callback)
```

If the `UdsServerAuxiliary` was configured with a valid ODX file, you can use ODX based configuration of callbacks like this (same format as above):

```
import typing

import pykiso
from pykiso.auxiliaries import uds_aux


@pykiso.define_test_parameters(suite_id=1, case_id=1, aux_list=[uds_aux])
class ExampleUdsServerTest(pykiso.BasicTest):

    def setUp(self):
        """Register various callbacks for the test."""
        # handle extended diagnostics session request
        # respond to an incoming request with default [0x50, 0x03]
        uds_aux.register_callback(
            request={
                "service": IsoServices.DiagnosticSessionControl,
                "data": {
                    "parameter": "Extended_DiagnosticSession"
                }
            }
        )
        odx_callback = UdsCallback(
            request={
                "service": IsoServices.ReadDataByIdentifier,
                "data": {
                    "parameter": "SoftwareVersion"
            },
```

(continues on next page)

(continued from previous page)

```
            response={"SoftwareVersion": "1.33.7"}
        )
        # handle incoming read data by identifier request with identifier parsed from␣
↪ODX,
        # e.g. [0x22, 0xA4, 0x55]
        # the response will be built by:
        # - creating the positive response corresponding to the request:␣
↪0x62A455312E33332E37
        uds_aux.register_callback(odx_callback)

    def test_run(self):
        """Actual test."""
        # can access the odx based callbacks with readable strings as well:
        read_software_version = uds_server_auxiliary.callbacks["ReadDataByIdentifier.
↪SoftwareVersion"]

    def tearDown(self):
        """Unregister all callbacks registered by the auxiliary."""

        for callback in uds_aux.callbacks:
            uds_aux.unregister_callback(callback)
```

5.5.3 Accessing UDS callbacks

Once registered, callbacks can be accessed inside a test via the `callbacks` attribute. This attribute is a dictionary linking the registered request as an **uppercase** hexadecimal string (e.g. `"0x2E0102"`) to the corresponding registered callback.

Accessing a callback can be useful for verifying if a callback was called at some point. Based on the test snippets above, the following complete test example aims to show this feature and provided an overview of all previously described features:

```
import typing

import pykiso
from pykiso.auxiliaries import uds_aux

# only used for type-hinting the custom callback
from pykiso.lib.auxiliaries.udsaux import UdsServerAuxiliary

from uds_callback_definition import UDS_CALLBACKS

def custom_callback(ecu_reset_request: typing.List[int], aux: UdsServerAuxiliary) ->␣
↪None:
    """Custom callback example for an ECU reset request.

    This simulates a pending response from the server before sending the
    corresponding positive response.

    :param ecu_reset_request: received ECU reset request from the client.
    :param aux: current UdsServerAuxiliary instance used in test.
```

(continues on next page)

(continued from previous page)

```
    """
    for _ in range(4):
        aux.send_response([0x7F, 0x78])
        time.sleep(0.1)
    aux.send_response([0x51, 0x01])


@pykiso.define_test_parameters(suite_id=1, case_id=1, aux_list=[uds_aux])
class ExampleUdsServerTest(pykiso.BasicTest):

    def setUp(self):
        """Register various callbacks for the test."""
        # register external pre-defined callbacks
        for callback in UDS_CALLBACKS:
            uds_aux.register_callback(callback)

        # handle extended diagnostics session request [0x10, 0x03]
        uds_aux.register_callback(request=0x1003, response=0x50031234)

        # handle incoming read data by identifier request with identifier [0x01, 0x02]
        uds_aux.register_callback(request=0x220102, response_data=b'DATA', data_length=6)

    def test_run(self):
        """Actual test. Simply wait a bit and expect the registered request to be␣
↪received
        (and the corresponding response to be sent to the client).
        """
        logging.info(
            f"--------------- RUN: {self.test_suite_id}, {self.test_case_id} ------------
↪---"
        )
        time.sleep(10)
        # access the previously registered callback
        extended_diag_session_callback = uds_aux.callbacks["0x1003"]
        self.assertGreater(
            extended_diag_session_callback.call_count,
            0,
            "Expected UDS request was not sent by the client after 10s",
        )

    def tearDown(self):
        """Unregister all callbacks registered by the auxiliary."""

        for callback in uds_aux.callbacks:
            uds_aux.unregister_callback(callback)
```

5.5.4 Modify the waiting time

Sending a huge amount of bytes over UDS can take some time and to avoid extra waiting time, users can modify the waiting time between two isotp packets of 4096 bytes.

It can be achieved using the public attribute from uds server auxiliary "tp_waiting_time".

5.6 Controlling an Yepkit USB hub

The ykush auxiliary offers commands to control an Yepkit USB hub, allowing to switch the USB ports on and off individually.

More information on the device can be found on the following link : YKUSH (Yepkit USB Switchable Hub).

5.6.1 Usage Examples

To use the auxiliary in your test scripts the auxiliary must be properly defined in the config yaml. Example:

```
auxiliaries:
    ykush_aux:
      config:
        # Serial number to connect to. Example: "YK00006"
        serial_number : null # null = auto detection.
      type: pykiso.lib.auxiliaries.ykush_auxiliary:YkushAuxiliary
```

Below find a example for the usage in a test script. All available methods are shown there.

```
##########################################################################
# Copyright (c) 2010-2022 Robert Bosch GmbH
# This program and the accompanying materials are made available under the
# terms of the Eclipse Public License 2.0 which is available at
# http://www.eclipse.org/legal/epl-2.0.
#
# SPDX-License-Identifier: EPL-2.0
##########################################################################

"""
Ykush auxiliary test
********************

:module: test_ykush

:synopsis: Example test that shows how to control the Ykush usb.

.. currentmodule:: test_ykush

"""

import logging

import pykiso
from pykiso.auxiliaries import ykush_aux
```

(continues on next page)

(continued from previous page)

```
@pykiso.define_test_parameters(suite_id=1, case_id=1, aux_list=[ykush_aux])
class ExampleYkushTest(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} ------------
↪---"
        )
        list_state = ykush_aux.get_all_ports_state()
        logging.info(f"The state of the ports are :{list_state}")

        logging.info("Power off all ports")
        ykush_aux.set_all_ports_off()

        logging.info("Check if all ports are off")
        self.assertEqual(ykush_aux.get_all_ports_state(), [0, 0, 0])

        logging.info("Power on the port 1")
        ykush_aux.set_port_on(port_number=1)

        logging.info("Get the state of the port 1")
        state_port_1 = ykush_aux.get_port_state(port_number=1)
        logging.info(f"Port 1 is {state_port_1}")

        logging.info("Check if the port is on")
        self.assertTrue(ykush_aux.is_port_on(port_number=1))

        logging.info("Power off the port number 1")
        ykush_aux.set_port_off(port_number=1)

        logging.info("Check if the port is off")
        self.assertTrue(ykush_aux.is_port_off(port_number=1))

        list_state = ykush_aux.get_all_ports_state()
        logging.info(f"The state of the ports are :{list_state}")

    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} -------
↪--------"
        )
```

CHAPTER

SIX

API DOCUMENTATION

6.1 Test Cases

6.1.1 Generic Test

module
test_case

synopsis
Basic extensible implementation of a TestCase, and of a Remote TestCase for Message Protocol / TestApp usage.

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*, *tag*, **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"]}
- **tag** (Optional[Dict[str, List[str]]]) – dictionary allowing users to filter the tests based on the keys and their value.

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

setUp()

Startup hook method to execute code before each test method.

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()

Closure hook method to execute code after each test method.

Return type
None

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

Base test-cases for Message Protocol / TestApp usage.

Initialize TestApp 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"]}
- **tag** (Optional[Dict[str, List[str]]]) – dictionary containing lists of variants and/or test levels when only a subset of tests needs to be executed

setUp()

Startup hook method to execute code before each test method.

Return type
None

tearDown()

Closure hook method to execute code after each test method.

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*, *tag=None*)

Decorator to fill out test parameters of the BasicTest and RemoteTest automatically.

pykiso.test_coordinator.test_case.**retry_test_case**(*max_try=2*, *rerun_setup=False*, *rerun_teardown=False*, *stability_test=False*)

Decorator: retry mechanism for testCase.

The aim is to cover the 2 following cases:

- Unstable test : get the test pass within the {max_try} attempt
- Stability test : run {max_try} time the test expecting no error

The **retry_test_case** comes with the possibility to re-run the setUp and tearDown methods automatically.

Parameters

- **max_try** (int) – maximum number of try to get the test pass.
- **rerun_setup** (bool) – call the "setUp" method of the test.
- **rerun_teardown** (bool) – call the "tearDown" method of the test.
- **stability_test** (bool) – run {max_try} time the test and raise an exception if an error occurs.

Returns
None, a testCase is not supposed to return anything.

Raises
Exception – if stability_test, the exception that occurred during the execution; if not stability_test, the exception that occurred at the last try.

6.2 Connectors

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

6.2.1 Included Connectors

Connector Interface

Interface Definition for Connectors, CChannels and Flasher

module
connector

synopsis
Interface for a channel

class pykiso.connector.**CChannel**(*processing=False*, ***kwargs*)

Abstract class for coordination channel.

Constructor.

Parameters
processing – if multiprocessing object is used.

abstract _cc_close()

Close the channel.

Return type
None

abstract _cc_open()

Open the channel.

Return type
None

abstract _cc_receive(*timeout*, ***kwargs*)

How to receive something from the channel.

Parameters

- **timeout** (float) – Time to wait in second for a message to be received
- **kwargs** – named arguments

Return type
Dict[str, Optional[bytes]]

Returns
dictionary containing the received bytes if successful, otherwise None

abstract _cc_send(*msg*, ***kwargs*)

Sends the message on the channel.

Parameters

- **msg** (Union[*Message*, bytes, str]) – Message to send out
- **kwargs** – named arguments

Return type
None

`cc_receive`(*timeout=0.1*, **args*, ***kwargs*)

Read a thread-safe message on the channel and send an acknowledgement.

Parameters

- **timeout** (float) – time in second to wait for reading a message
- **kwargs** – named arguments

Return type
Dict[str, Optional[bytes]]

Returns
the received message

`cc_send`(*msg*, **args*, ***kwargs*)

Send a thread-safe message on the channel and wait for an acknowledgement.

Parameters

- **msg** (Union[*Message*, bytes, str]) – message to send
- **kwargs** – named arguments

Return type
None

`close`()

Close a thread-safe channel.

Return type
None

`open`()

Open a thread-safe channel.

Return type
None

`shutdown`()

Unitialize channel. Will be called at the end of the test session.

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 (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]) – 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.

cc_example

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

_cc_close()

Close the channel.

Return type
None

_cc_open()

Open the channel.

Return type
None

_cc_receive(*timeout=0.1*)

Craft a Message that would be received from the device under test.

Parameters
timeout (float) – not use

Return type
Dict[str, Optional[bytes]]

Returns
dictionary containing the received bytes if successful, otherwise None

_cc_send(*msg*)

Sends the message on the channel.

Parameters
msg (bytes) – message to send.

Return type
None

cc_fdx_lauterbach

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** (str) – full path of Trace32 app to execute
- **t32_config** (str) – full path of Trace32 configuration file
- **t32_main_script_path** (str) – full path to the main cmm script to execute
- **t32_reset_script_path** (str) – full path to the reset cmm script to execute
- **t32_fdx_clr_buf_script_path** (str) – full path to the FDX reset cmm script to execute
- **t32_in_test_reset_script_path** (str) – full path to the board reset cmm script to execute
- **t32_api_path** (str) – full path of remote api
- **port** (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)

_cc_close()

Close FDX connection, UDP socket and shut down Trace32 App.

Return type
None

_cc_open()

Load the Trace32 library, open the application and open the FDX channels (in/out).

Return type
bool

Returns
True if Trace32 is correctly open otherwise False

_cc_receive(*timeout=0.1*)

Receive message using the FDX channel.

Return type
Dict[str, Union[bytes, str, None]]

Returns
message

_cc_send(*msg*)

Sends a message using FDX channel.

Parameters
msg (bytes) – message

Return type
int

Returns
poll length

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*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Available state for any scripts loaded into TRACE32.

cc_mp_proxy

Multiprocessing Proxy Channel

module
cc_mp_proxy

synopsis
concrete implementation of a multiprocessing proxy channel

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_mp_proxy.CCMpProxy(**kwargs)`

Multiprocessing Proxy CChannel for multi auxiliary usage.

Initialize attributes.

`_bind_channel_info(proxy_aux)`

Bind a *MpProxyAuxiliary* instance that is instanciated in order to handle the connection of multiple auxiliaries to a single communication channel in order to hide the underlying proxy setup.

Parameters
proxy_aux (*MpProxyAuxiliary*) – the proxy auxiliary instance that is holding the real communication channel.

`_cc_close()`

Close proxy channel.

Due to usage of multiprocessing the queue_in and queue_out state doesn't have to change in order to ensure that ProxyAuxiliary works even if suspend or resume is called.

Return type
None

`_cc_open()`

Open proxy channel.

Due to usage of multiprocessing the queue_in and queue_out state doesn't have to change in order to ensure that ProxyAuxiliary works even if suspend or resume is called.

Return type
None

`_cc_receive(timeout=0.1)`

Depopulate the queue out of the proxy connector.

Parameters
timeout (float) – not used

Return type
Union[Dict[str, Union[bytes, int]], Dict[str, Optional[bytes]], Dict[str, Optional[*Message*]], Dict[str, None]]

Returns
bytes and source when it exist. if queue timeout is reached return None

`_cc_send(*args, **kwargs)`

Populate the queue in of the proxy connector.

Parameters

- **args** (tuple) – tuple containing positionnal arguments
- **kwargs** (dict) – dictionary containing named arguments

Return type
None

cc_pcan_can

cc_proxy

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 to bind multiple auxiliaries to a single 'physical' CChannel.

Initialize attributes.

_bind_channel_info(*proxy_aux*)

Bind a *ProxyAuxiliary* instance that is instanciated in order to handle the connection of multiple auxiliaries to a single communication channel.

This allows to access the real communication channel's attributes and hides the underlying proxy setup.

Parameters
proxy_aux (*ProxyAuxiliary*) – the proxy auxiliary instance that is holding the real communication channel.

_cc_close()

Close proxy channel.

Return type
None

_cc_open()

Open proxy channel.

Return type
None

_cc_receive(*timeout=0.1*)

Depopulate the queue out of the proxy connector.

Parameters
timeout (float) – not used

Return type
Union[Dict[str, Union[bytes, int]], Dict[str, Optional[bytes]], Dict[str, Optional[*Message*]], Dict[str, None]]

Returns
bytes and source when it exist. if queue timeout is reached return None

`_cc_send`(**args*, ***kwargs*)
Call the attached ProxyAuxiliary's transmission callback with the provided arguments.

Parameters
- **args** (Any) – positionnal arguments to pass to the callback
- **kwargs** (Any) – named arguments to pass to the callback

Return type
None

`attach_tx_callback`(*func*)
Attach to a callback to the _cc_send method.

Parameters
func (Callable) – function to call when _cc_send is called

Return type
None

`close`()
Close a thread-safe channel.

Return type
None

`detach_tx_callback`()
Detach the current callback.

Return type
None

`open`()
Open a thread-safe channel.

Return type
None

cc_raw_loopback

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.

Parameters
processing – if multiprocessing object is used.

_cc_close()

Close loopback channel.

Return type
None

_cc_open()

Open loopback channel.

Return type
None

_cc_receive(*timeout*)

Read message by simply removing an element from the left side of deque.

Parameters
timeout (float) – timeout applied on receive event

Return type
Dict[str, Optional[bytes]]

Returns
dictionary containing the received bytes if successful, otherwise None

_cc_send(*msg*)

Send a message by simply putting message in deque.

Parameters
msg (Union[*Message*, bytes, str]) – message to send, should be Message type or bytes.

Return type
None

cc_rtt_segger

cc_serial

cc_socket_can

Setup SocketCAN

To use the socketCAN connector you have to make sure that your can socket has been initialized correctly.

```
sudo ip link set can0 up type can bitrate 500000 sample-point 0.75 dbitrate 2000000␣
↪dsample-point 0.8 fd on
sudo ip link set up can0
```

Make sure that ifconfig shows a socket can interface. Example shows can0 as available interface:

```
ifconfig
# outputs->
can0: flags=193<UP,RUNNING,NOARP>  mtu 72
        unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00  txqueuelen 1000  (UNSPEC)
        RX packets 30  bytes 90 (90.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 30  bytes 90 (90.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Warning: SocketCAN is only available under Linux.

cc_tcp_ip

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

_cc_close()

Close UDP socket.

Return type
None

_cc_open()

Connect to socket with configured port and IP address.

Return type
None

_cc_receive(*timeout=0.01*)

Read message from socket.

Parameters
timeout – time in second to wait for reading a message

Return type
Dict[str, Optional[bytes]]

Returns
Message if successful, otherwise none

_cc_send(*msg*)

Send a message via socket.

Parameters
msg (bytes) – message to send

Return type
None

cc_uart

cc_udp

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

_cc_close()

Close the udp socket.

Return type
None

_cc_open()

Open the udp socket.

Return type
None

_cc_receive(*timeout=1e-07*)

Read message from socket.

Parameters
timeout (float) – timeout applied on receive event

Return type
Dict[str, Optional[bytes]]

Returns
dictionary containing the received bytes if successful, otherwise None

_cc_send(*msg*)

Send message using udp socket

Parameters
msg (bytes) – message to send, should bytes.

Return type
None

cc_udp_server

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

_cc_close()

Close UDP socket.

Return type
None

_cc_open()

Bind UDP socket with configured port and IP address.

Return type
None

_cc_receive(*timeout=1e-07*)

Read message from UDP socket.

Parameters
timeout – timeout applied on receive event

Return type
Dict[str, Optional[bytes]]

Returns
Message if successful, otherwise none

_cc_send(*msg*)

Send back a UDP message to the previous sender.

Parameters
msg (bytes) – message to sent, should be bytes

Return type
None

cc_usb

cc_vector_can

cc_visa

cc_process

Process Channel

module
cc_process

synopsis
CChannel implementation for process execution.

The CCProcess channel provides functionality to start a process and to communicate with it.

`class pykiso.lib.connectors.cc_process.CCProcess(`*shell=False*, *pipe_stderr=False*, *pipe_stdout=True*, *pipe_stdin=False*, *text=True*, *cwd=None*, *env=None*, *encoding=None*, *executable=None*, *args=[]*, ***kwargs*`)`

Channel to run processes

Initialize a process

Parameters

- **shell** (bool) – Start process through shell
- **pipe_stderr** (bool) – Pipe stderr for reading with this connector
- **pipe_stdout** (bool) – Pipe stdout for reading with this connector
- **pipe_stdin** (bool) – Pipe stdin for writing with this connector
- **text** (bool) – Read/write stdout, stdin, stderr in binary mode
- **cwd** (Optional[str]) – The current working directory for the new process
- **env** (Optional[str]) – Environment variables for the new process
- **encoding** (Optional[str]) – Encoding to use in text mode
- **executable** (Optional[str]) – The path of the executable for the process
- **args** (List[str]) – Process arguments

`_cc_close()`

Close the channel.

Return type
None

`_cc_open()`

Implement abstract method

Return type
None

_cc_receive(*timeout=0.0001*)

Receive messages

Parameters

- **timeout** (float) – Time to wait in seconds for a message to be received
- **size** – unused

Return type
Union[str, ByteString]

return The received message

_cc_send(*msg*, ***kwargs*)

Execute process commands or write data to stdin

Parameters
msg (Union[str, ByteString]) – data to send

Raises
CCProcessError – Stdin pipe is not enabled

Return type
None

_cleanup()

Cleanup threads and process objects

Return type
None

static _create_message_dict(*msg*)

Create a dict from an entry in the process queue

Parameters
msg (Union[*ProcessMessage*, *ProcessExit*]) – The message to convert

Return type
dict

Returns
The dictionary

_read_existing()

Read buffered messages that where already received from the process. Messages from the same stream are combined. This is only used in binary mode.

Return type
Optional[*ProcessMessage*]

Returns
Existing messages

_read_thread(*stream*, *name*)

Thread for reading data from stdout or stderr

Parameters

- **stream** (IO) – The stream to read from
- **name** (str) – The name of the stream

Return type
None

_start_read_thread(*stream*, *name*)

Start a read thread

Parameters

- **stream** (IO) – The stream to read from
- **name** (str) – The name of the stream

Return type
Thread

Returns
The thread object

start(*executable=None*, *args=None*)

Start a process

Parameters

- **executable** (Optional[str]) – The executable path. Default to path specified in yaml if not given.
- **args** (Optional[List[str]]) – The process arguments. Default to arguments specified in yaml if not given.

Raises
CCProcessError – Process is already running

exception pykiso.lib.connectors.cc_process.**CCProcessError**

class pykiso.lib.connectors.cc_process.**ProcessExit**(*exit_code*)

Contains information about process exit

class pykiso.lib.connectors.cc_process.**ProcessMessage**(*stream*, *data*)

Holds the data that is read from the process

6.2.2 Flashers

flash_jlink

flash_lauterbach

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** (str) – full path of Trace32 app to execute
- **t32_config** (str) – full path of Trace32 configuration file
- **t32_script_path** (str) – full path to .cmm flash script to execute
- **t32_api_path** (str) – full path of remote api
- **port** (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.

Return type
None

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.

open()

Open UDP socket between ITF and Trace32 loaded app.

The open command leads to the following sub-tasks execution: :rtype: None

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

class pykiso.lib.connectors.flash_lauterbach.**MessageLineState**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Use to determine Message reading command.

class pykiso.lib.connectors.flash_lauterbach.**ScriptState**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Use to determine script command execution.

cc_flasher_example

Fake Flasher Channel for testing

module
cc_flasher_example

synopsis
fake flasher implementation

class pykiso.lib.connectors.cc_flasher_example.**FlasherExample**(*name*, ***kwargs*)

A Flasher adapter for testing purpose only.

Constructor.

Parameters

- **name** (str) – flasher's alias
- **kwargs** – named arguments

close()

Close flasher and free resources.

Return type
None

flash()

Fake a firmware update.

Return type
None

open()

Initialize the flasher.

Return type
None

6.3 Auxiliaries

6.3.1 Auxiliary interfaces

auxiliary

Auxiliary common Interface Definition

module
auxiliary

synopsis
base auxiliary interface

`class pykiso.auxiliary.AuxiliaryCommon`

Class use to encapsulate all common methods/attributes for both multiprocessing and thread auxiliary interface.

Auxiliary common attributes initialization.

`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_copy`(**args*, ***config*)

Create a copy of the actual auxiliary instance with the new desired configuration.

Note: only named arguments have to be used

Warning: the call of create_copy will automatically suspend the current auxiliary until the it copy is destroyed

Parameters
config (dict) – new desired auxiliary configuration

Return type
AuxiliaryCommon

Returns
a brand new auxiliary instance

Raises
Exception – if positional parameters is given or unknown named parameters are given

`abstract create_instance`()

Handle auxiliary creation.

Return type
bool

`abstract delete_instance`()

Handle auxiliary deletion.

Return type
bool

destroy_copy()

Stop the current auxiliary copy and resume the original. :rtype: None

> **Warning:** stop the copy auxiliary will automatically start the base/original one

lock_it(*timeout_in_s*)

Lock to ensure exclusivity.

Parameters
timeout_in_s (float) – 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, by running the create_instance method in the background. :rtype: None

> **Warning:** due to the usage of create_instance if an issue occurred the exception AuxiliaryCreationError is raised.

abstract run()

Run function of the auxiliary.

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** (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

mp_auxiliary

Multiprocessing based Auxiliary Interface

module
mp_auxiliary

synopsis
common multiprocessing based auxiliary interface

class pykiso.interfaces.mp_auxiliary.**MpAuxiliaryInterface**(*name=None*, *is_proxy_capable=False*, *activate_log=None*)

Defines the interface of all multiprocessing based auxiliaries.

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

Auxiliary initialization.

Parameters

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

create_instance()

Create an auxiliary instance and ensure the communication to it.

Return type
bool

Returns
verdict on instance creation, True if everything was fine otherwise False

Raises
AuxiliaryCreationError – if instance creation failed

`delete_instance()`

Delete an auxiliary instance and its communication to it.

Return type
bool

Returns
verdict on instance deletion, False if everything was fine otherwise True(instance was not deleted correctly)

`initialize_loggers()`

Initialize the logging mechanism for the current process.

Return type
None

`run()`

Run function of the auxiliary process.

Return type
None

simple_auxiliary

Simple Auxiliary Interface

module
simple_auxiliary

synopsis
common auxiliary interface for very simple auxiliary (without usage of thread or multiprocessing)

`class pykiso.interfaces.simple_auxiliary.SimpleAuxiliaryInterface(name=None, activate_log=None)`

Define the interface for all simple auxiliary where usage of thread or mulitprocessing is not necessary.

Auxiliary initialization.

Parameters

- **activate_log** (List[str]) – loggers to deactivate
- **name** (str) – alias of the auxiliary instance

`create_instance()`

Create an auxiliary instance and ensure the communication to it.

Return type
bool

Returns
True if creation was successful otherwise False

Raises
AuxiliaryCreationError – if instance creation failed

delete_instance()

Delete an auxiliary instance and its communication to it.

Return type
bool

Returns
True if deletion was successful otherwise False

resume()

Resume current auxiliary's run, by running the create_instance method in the background. :rtype: None

> **Warning:** due to the usage of create_instance if an issue occurred the exception AuxiliaryCreationError is raised.

stop()

Stop the auxiliary

suspend()

Suspend current auxiliary's run.

Return type
None

thread_auxiliary

Thread based Auxiliary Interface

module
thread_auxiliary

synopsis
common thread based auxiliary interface

> **Warning:** AuxiliaryInterface will be deprecated in a few releases!

class pykiso.interfaces.thread_auxiliary.**AuxiliaryInterface**(*name=None*, *is_proxy_capable=False*, *activate_log=None*, *auto_start=True*)

Defines the Interface of all thread based auxiliaries.

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

Auxiliary initialization.

Parameters

- **name** (str) – alias of the auxiliary instance
- **is_proxy_capable** (bool) – notify if the current auxiliary could be (or not) associated to a proxy-auxiliary.
- **activate_log** (List[str]) – loggers to deactivate
- **auto_start** (bool) – determine if the auxiliayry is automatically started (magic import) or manually (by user)

`create_instance`()

Create an auxiliary instance and ensure the communication to it.

Return type
`bool`

Returns
message.Message() - Contain received message

Raises
AuxiliaryCreationError – if instance creation failed

`delete_instance`()

Delete an auxiliary instance and its communication to it.

Return type
`bool`

Returns
message.Message() - Contain received message

`run`()

Run function of the auxiliary thread.

Return type
`None`

`start`()

Start the thread and create the auxiliary only if auto_start flag is False.

Return type
`None`

Double-threaded auxiliary

Double Threaded based Auxiliary Interface

module
dt_auxiliary

synopsis
common double threaded based auxiliary interface

`class pykiso.interfaces.dt_auxiliary.`**AuxCommand**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Contain all available auxiliary's commands.

CREATE_AUXILIARY = 1

create auxiliary command id

DELETE_AUXILIARY = 2

delete auxiliary command id

`class pykiso.interfaces.dt_auxiliary.`**DTAuxiliaryInterface**(*name=None*, *is_proxy_capable=False*, *connector_required=True*, *activate_log=None*, *tx_task_on=True*, *rx_task_on=True*, *auto_start=True*)

Common interface for all double threaded auxiliary. A so called << double threaded >> auxiliary, simply encapsulate two threads one for the reception and one for the transmmission.

Initialize auxiliary attributes

Parameters

- **name** (str) – alias of the auxiliary instance
- **is_proxy_capable** (bool) – notify if the current auxiliary could be (or not) associated to a proxy-auxiliary.
- **connector_required** (bool) – define if a connector is required for this auxiliary.
- **activate_log** (List[str]) – loggers to deactivate
- **tx_task_on** – enable or not the tx thread
- **rx_task_on** – enable or not the rx thread
- **auto_start** (bool) – determine if the auxiliayry is automatically started (magic import) or manually (by user)

create_instance()

Start auxiliary's running tasks and activities.

Return type
bool

Returns
True if the auxiliary is created otherwise False

Raises
AuxiliaryCreationError – if instance creation failed

delete_instance()

Stop auxiliary's running tasks and activities.

Return type
bool

Returns
True if the auxiliary is deleted otherwise False

resume()

Resume current auxiliary's run.

> **Warning:** due to the usage of create_instance if an issue occurred the exception AuxiliaryCreationError is raised.

Return type
bool

Returns
True if the auxiliary is resumed otherwise False

run_command(*cmd_message*, *cmd_data=None*, *blocking=True*, *timeout_in_s=5*, *timeout_result=None*)

Send a request by transmitting it through queue_in and waiting for a response using queue_out.

Parameters

- **cmd_message** (Any) – command request to the auxiliary
- **cmd_data** (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
- **timeout_result** (Any) – Value to return when the command times out. Defaults to None.

Raises
pykiso.exceptions.AuxiliaryNotStarted – if a command is executed although the auxiliary was not started.

Return type
Any

Returns
True if the request is correctly executed otherwise False

shutdown()

Uninitialize method. Will be called at the end of the test session.

start()

Force the auxiliary to start all running tasks and activities.

> **Warning:** due to the usage of create_instance if an issue occurred the exception AuxiliaryCreationError is raised.

Return type
bool

Returns
True if the auxiliary is started otherwise False

stop()

Force the auxiliary to stop all running tasks and activities.

Return type
bool

Returns
True if the auxiliary is stopped otherwise False

suspend()

Supend current auxiliary's run.

Return type
bool

Returns
True if the auxiliary is suspend otherwise False

wait_for_queue_out(*blocking=False*, *timeout_in_s=0*)

Wait for data from the queue out.

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
Optional[Any]

Returns
data contained in the auxiliary's queue_out

pykiso.interfaces.dt_auxiliary.**close_connector**(*func*)

Close current associated auxiliary's channel.

Parameters
func (Callable) – decorated method

Return type
Callable

Returns
inner decorated function

pykiso.interfaces.dt_auxiliary.**flash_target**(*func*)

Flash firmware on the target, using associated auxiliary's flasher channel.

Parameters
func (Callable) – decorated method

Return type
Callable

Returns
inner decorated function

pykiso.interfaces.dt_auxiliary.**open_connector**(*func*)

Open current associated auxiliary's channel.

Parameters
func (Callable) – decorated method

Return type
Callable

Returns
inner decorated function

6.3.2 Existing Auxiliaries

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

acroname_auxiliary

Example can be found here *Controlling an acroname USB hub*.

Acroname Control Auxiliary

module
acroname_auxiliary

synopsis
Auxiliary used to control acroname usb hubs.

class pykiso.lib.auxiliaries.acroname_auxiliary.**AcronameAuxiliary**(*serial_number=None*, ***kwargs*)

Auxiliary used to control acroname usb hubs

Constructor

Parameters
serial_number (str) – serial number to connect to as hex string. Example "0x66F4859B"

static eval_result(*result*)

Log error message if exist from acroname Result object. :type result: Result :param result: result code to evaluate

Return type
None

get_port_current(*port*, *unit='A'*)

Get the current through the power line for selected usb port.

Parameters

- **port** (int) – the USB port number
- **unit** (str) – unit of the result in "uA", "mA" or "A". Default "A"

Return type
Optional[float]

Returns
port current for given unit. None if unit is not supported.

get_port_current_limit(*port*, *unit='A'*)

Get the current limit for the port.

Parameters

- **port** (int) – the USB port number
- **unit** (str) – unit of the result in "uA", "mA" or "A". Default "A"

Return type
Optional[float]

Returns
port current limit for given unit. None if unit is not supported.

get_port_voltage(*port*, *unit='V'*)

Get the voltage of the selected usb port.

Parameters

- **port** (int) – the USB port number
- **unit** (str) – unit of the result in "uV", "mV" or "V". Default "V"

Return type
Optional[float]

Returns
port voltage for given unit. None if unit is not supported.

`set_port_current_limit`(*port*, *amps*, *unit='A'*)

Set the current limit for the port. If the set limit is not achievable, devices will round down to the nearest available current limit setting.

Parameters

- **port** (int) – the USB port number
- **amps** (float) – value for port current to set in "uA", "mA" or "A". Default "A"
- **unit** (str) – unit for the value to set. Default Ampere

Return type
int

Returns
brainstem error code. 0 if no error.

`set_port_disable`(*port*)

Disable power and data lines for a USB port.

Parameters
port (int) – the USB port number

Return type
int

Returns
brainstem error code. 0 if no error.

`set_port_enable`(*port*)

Enable power and data lines for a USB port.

Parameters
port (int) – the USB port number

Return type
int

Returns
brainstem error code. 0 if no error.

communication_auxiliary

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

clear_buffer()

Clear buffer from old stacked objects

Return type
None

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

Receive a raw message.

Parameters

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

Return type
Optional[bytes]

Returns
raw message

run_command(*cmd_message*, *cmd_data=None*, *blocking=True*, *timeout_in_s=None*)

Send a request by transmitting it through queue_in and populate queue_tx with the command verdict (successful or not).

Parameters

- **cmd_message** (Any) – command to send
- **cmd_data** (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 if the request is correctly executed otherwise False

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

dut_auxiliary

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**(*com=None*, *flash=None*, ***kwargs*)

Device Under Test(DUT) auxiliary implementation.

Constructor.

Parameters

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

create_instance()

Create DUT auxiliary instance.

Overridden from base interface in order to use the TX and RX tasks, and not duplicate auxiliary method. Execute directly the ping-pong to initiate the communication with the DUT.

Return type
bool

Returns
True if the auxiliary is created and ping-pong successful otherwise False

Raises
AuxiliaryCreationError – if instance creation failed

evaluate_report(*report_msg*)

Evaluate the report type and log the appropriated message.

Parameters
report_msg (*Message*) – reeceived report message

Return type
None

evaluate_response(*response*)

Evaluate if the received message is knowned and type of report.

Note: if a log message type is received just log it

Parameters
response (*Message*) – reeceived response

Return type
bool

Returns
True if the response is a report otherwise False

send_abort_command(*timeout*)

Send abort command to the DUT.

> **Warning:** if the DUT doesn't acknowledge to the abort command the auxiliary is restarted, a brand new connection is established

Parameters
timeout (int) – how long to wait for an acknowledgement from the DUT (seconds)

Returns
True if the command is acknowledged otherwise False

send_fixture_command(*command*, *timeout*)

Send command related to test execution (test case setup, test case run...) to the DUT.

Parameters

- **command** (*Message*) – command to execute on DUT side
- **timeout** (int) – how long to wait for an acknowledgement from the DUT (seconds)

Return type
bool

Returns
True if the command is acknowledged otherwise False

send_ping_command(*timeout*)

Send ping command to the DUT.

Parameters
timeout (int) – how long to wait for an acknowledgement from the DUT (seconds)

Return type
bool

Returns
True if the command is acknowledged otherwise False

wait_and_get_report(*blocking=False*, *timeout_in_s=0*)

Wait for the report coming from the DUT.

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
Optional[*Message*]

Returns
if a report is received return it otherwise None

pykiso.lib.auxiliaries.dut_auxiliary.**check_acknowledgement**(*func*)

Check ifthe DUT has acknowledge the previous sent command.

Parameters
func (Callable) – decorated method

Return type
Callable

Returns
decorator inner function

pykiso.lib.auxiliaries.dut_auxiliary.**restart_aux**(*func*)

Force he auxiliary restart if the command is not acknowledge

Parameters
func (Callable) – decorated method

Return type
Callable

Returns
decorator inner function

pykiso.lib.auxiliaries.dut_auxiliary.**retry_command**(*tries*)

Force to resend the command a define number of times in case of failure.

Parameters
tries (int) – maximum number of try to get the acknowledgement from the DUT

Return type
Callable

Returns
inner decorator function

instrument_control_auxiliary

Example can be found here *Controlling an Instrument*.

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	InstrumentControl Auxiliary
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

InstrumentControl 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.InstrumentControl

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** (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
Optional[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
Optional[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** (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

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

Send a write request to the instrument.

Parameters

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

Return type
str

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** (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

Library of instruments communicating via VISA

module
lib_instruments

synopsis
Dictionaries containing the appropriate SCPI commands for some instruments.

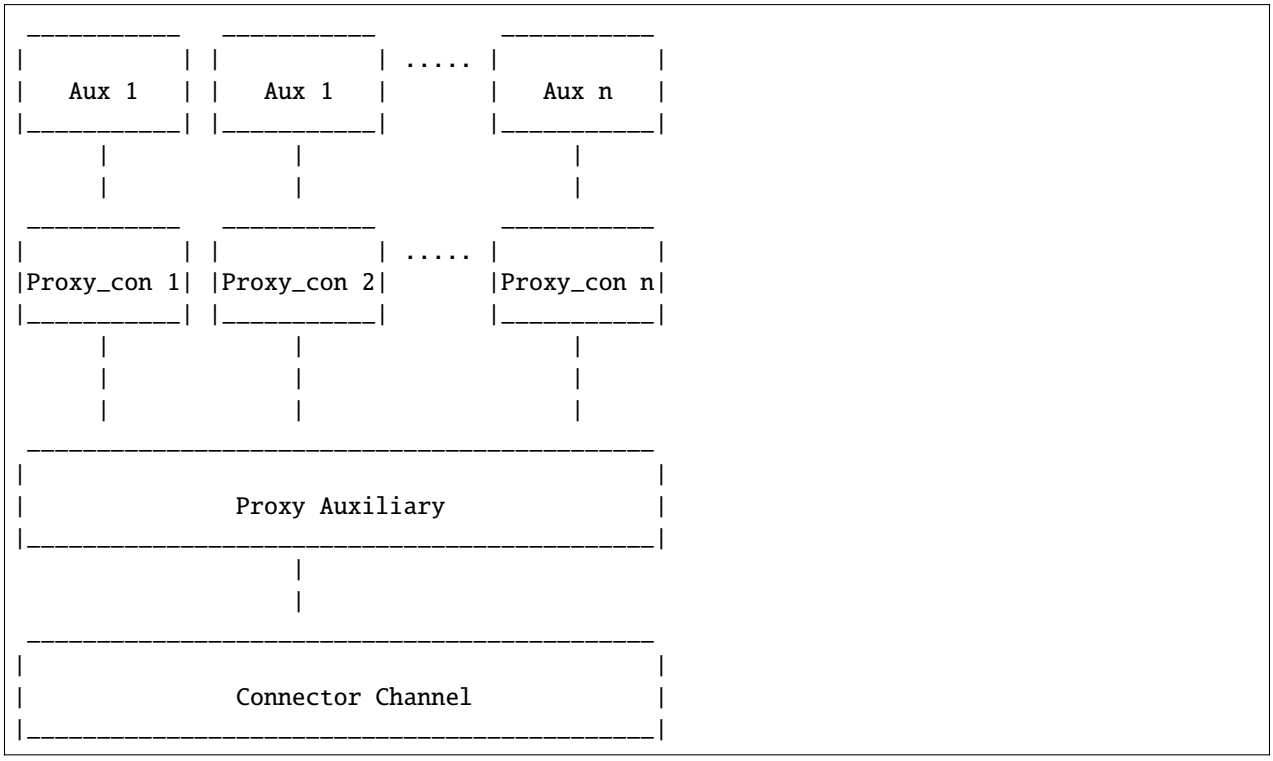
mp_proxy_auxiliary

Multiprocessing Proxy Auxiliary

module
mp_proxy_auxiliary

synopsis
concrete implementation of a multiprocessing proxy auxiliary

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



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

Proxy auxiliary for multi auxiliaries communication handling.

..note :: this auxiliary version is using the multiprocessing
auxiliary interface.

Initialize attributes.

Parameters

- **com** (*CChannel*) – Communication connector
- **aux_list** (List[str]) – list of auxiliary's alias
- **activate_trace** (bool) – True if the trace is activate otherwise False
- **trace_dir** (Optional[str]) – trace directory path (absolute or relative)
- **trace_name** (Optional[str]) – trace full name (without file extension)

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[*CCMpProxy*]

Returns
tuple containing all connectors associated to all given auxiliaries

run()

Run function of the auxiliary process.

Return type
None

class pykiso.lib.auxiliaries.mp_proxy_auxiliary.**TraceOptions**(*activate*, *dir*, *name*)

Create new instance of TraceOptions(activate, dir, name)

activate

Alias for field number 0

dir

Alias for field number 1

name

Alias for field number 2

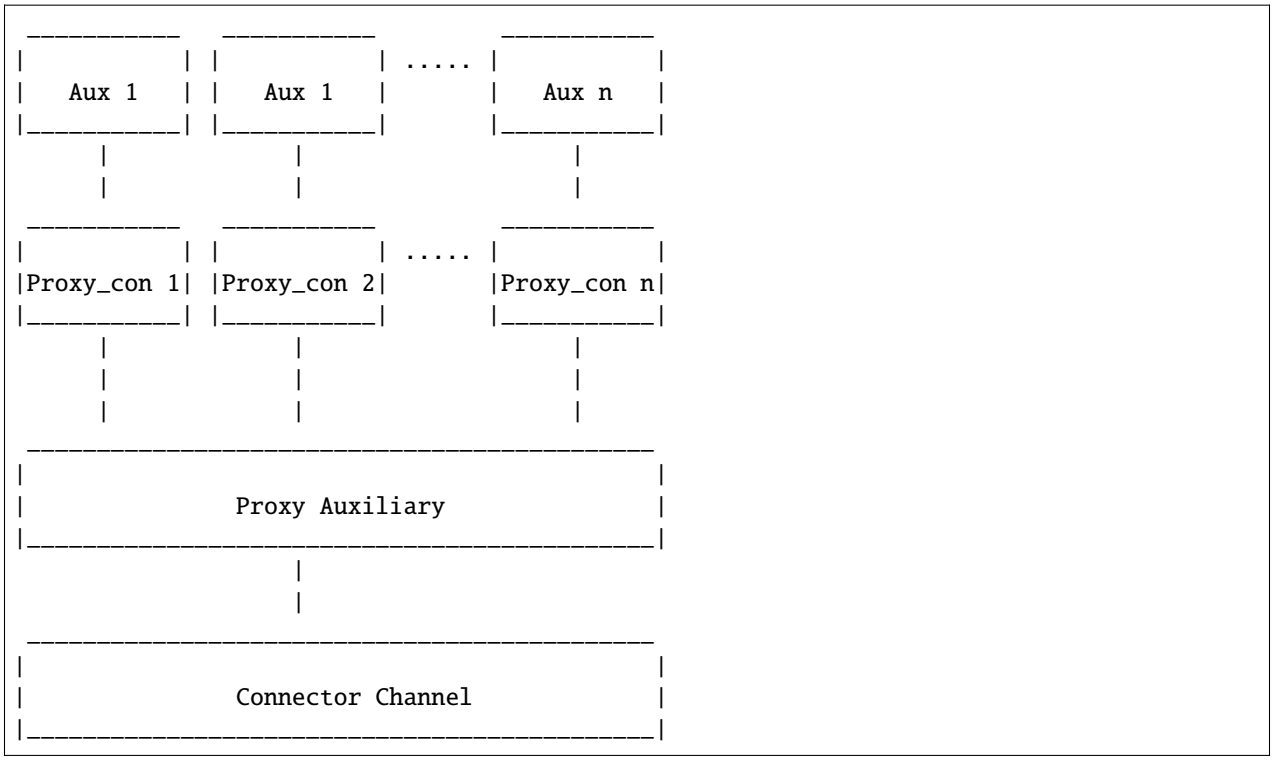
proxy_auxiliary

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
- **activate_trace** (bool) – log all received messages in a dedicated trace file or not
- **trace_dir** (Optional[str]) – where to place the trace
- **trace_name** (Optional[str]) – trace's file name

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 aliases or instances.

Return type
Tuple[*CCProxy*, ...]

Returns
tuple containing all connectors associated to all given auxiliaries.

run_command(*conn*, **args*, ***kwargs*)

Transmit an incoming request from a linked proxy channel to the proxy auxiliary's channel.

Parameters

- **conn** (*CChannel*) – current proxy channel instance which the command comes from
- **args** (tuple) – postional arguments
- **args** – named arguments

Return type
None

record_auxiliary

Example can be found here *Passively record a channel*.

Record Auxiliary

module
record_auxiliary

synopsis
Auxiliary used to record a connectors receive channel.

class pykiso.lib.auxiliaries.record_auxiliary.**RecordAuxiliary**(*com*, *is_active=False*, *timeout=0*, *log_folder_path=''*, *max_file_size=50000000*, *multiprocess=False*, *manual_start_record=False*, ***kwargs*)

Auxiliary used to record a connectors receive channel.

Constructor.

Parameters

- **com** (*CChannel*) – Communication connector to record
- **is_active** (bool) – Flag to actively poll receive channel in another thread
- **timeout** (float) – timeout for the receive channel
- **log_path** – path to the log folder
- **max_file_size** (int) – maximal size of the data string
- **multiprocess** (bool) – use a Process instead of a Thread for active polling. Note1: the data will automatically be saved. Note2: if proxy usage, all connectors should be 'CCMpProxy' and 'processing' flag set to True
- **manual_start_record** (bool) – flag to not start recording on auxiliary creation

clear_buffer()

Clean the buffer that contain received messages.

Return type
None

dump_to_file(*filename*, *mode='w+'*, *data=None*)

Writing data in file.

Parameters

- **filename** (str) – name of the file where data are saved
- **mode** (str) – modes of opening a file (eg: w, a)
- **data** (str) – Optional write/append specific data to the file.

Return type
bool

Returns
True if the dumping has been successful, False else

Raises
FileNotFoundError – if the given folder path is not a folder

get_data()

Return the entire log buffer content.

Return type
str

Returns
buffer content

is_log_empty()

Check if logs are available in the log buffer.

Return type
bool

Returns
True if log is empty, False either

is_message_in_full_log(*message*)

Check for a message being in log.

Parameters
message (str) – message to check presence in logs.

Returns
True if a message is in log, False otherwise

is_message_in_log(*message*, *from_cursor=True*, *set_cursor=True*, *display_log=False*)

Check for a message being in log.

Parameters

- **message** (str) – str message to check presence in logs.
- **from_cursor** (bool) – whether to get the logs from the last cursor position (True) or the full logs
- **set_cursor** (bool) – whether to update the cursor
- **display_log** (bool) – whether to log (via logging) the retrieved part or just return it

Return type
bool

Returns
True if a message is in log, False otherwise.

`new_log()`

Get new entries (after cursor position) from the log. This will set the cursor.

Return type
`str`

Returns
return log after cursor

`static parse_bytes(data)`

Decode the received bytes

Parameters
data (`bytes`) – data to be decoded

Return type
`str`

Returns
data decoded

`previous_log()`

set cursor position to current position.

This will also display the logs from the last cursor position in the log.

Return type
`str`

Returns
log from the last current position

`receive()`

Open channel and actively poll the connectors receive channel. Stop and close connector when stop receive event has been set.

Return type
`None`

`search_regex_current_string(regex)`

Returns all occurrences found by the regex in the logs and
message received.

Parameters
regex (`str`) – str regex to compare to logs

Return type
`Optional[List[str]]`

Returns
list of matches with regular expression in the current string

`search_regex_in_file(regex, filename)`

Returns all occurrences found by the regex in the logs and message received.

Parameters

- **regex** (`str`) – str regex to compare to logs

- **filename** (str) – filename of the desired file

Return type
Optional[List[str]]

Returns
list of matches with regular expression in the chosen file

search_regex_in_folder(*regex*)

Returns all occurrences found by the regex in the logs and message received.

Parameters
regex (str) – str regex to compare to logs

Return type
Optional[Dict[str, List[str]]]

Returns
dictionary with filename and the list of matches with regular expression

Raises
FileNotFoundError – if the given folder path is not a folder

set_data(*data*)

Add data to the already existing data string.

Parameters
data (str) – the data to be write over the existing string

Return type
None

start_recording()

Clear buffer and start recording.

Return type
None

stop_recording()

Stop recording.

Return type
None

wait_for_message_in_log(*message*, *timeout=10.0*, *interval=0.1*, *from_cursor=True*, *set_cursor=True*, *display_log=False*, *exception_on_failure=True*)

Poll log at every interval time, fail if messages has not shown up within the specified timeout and exception set to True, log an error otherwise.

Parameters

- **message** (str) – str message expected to show up
- **timeout** (float) – int timeout in seconds for the check
- **interval** (float) – int period in seconds for the log poll
- **from_cursor** (bool) – whether to get the logs from the last cursor position (True) or the full logs
- **set_cursor** (bool) – whether to update the cursor to the last log position
- **display_log** (bool) – whether to log (via logging) the retrieved part or just return it

- **exception_on_failure** (bool) – if set, raise a TimeoutError if the expected messages wasn't found in the logs. Otherwise, simply output a warning.

Return type
bool

Returns
True if the message have been received in the log, False otherwise

Raises
TimeoutError – when a given message has not arrived in time

class pykiso.lib.auxiliaries.record_auxiliary.**StringIOHandler**(*multiprocess=False*)

Constructor

Parameters
multiprocess (bool) – use a thread or multiprocessing lock.

get_data()

Get data from the string

Return type
str

Returns
data from the string

set_data(*data*)

Add data to the already existing data string

Parameters
data (str) – the data to be write over the existing string

Return type
None

simulated_auxiliary

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

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

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

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

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.

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.

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

UDS Auxiliary

UDS Server Auxiliary

ykush_auxiliary

Example can be found here *Controlling an Yepkit USB hub*.

Ykush Auxiliary

module
ykush_auxiliary

synopsis
Auxiliary that can power on and off ports on an Ykush device.

class pykiso.lib.auxiliaries.ykush_auxiliary.**PortState**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

class pykiso.lib.auxiliaries.ykush_auxiliary.**YkushAuxiliary**(*serial_number=None*, ***kwargs*)

Auxiliary used to power on and off the ports.

Initialize attribute

Parameters
serial – Serial number of the device to connect, if he is not defined then it will connect to the first Ykush device it find, defaults to None.

Raises
YkushDeviceNotFound – If no device is found or the serial number is not the serial of one device.

check_port_number(*port_number*)

Check if the port indicated is a port of the device

Raises
YkushPortNumberError – Raise error if no port has this number

connect_device(*serial=None*)

Find an Ykush device, will automatically connect to the first one it find, if you have multiple connected you have to precise the serial number of the device.

Parameters
serial (int) – serial number of the device, defaults to None

Raises
YkushDeviceNotFound – if no ykush device is found

get_all_ports_state()

Returns the state of all the ports.

Raises
YkushStatePortNotRetrieved – The states couldn't be retrieved

Return type
List[*PortState*]

Returns
list with 0 if a port is off, 1 if on

`get_firmware_version()`

Returns a tuple with YKUSH firmware version in format (major, minor).

Return type
`Tuple[int, int]`

`get_number_of_port()`

Returns the number of port on the ykush device

Return type
`int`

`get_port_state(port_number)`

Returns a specific port state.

Raises
YkushStatePortNotRetrieved – If the state couldn't be retrieved

Return type
PortState

Returns
0 if the port is off, 1 if the port is on

`get_serial_number_string()`

Returns the device serial number string

Return type
`str`

`static get_str_state(state)`

Return the str of a state

Parameters
state (`int`) – 1 (port on), 0 (port off)

Return type
`str`

Returns
ON, OFF

`is_port_off(port_number)`

Check if a port is off.

Return type
`bool`

Returns
True if a port is off, else False

`is_port_on(port_number)`

Check if a port is on.

Return type
`bool`

Returns
True if a port is on, else False

set_all_ports(*state*)

Power off or on all YKUSH ports.

Parameters
state (int) – state wanted 1 for On, 0 for Off

Raises
YkushSetStateError – if the operation had an error

set_all_ports_off()

Power off all YKUSH downstreams ports.

Raises
YkushSetStateError – if the operation had an error

set_all_ports_on()

Power on all YKUSH downstreams ports.

Raises
YkushSetStateError – if the operation had an error

set_port_off(*port_number*)

Power off a specific port.

Raises
YkushSetStateError – if the operation had an error

set_port_on(*port_number*)

Power on a specific port.

Raises
YkushSetStateError – if the operation had an error

set_port_state(*port_number*, *state*)

Set a specific port On or Off.

Raises
YkushSetStateError – if the operation had an error

Parameters
state (int) – 1 (port on), 0 (port off)

exception pykiso.lib.auxiliaries.ykush_auxiliary.**YkushDeviceNotFound**

Raised when no Ykush device is found.

exception pykiso.lib.auxiliaries.ykush_auxiliary.**YkushError**

General Ykush specific exception used as basis for all others.

exception pykiso.lib.auxiliaries.ykush_auxiliary.**YkushPortNumberError**

Raised when the port number doesn't exist.

exception pykiso.lib.auxiliaries.ykush_auxiliary.**YkushSetStateError**

Raised when a port couldn't be switched on or off.

exception pykiso.lib.auxiliaries.ykush_auxiliary.**YkushStatePortNotRetrieved**

Raised when the state of a port can't be retrieved.

Warning: This auxiliary has only been tested on the Ykush Device.

Ykush 3 and Ykush XS have not been tested.

A command line to get information about the device or port can be found on the following link : https://github.com/Yepkit/ykush

6.4 Message Protocol

6.4.1 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_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*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

List of possible received messages.

class pykiso.message.**MessageCommandType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

List of commands allowed.

class pykiso.message.**MessageLogType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

List of possible received log messages.

class pykiso.message.**MessageReportType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

List of possible received messages.

class pykiso.message.**MessageType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

List of messages allowed.

class pykiso.message.**TlvKnownTags**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

List of known / supported tags.

6.5 Import Magic

6.5.1 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.

6.5.2 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.

Internally patch the user-configuration if multiple auxiliaries share a same communication channel.

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[NewType(AuxiliaryAlias, str), *AuxiliaryCommon*]

Returns
dictionary with alias as keys and instances as values

classmethod get_aux_by_alias(*alias*)

Return the associated auxiliary instance to the given alias.

Parameters
alias (NewType(AuxiliaryAlias, str)) – auxiliary's alias

Return type
AuxiliaryCommon

Returns
auxiliary instance created by the dymanic loader

classmethod get_aux_config(*name*)

Return the registered auxiliary configuration based on his name.

Parameters
name (NewType(AuxiliaryAlias, str)) – auxiliary's alias

Return type
Dict[str, Any]

Returns
auxiliary's configuration (yaml content)

classmethod get_auxes_alias()

return all created auxiliaries alias.

Return type
List[NewType(AuxiliaryAlias, str)]

Returns
list containing all auxiliaries alias

classmethod get_auxes_by_type(*aux_type*)

Return all auxiliaries who match a specific type.

Parameters
aux_type (Type[*AuxiliaryCommon*]) – auxiliary class type (DUTAuxiliary, CommunicationAuxiliary...)

Return type
Dict[str, *AuxiliaryCommon*]

Returns
dictionary with alias as keys and instances as values

classmethod provide_auxiliaries(*cls*, *config*)

Context manager that registers importable auxiliary aliases and cleans them up at exit.

Parameters
config (ConfigDict) – config dictionary from the YAML configuration file.

Yield
None

Return type
Iterator[None]

classmethod register_aux_con(*config*)

Create import hooks. Register auxiliaries and connectors.

Parameters
config (ConfigDict) – dictionary containing yaml configuration content

Return type
None

6.6 Test Suites

6.6.1 Test Suite

module
test_suite

synopsis
Create a generic test-suite based on the connected modules, and gray test-suite for Message Protocol / TestApp usage.

class pykiso.test_coordinator.test_suite.**BaseTestSuite**(*test_suite_id*, *test_case_id*, *aux_list*, *setup_timeout*, *run_timeout*, *teardown_timeout*, *test_ids*, *tag*, **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"]}
- **tag** (Optional[Dict[str, List[str]]]) – dictionary containing lists of variants and/or test levels when only a subset of tests needs to be executed

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.

Note:

1. Will Load from the given path the integration test modules under test
2. Sort the given test case list by test suite/case id

3. Place Test suite setup and teardown respectively at top and bottom of test case list
4. Add sorted test case list to test suite

---

check_suite_setup_failed(*test*, *result*)

Check if the suite setup has failed and store failed suite id. Search in the global unittest result object, which save all the results of the tests performed up to that point, if the suite setup that ran has failed then store the suite id.

Parameters

- **test** (*BasicTest*) – test to check
- **result** (*BannerTestResult*) – unittest result object

Return type
None

run(*result*, *debug=False*)

Override run method from unittest.suite.TestSuite. Added functionality: Skip suite tests if the parent test suite setup has failed.

Parameters

- **result** (*BannerTestResult*) – unittest result storage
- **debug** (bool) – True to enter debug mode, defaults to False

Return type
BannerTestResult

Returns
test suite result

class pykiso.test_coordinator.test_suite.**BasicTestSuiteSetup**(*test_suite_id*, *test_case_id*, *aux_list*, *setup_timeout*, *run_timeout*, *teardown_timeout*, *test_ids*, *tag*, **args*, ***kwargs*)

Inherit from unittest testCase and represent setup fixture.

Initialize Message Protocol / TestApp 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"]}
- **tag** (Optional[Dict[str, List[str]]]) – dictionary containing lists of variants and/or test levels when only a subset of tests needs to be executed

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*, *tag*, **args*, ***kwargs*)

Inherit from unittest testCase and represent teardown fixture.

Initialize Message Protocol / TestApp 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"]}
- **tag** (Optional[Dict[str, List[str]]]) – dictionary containing lists of variants and/or test levels when only a subset of tests needs to be executed

test_suite_tearDown()

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

class pykiso.test_coordinator.test_suite.**RemoteTestSuiteSetup**(*test_suite_id*, *test_case_id*, *aux_list*, *setup_timeout*, *run_timeout*, *teardown_timeout*, *test_ids*, *tag*, **args*, ***kwargs*)

Inherit from unittest testCase and represent setup fixture when Message Protocol / TestApp is used.

Initialize Message Protocol / TestApp 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"]}

- **tag** (Optional[Dict[str, List[str]]]) – dictionary containing lists of variants and/or test levels when only a subset of tests needs to be executed

test_suite_setUp()

Test method for constructing the actual test suite.

class pykiso.test_coordinator.test_suite.**RemoteTestSuiteTeardown**(*test_suite_id*, *test_case_id*, *aux_list*, *setup_timeout*, *run_timeout*, *teardown_timeout*, *test_ids*, *tag*, **args*, ***kwargs*)

Inherit from unittest testCase and represent teardown fixture when Message Protocol / TestApp is used.

Initialize Message Protocol / TestApp 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"]}
- **tag** (Optional[Dict[str, List[str]]]) – dictionary containing lists of variants and/or test levels when only a subset of tests needs to be executed

test_suite_tearDown()

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

6.7 Test Execution

6.7.1 Execution of tests

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*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

List of possible exit codes

class pykiso.test_coordinator.test_execution.**TestFilterPattern**(*test_file*, *test_class*, *test_case*)

Create new instance of TestFilterPattern(test_file, test_class, test_case)

test_case

Alias for field number 2

test_class

Alias for field number 1

test_file

Alias for field number 0

pykiso.test_coordinator.test_execution.**abort**(*reason=None*)

Quit ITF test and log an error if a reason is indicated and if any errors occurred it logs them.

Parameters
reason (str) – reason to abort, defaults to None

Return type
None

pykiso.test_coordinator.test_execution.**apply_tag_filter**(*all_tests_to_run*, *usr_tags*)

Filter the test cases based on user tags provided via CLI.

Parameters

- **all_tests_to_run** (TestSuite) – a dict containing all testsuites and testcases
- **usr_tags** (Dict[str, List[str]]) – encapsulate user's variant choices

Return type
None

pykiso.test_coordinator.test_execution.**apply_test_case_filter**(*all_tests_to_run*, *test_class_pattern*, *test_case_pattern*)

Apply a filter to run only test cases which matches given expression

Parameters

- **all_tests_to_run** (TestSuite) – a dict containing all testsuites and testcases
- **test_class_pattern** (str) – pattern to select test class as unix filename pattern
- **test_case_pattern** (str) – pattern to select test case as unix filename pattern

Return type
TestSuite

Returns
new test suite with filtered test cases

pykiso.test_coordinator.test_execution.**collect_test_suites**(*config_test_suite_list*, *test_filter_pattern=None*)

Collect and load all test suites defined in the test configuration.

Parameters

- **config_test_suite_list** (List[SuiteConfig]) – list of dictionaries from the configuration file corresponding each to one test suite.
- **test_filter_pattern** (Optional[str]) – optional filter pattern to overwrite the one defined in the test suite configuration.

Raises
pykiso.TestCollectionError – if any test case inside one of the configured test suites failed to be loaded.

Return type
List[*BasicTestSuite*]

Returns
a list of all loaded test suites.

pykiso.test_coordinator.test_execution.**create_test_suite**(*test_suite_dict*)

create a test suite based on the config dict

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

Return type
BasicTestSuite

pykiso.test_coordinator.test_execution.**enable_step_report**(*all_tests_to_run*, *step_report*)

Decorate all assert method from Test-Case.

This will allow to save the assert inputs in order to generate the HTML step report.

Parameters
all_tests_to_run (TestSuite) – a dict containing all testsuites and testcases

Return type
None

pykiso.test_coordinator.test_execution.**execute**(*config*, *report_type='text'*, *report_name=''*, *user_tags=None*, *step_report=None*, *pattern_inject=None*, *failfast=False*)

Create test environment based on test configuration.

Parameters

- **config** (ConfigDict) – dict from converted YAML config file
- **report_type** (str) – str to set the type of report wanted, i.e. test or junit
- **report_name** (str) – name of the junit report
- **user_tags** (Optional[Dict[str, List[str]]]) – test case tags to execute
- **step_report** (Optional[Path]) – file path for the step report or None
- **pattern_inject** (Optional[str]) – optional pattern that will override test_filter_pattern for all suites. Used in test development to run specific tests.
- **failfast** (bool) – stop the test run on the first error or failure.

Return type
int

Returns
exit code corresponding to the result of the test execution (tests failed, unexpected exception, . . .)

pykiso.test_coordinator.test_execution.**failure_and_error_handling**(*result*)

provide necessary information to Jenkins if an error occur during tests execution

Parameters
result (TestResult) – encapsulate all test results from the current run

Return type
int

Returns
an ExitCode object

pykiso.test_coordinator.test_execution.**parse_test_selection_pattern**(*pattern*)

Parse test selection pattern from cli.

For example: test_file.py::TestCaseClass::test_method

Parameters
pattern (str) – test selection pattern

Return type
TestFilterPattern

Returns
pattern for file, class name and test case name

6.8 Test-Message Handling

6.8.1 Handle common communication with device under test

When using a Remote TestCase/TestSuite, the integration test framework handles internal messaging and control flow using a message format defined in pykiso.Message.

pykiso.test_message_handler defines the messaging protocol from a behavioral point of view.

module
test_message_handler

synopsis
default communication between TestManagement and DUT.

pykiso.test_coordinator.test_message_handler.**test_app_interaction**(*message_type*, *timeout_cmd=5*)

Handle test app basic interaction depending on the decorated method.

Parameters

- **message_type** (*MessageCommandType*) – message command sub-type (test case/suite run, setup, teardown. . . .)
- **timeout_cmd** (int) – timeout in seconds for auxiliary run_command

Return type
Callable

Returns
inner decorator function

6.9 Test Results

6.9.1 XML test result for JUnit support

module
xml_result

synopsis
overwrite xmlrunner.result to add the test IDs into the xml report.

class pykiso.test_result.xml_result.**TestInfo**(*test_result*, *test_case*, *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** (Optional[Tuple[Type[BaseException], BaseException, TracebackType]]) – exception information of an error that was raised during the test
- **subTest** (TestCase) – optional subTest that was run
- **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_result.xml_result.**XmlTestResult**(*stream=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>*, *descriptions=True*, *verbosity=0*, *elapsed_times=True*, *properties=None*, *infoclass=<class 'pykiso.test_result.xml_result.TestInfo'>*)

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

Initialize both base classes with the appropriate parameters.

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** (Optional[Dict[str, Any]]) – junit testsuite properties
- **infoclass** (_TestInfo) – class containing the test information

addSuccess(*test*)

Calls only _XMLTestResult's addSuccess as BannerTestResult's appends TestCase instances to the successes list, but _XMLTestResult wraps them in TestInfo instances with additional attributes.

Parameters
test (TestCase) – current succeeded TestCase.

Return type
None

report_testcase(*xml_testsuite*, *xml_document*)

Appends a testcase section to the XML document.

6.9.2 Text Test Result with banners

module
text_result

synopsis
implements a test result that displays the test execution wrapped in banners.

class pykiso.test_result.text_result.**BannerTestResult**(*stream*, *descriptions*, *verbosity*)

TextTestResult subclass showing results wrapped in banners (frames).

Constructor. Initialize TextTestResult and the banner's width.

The banner's width is set to the terminal size. In the case where this fails the fallback width corresponds to the default width of a Jenkins "console".

Parameters

- **stream** (TextIO) – stream to print the result information (default: stderr)
- **descriptions** (bool) – unused (required for TextTestResult)
- **verbosity** (int) – unused (required for TextTestResult)

addError(*test*, *err*)

Set the error flag when an error occurs in order to get the individual test case result.

Parameters

- **test** (Union[*BasicTest*, *BaseTestSuite*]) – running testcase that errored out
- **err** (Tuple[Type[BaseException], BaseException, TracebackType]) – tuple returned by sys.exc_info

Return type
None

addFailure(*test*, *err*)

Set the error flag when a failure occurs in order to get the individual test case result.

Parameters

- **test** (Union[*BasicTest*, *BaseTestSuite*]) – testcase which failure will be reported
- **err** (Tuple[Type[BaseException], BaseException, TracebackType]) – tuple returned by sys.exc_info

Return type
None

addSubTest(*test*, *subtest*, *err*)

Set the error flag when an error occurs in a subtest.

Parameters

- **test** (Union[*BasicTest*, *BaseTestSuite*]) – running testcase
- **subtest** (_SubTest) – subtest runned
- **err** (Tuple[Type[BaseException], BaseException, TracebackType]) – tuple returned by sys.exc_info

Return type
None

addSuccess(*test*)

Add a testcase to the list of succeeded test cases.

Parameters
test (Union[*BasicTest*, *BaseTestSuite*]) – running testcase that succeeded

Return type
None

getDescription(*test*)

Return the entire test method docstring.

Parameters
test (Union[*BasicTest*, *BaseTestSuite*]) – running testcase

Return type
str

Returns
the wrapped docstring

printErrorList(*flavour*, *errors*)

Print all errors at the end of the whole tests execution.

Overwrites the unittest method to have a nicer output.

Parameters

- **flavour** (str) – failure reason
- **errors** (List[tuple]) – list of failed tests with their error message

startTest(*test*)

Print a banner containing the test information and the test method docstring when starting a test case.

Parameters
test (Union[*BasicTest*, *BaseTestSuite*]) – testcase that is about to be run

Return type
None

stopTest(*test*)

Print a banner containing the test information and its result.

Parameters
test (Union[*BasicTest*, *BaseTestSuite*]) – terminated testcase

Return type
None

class pykiso.test_result.text_result.**ResultStream**(*file*)

Class that duplicates sys.stderr to a log file if a file path is provided.

When passed to a TestRunner or a TestResult, this allows to display the information from the test run in the log file.

Initialize the streams.

Parameters
file (Union[str, Path, None]) – file where stderr should be written.

close()

Close or restore each stream.

6.9.3 Create a Step report

module
assert_step_report

synopsis
Provide a detailed test view containing: - test name - test description - date of execution + elapsed time - information gathered during test - assertion detail: data_in, variable, name of the data_in, expected, message

pykiso.test_result.assert_step_report.**MUTE_CONTENT_ASSERTION = ['assertIsInstance', 'assertNotIsInstance']**

content all assertion methods where the checked value is not shown

class pykiso.test_result.assert_step_report.**StepReportData**(*header=<factory>*, *message=''*, *success=True*, *last_error_message=''*, *current_table=None*)

pykiso.test_result.assert_step_report.**add_retry_information**(*test*, *result_test*, *retry_nb*, *max_try*, *exc*)

Add information in the step report if a test fails and is retried.

Parameters

- **test** (*BasicTest*) – test failed
- **result_test** (bool) – result of the tests of the class before the retry
- **retry_nb** (int) – number of the current try
- **max_try** (int) – maximum tries that will be done
- **exc** (Exception) – exception caught

Return type
None

pykiso.test_result.assert_step_report.**assert_decorator**(*assert_method*)

Decorator to gather assertion information

- **MyTestClass**
  - header: additional data
  - description: test purpose
  - file_path: test location

- time_result: start/end/elapsed time
- **test_list: store the steps result**
  - setUp : list of assert
  - test_run: list of assert

Note: header is based on the variable `step_report_header` and description is based on the docstring of the test_run method

---

```
MyTest(pykiso.BasicTest):
    def test_run(self):
        '''Here is my test description'''
        self.step_report.header["Voltage"] = 5
```

Parameters
func – function to decorate (expected assert method)

return: The func output if it exists. Otherwise, None

pykiso.test_result.assert_step_report.**determine_parent_test_function**(*test_name*)

Determine the parent test function.

This function attached the nested assertion to the correct parent test function.

Parameters
test_name (str) – current test function

Return type
str

Returns
parent test function

pykiso.test_result.assert_step_report.**generate_step_report**(*test_result*, *output_file*)

Generate the HTML step report based on Jinja2 template

Parameters

- **test_result** (Union[*BannerTestResult*, *XmlTestResult*]) – Result of tests to generate the report from
- **output_file** (str) – Report output file path

Return type
None

pykiso.test_result.assert_step_report.**is_test_success**(*test*)

Check if test was successful.

Parameters
tests – test

Return type
bool

Returns
True if each step in a test was successful and no unexpected error was raised else False

CHAPTER

SEVEN

ADDITIONAL TOOLS

7.1 Pykiso to Pytest

If you want to use as testing framework pytest while using the test-setup generator of pykiso (YAML-based generation of auxiliaries and communication channels), you can use the pykiso_to_pytest command to convert existing pykiso yaml configuration into a pytest fixture.

```
pykitest examples/dummy.yaml
```

Example can be found inside examples/pytest.

7.2 Show and export test suite tags

The `pykiso-tags` CLI utility takes as input YAML configuration files and passively loads all the specified test suites in order to create a test information table.

This table contains the number of test cases that will be run when providing this configuration file to pykiso and the test tags that are specified in each test suite.

Another options can be specified and the table can be exported to various formats. See:

```
pykiso-tags --help
```

A minimal invocation of the tool would be:

```
pykiso-tags -c kiso-testing/examples/dummy.yaml
```

Which results in the following output:

```
Start analyzing provided configuration file...

All valid configuration files have been processed successfully:


| File name   |   Number of tests | variant   | branch_level   |

| dummy.yaml  |                 7 | variant1  | daily          |
|             |                   | variant3  | nightly        |
```

Note: If an environment variable without a default value is not found, the tool will skip the configuration file. Also, configuration files for Robot framework tests are not supported yet.

7.3 Export results on TestRail

The `testrail` CLI utility takes your Pykiso Junit report and export them on TestRail.

7.3.1 Upload your results

To upload your results on TestRail users have to follow the command :

```
testrail --user USER_ID --password MY_API_KEY --url https\\:testrail_server.com upload --
↪run--name "sample run" --project "project sample" --suite "suite 1" --milestone
↪"sample 2023" --results path/to/reports/folder --tag VTestId --custom-field custom_
↪vteststudio_id
```

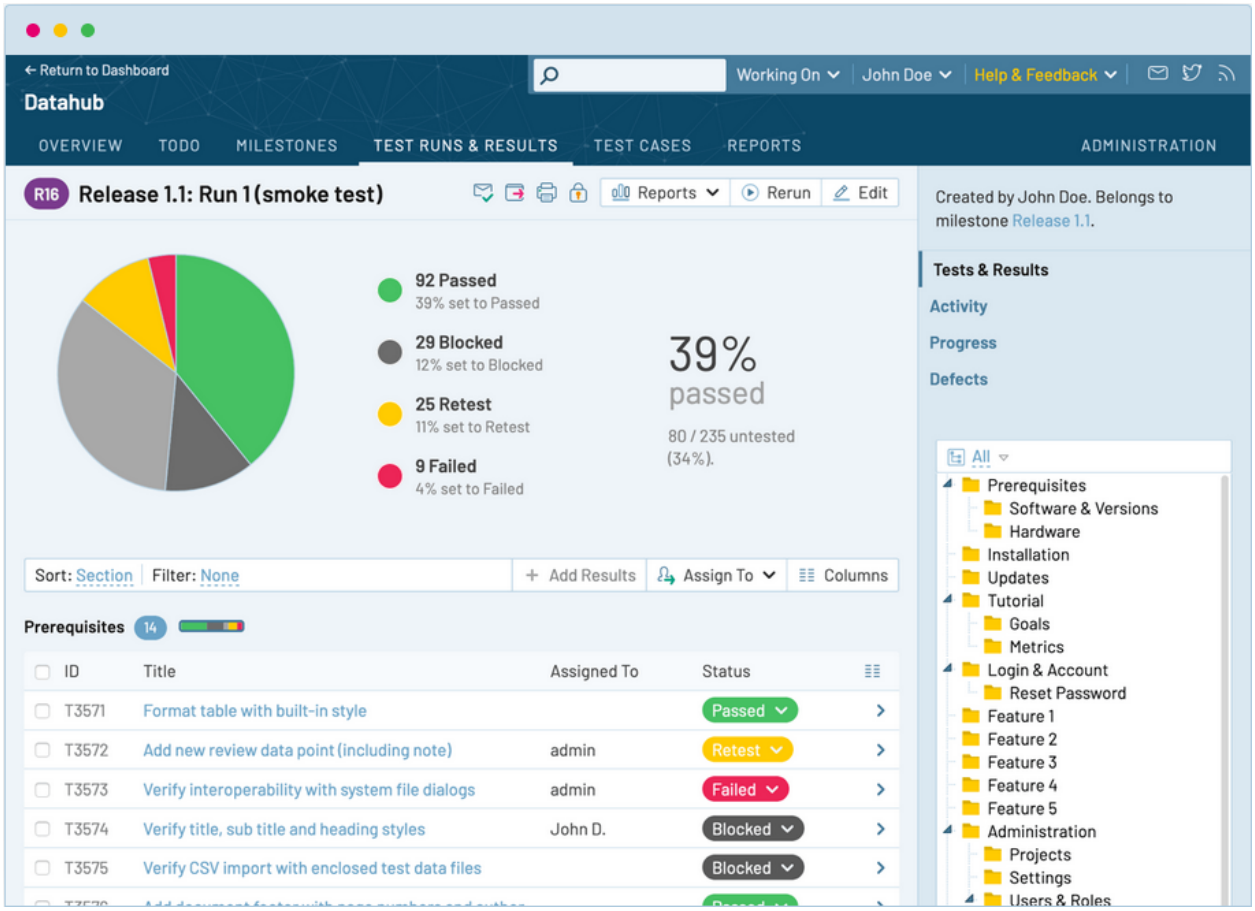
Options:

--user TEXT	TestRail user id [required]
--password TEXT	Valid TestRail API key (if not given ask at command prompt level) [optional]
--url TEXT	URL of TestRail server [required]
-n, --run-name TEXT	How to name the created run on TestRail [required]
-p, --project TEXT	TestRail's project name [required]
-s, --suite TEXT	TestRail's suite name [required]
-m, --milestone TEXT	TestRail's milestone name [required]
-r, --results PATH	full path to the folder containing the JUNIT reports [required]
--tag TEXT	attribute in JUNIT report use to store requirements ids [optional] [default value: VTestId]
--custom-field TEXT	TestRail's case custom field use to store the requirement id [optional][default value: custom_vteststudio_id]
--help	Show this message and exit.

The above command will create a brand new run on TestRail side with the following values :

- associated the run to the TestRail project -> "project sample"
- for a suite called -> "suite 1"
- define for the milestone -> "sample 2023"
- upload all the results contained in -> path/to/reports/folder
- all TestRail's ids will be found under tag "VTestId" in each JUNIT report
- the ids under tag "VTestId" will be associated to a custom id created on TestRail case side call custom_vteststudio_id

After a successful command a new run is added :



7.3.2 Useful commands

Find below additional commands use to display different entities of TestRail (suite, case, project, milestone…).

Returns the list of available projects :

```
testrail --user USER_ID --password MY_API_KEY --url https\\:testrail_server.com projects
```

Returns a list of all the test suites contained in a given project.

```
testrail --user USER_ID --password MY_API_KEY --url https\\:testrail_server.com suites --
↪project "super project"
```

Returns a list of all the cases contained in a given project.

```
testrail --user USER_ID --password MY_API_KEY --url https\\:testrail_server.com cases --
↪project "super project"
```

Returns a list of all the runs contained in a given project.

```
testrail --user USER_ID --password MY_API_KEY --url https\\:testrail_server.com runs --
↪project "super project"
```

Returns the list of all milestones contained in a given project.

```
testrail --user USER_ID --password MY_API_KEY --url https\\:testrail_server.com␣
↪milestones --project "super project"
```

CHAPTER

EIGHT

PYTEST INTEGRATION

Pytest is one of the most mature and widely used testing frameworks when it comes to Python.

It comes with plenty of plugins that make it highly customizable. This is why Pykiso can also be integrated within pytest as a plugin.

8.1 Using Pykiso with Pytest

The pykiso-pytest plugin allows to run pykiso test suites with pytest without any migration effort. Simply run `pytest my_test_configuration_file.yaml` and enjoy!

Still, to benefit from all of pytest's features, your existing test suites can be migrated from the pykiso (unittest) style to the pytest style.

8.1.1 Running Pykiso tests with pytest

Let us take as example the following test configuration file named `example.yaml`:

```yaml
auxiliaries:
  aux1:
    connectors:
      com: chan1
    config: null
    type: pykiso.lib.auxiliaries.dut_auxiliary:DUTAuxiliary
  aux2:
    connectors:
      com: chan2
    type: pykiso.lib.auxiliaries.dut_auxiliary:DUTAuxiliary
connectors:
  chan1:
    config: null
    type: pykiso.lib.connectors.cc_example:CCExample
  chan2:
    type: pykiso.lib.connectors.cc_example:CCExample
test_suite_list:
- suite_dir: ./
  test_filter_pattern: 'test_suite.py'
  test_suite_id: 1
```

Along with the following test case as a python module named `test_suite.py`, located in the same folder as the test configuration file:

```
import pykiso
from pykiso.auxiliaries import aux1, aux2

@pykiso.define_test_parameters(suite_id=1, aux_list=[aux1, aux2])
class SuiteSetup(pykiso.BasicTestSuiteSetup):
    def test_suite_setUp(self):
        pass


@pykiso.define_test_parameters(suite_id=1, aux_list=[aux1, aux2])
class SuiteTearDown(pykiso.BasicTestSuiteTeardown):
    def test_suite_tearDown(self):
        pass


@pykiso.define_test_parameters(suite_id=1, case_id=1, aux_list=[aux1], tag={"variant": [
↪"v2"]})
class MyTest1(pykiso.BasicTest):

    def setUp(self):
        self.side_effect = iter([False, True])

    @pykiso.retry_test_case(max_try=2)
    def test_run(self):
        # the retry feature is also available out of the box
        self.assertTrue(next(self.side_effect))


@pykiso.define_test_parameters(suite_id=1, case_id=2, aux_list=[aux2], tag={"variant": [
↪"v1"]})
class MyTest2(pykiso.BasicTest):
    def test_run(self):
        self.assertTrue(aux2.is_instance)
```

Then, executing `pytest example.yaml -v` will produce the following output:

```
$ pytest example.yaml -v
============================================== test session starts ==============================================
platform win32 -- Python 3.10.11, pytest-7.2.1, pluggy-1.0.0 -- [illegible]\AppData\Local\pypoetry\Cache\virtualenvs\pykiso-fAXnA
4jD-py3.10\Scripts\python.exe
cachedir: .pytest_cache
rootdir: C:\tmp_example
plugins: pykiso-0.21.2, cov-4.0.0, mock-3.10.0
collected 4 items

test_suite.py::SuiteSetup::test_suite_setUp PASSED                                                    [ 25%]
test_suite.py::MyTest1::test_run PASSED                                                               [ 50%]
test_suite.py::MyTest2::test_run PASSED                                                               [ 75%]
test_suite.py::SuiteTearDown::test_suite_tearDown PASSED                                              [100%]

============================================== 4 passed in 0.28s ==============================================
```

8.1.2 Running pytest tests with pykiso test auxiliaries

One of Pytest's main features are test fixtures. These allow you to create a context for your test cases, by specifying a test setup, teardown or more generally by initializing resources that are then provided to the test cases.

Pykiso's pytest plugin uses fixtures to provide test auxiliaries to test cases. Similarly to the importable auxiliary aliases in pykiso, the plugin will create fixtures available under this alias.

For example, taking as reference the previous test configuration file and pykiso test suite, the pykiso test case `MyTest2` could be rewritten to:

```
@pytest.mark.tags(variant=["var1"])
def test_mytest2(aux2):
    assert aux2.is_instance == True
```

Just like pykiso, an auxiliary will be started as soon as it is used for the first time. It will then keep running until the end of the test session. To change this behaviour, please refer to *pytest_auxiliary_scope*.

Note: When writing test cases with pytest, always use plain `assert` statements instead of e.g. `assertTrue`. Otherwise pytest does not provide assertion introspection.

8.1.3 Ported pykiso features

Filtering test cases

In order to select a subset of the tests to run, 2 options are available:

- Use pytest's builtin option `-k`. By modifying the previous command to e.g. `pytest example.yaml -v -k MyTest1`, only `MyTest1` will be run.
- Use pykiso's test tags feature. By modifying the previous command to e.g. `pytest example.yaml -v --tags variant=v1`, only `MyTest1` will be skipped as it is only meant for a variant tag `v2`. For more information regarding this pykiso feature, please refer to *Filter the test cases to run with tags*.

Adding test case information to JUnit reports

Assign test requirements to test cases is also supported by the pykiso pytest plugin:

```
@pykiso.define_test_parameters(test_ids={"Variant1": ["Requirement123"]})
class MyTest2(pykiso.BasicTest):
    def test_run(self):
        self.assertTrue(aux2.is_instance)


@pytest.mark.test_ids(Variant1=["Requirement123"])
def test_mytest2(aux2):
    assert aux2.is_instance == True
```

In order to generate a JUnit report for your test session, simply use pytest's built-in option:

```
pytest ./example.yaml --junit-xml=./report.xml
```

8.1.4 Pytest related features

Changing the auxiliary fixtures scope

As stated previously, the test auxiliary fixtures that are generated from the test configuration file as scoped session-widely.

However, this can be changed by setting the value `auxiliary_scope` to one of the following scope values:

- `function`: an auxiliary will be stopped and restarted between each test case
- `class`: an auxiliary will be stopped and restarted between each test class containing test cases
- `module`: an auxiliary will be stopped and restarted between each test module

This value only needs to be added to the pytest configuration file (`pytest.ini` or `pyproject.toml`). For more information refer to the pytest documentation.

Customizing an auxiliary setup/teardown

Pytest allows to 'overwrite' existing features in order to customize then. The fixture only needs to have the same name as the auxiliary defined in your YAML configuration file and will wrap the default behaviour of the auxiliary fixture.

Depending on your use-case, this can be done directly within your test module or within pytest's conftest.py file.

Consider a test module were we would need a *CommunicationAuxiliary* named `my_aux` within the test configuration file to send data over some communication protocol. At startup the auxiliary has to send `hello` and at teardown `goodbye`.

With fixtures, this can easily be achieved with:

```
import pytest

@pytest.fixture(scope="module")
def my_aux(my_aux):
    # my_aux.create_instance() has been called from the base my_aux fixture
    my_aux.send_message(b'hello')
    yield my_aux
    my_aux.send_message(b'goodbye')
    # my_aux.delete_instance() will now be called from the base my_aux fixture
```

Warning: The customized auxiliary fixture cannot have a higher scope than the wrapping one.

Customizing the setup/teardown of all auxiliaries

The pykiso pytest plugin adds two hooks for the startup and teardown of all test auxiliaries: `pykiso.pytest.hooks.pytest_auxiliary_start()` and `pykiso.pytest.hooks.pytest_auxiliary_stop()`. In very specific use cases, these hooks can be implemented within a `conftest.py` file. The execution of registered hook functions will stop when one hook function returns a value different from `None`.

For more information about pytest's hook functions, please refer to *Writing hook functions <https://docs.pytest.org/en/7.1.x/how-to/writing_hook_functions.html>*.

Other useful pytest features

This section aims to provide a non-exhaustive list of potentially interesting features provided by pytest and by 3rd party pytest plugins:

Feature	Plugin	Option
Stop test execution at the first failure	native	-x, –exitfirst
Re-run the last failed tests	native	–lf, –last-failed
Enter the Python debugger (pdb) on test failure	native	-s –pdb
Enter the Python debugger (pdb) on test start	native	-s –trace
Use another Python debugger (here pudb)	native	–pdbcls=pudb.debugger:Debugger
Generate an HTML test report	pytest-html	–html=report.html –self-contained-html
Repeating one or multiple tests for a certain amount of times	pytest-repeat	–count=N
Rerun failing tests for a maximum amount of times	pytest-rerunfailures	–reruns=N

8.2 API Documentation

8.2.1 Test collection and setup

8.2.2 Pykiso specific hooks

8.2.3 Command line interface customization

8.2.4 Logging patching

8.2.5 Markers for test IDs and test tags

8.2.6 Add the test IDs to the test report

8.2.7 Common utilities

CHAPTER

NINE

ROBOT FRAMEWORK INTEGRATION

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...

9.1 How to integrate

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
```

9.2 Ready to Use Auxiliaries

9.2.1 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}
```

9.2.2 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}
```

9.2.3 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
```

9.2.4 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!

9.2.5 Acroname Auxiliary

This plugin can be used to control a acroname usb hub.

Find below an example with all available features:

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

Library    pykiso.lib.robot_framework.acroname_auxiliary.AcronameAuxiliary    WITH NAME␣
↪  AcroAux

*** Variables ***
${NO_ERROR} =    ${0}

*** Test Cases ***

```

(continues on next page)

(continued from previous page)

```
Disable / Enable USB Ports
    [Documentation]    Disable and Enable USB Ports

    Log    Disable all Ports

    FOR    ${index}    IN RANGE    0    4
        Log    Disable USB port ${index}
        ${state}  Set port disable    acroname_aux    ${index}
        Should Be Equal    ${state}    ${NO_ERROR}
    END

    Sleep    1s

    FOR    ${index}    IN RANGE    0    4
        Log    Enable USB port ${index}
        ${state}  Set port disable    acroname_aux    ${index}
        Should Be Equal    ${state}    ${NO_ERROR}
    END

    Sleep    1s

Get Port Current
    [Documentation]    Read usb port current

    Log    Read port current

    FOR    ${index}    IN RANGE    0    4
        ${current}  Get port current    acroname_aux    ${index}    mA
        Log    Current on port ${index} is ${current} mA
    END

    Sleep    1s

Get Port Voltage
    [Documentation]    Read usb port voltage

    Log    Read port voltage

    FOR    ${index}    IN RANGE    0    4
        ${voltage}  Get port voltage    acroname_aux    ${index}    mV
        Log    Voltage on port ${index} is ${voltage} mV
    END

    Sleep    1s

Set Port Current Limit
    [Documentation]    Set usb port current

    Log    Read port current

    FOR    ${index}    IN RANGE    0    4
        Log    Set port current on port ${index} to 500 mA
```

(continues on next page)

(continued from previous page)

```
        ${state}  Set port current limit    acroname_aux    ${index}    ${500}    mA
        Should Be Equal   ${state}    ${NO_ERROR}
    END

    Sleep    1s

Get Port Current Limit
    [Documentation]    Get usb port current limit

    Log    Read port current

    FOR    ${index}    IN RANGE    0    4
        ${current}  Get port current limit    acroname_aux    ${index}    mA
        Log    Port limit on port ${index} is ${current} mA
    END

    Sleep    1s

Set Port Current Limit to max
    [Documentation]    Set usb port current limit

    Log    Read port current

    FOR    ${index}    IN RANGE    0    4
        Log    Set port current on port ${index} to 1500 mA
        ${state}  Set port current limit    acroname_aux    ${index}    ${1500}    mA
        Should Be Equal   ${state}    ${NO_ERROR}
    END

    Sleep    1s
```

To run the available example:

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

9.2.6 Record Auxiliary

Auxiliary used to record a connectors receive channel which are configured in the yaml config. The library needs then only to be loaded. See example below:

config.yaml:

```yaml
auxiliaries:
  record_aux:
    connectors:
      com: rtt_channel
    config:
      # When is_active is set, it actively polls the connector. It demands if
      # the used connector needs to be polled actively.
      is_active: False # False because rtt_channel has its own receive thread
```

(continues on next page)

(continued from previous page)

```
    type: pykiso.lib.auxiliaries.record_auxiliary:RecordAuxiliary

connectors:
  rtt_channel:
    config:
      chip_name: "STM12345678"
      speed: 4000
      block_address: 0x12345678
      verbose: True
      tx_buffer_idx: 1
      rx_buffer_idx: 1
      # Path relative to this yaml where the RTT logs should be written to.
      # Creates a file named rtt.log
      rtt_log_path: ./
      # RTT channel from where the RTT logs should be read
      rtt_log_buffer_idx: 0
      # Manage RTT log CPU impact by setting logger speed. eg: 100% CPU load
      # default: 1000 lines/s
      rtt_log_speed: null
    type: pykiso.lib.connectors.cc_rtt_segger:CCRttSegger

test_suite_list:
- suite_dir: test_record
  test_filter_pattern: '*.py'
  test_suite_id: 1
```

Robot file:

```
*** Settings ***
Documentation   Robot framework Demo for record auxiliary

# Library import will start recording
Library    pykiso.lib.robot_framework.record_auxiliary.RecordAuxiliary

*** Keywords ***

*** Test Cases ***

Test Something
    [Documentation]    Record channel in the background

    Sleep    5s
```

To run the available example:

```
cd examples
robot robot_test_suite/test_record/
```

9.2.7 UDS Auxiliary

To run the example:

```yaml
auxiliaries:
  uds_aux:
    connectors:
        com: can_channel
    config:
      odx_file_path: null
      request_id : 0x123
      response_id : 0x321
      uds_layer:
        transport_protocol: 'CAN'
        # p2_can specifies receive time outs in seconds
        p2_can_client: 5
        p2_can_server: 1
      tp_layer:
        req_id: 0xAB
        res_id: 0xAC
        addressing_type: 'NORMAL'
        n_sa: 0xFF
        n_ta: 0xFF
        n_ae: 0xFF
        m_type: 'DIAGNOSTICS'
        discard_neg_resp: False
    type: pykiso.lib.auxiliaries.udsaux.uds_auxiliary:UdsAuxiliary
connectors:
  can_channel:
    config:
      interface : 'pcan'
      channel: 'PCAN_USBBUS1'
      state: 'ACTIVE'
    type: pykiso.lib.connectors.cc_pcan_can:CCPCanCan
```

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

Note: All return types between "ITF" and "Robot" auxiliary's version stay identical! Only "Send uds raw" keywords return a list instead of bytes!

Please find below a complete correlation table:

ITF method	robot equivalent	Parameter 1	Parameter 2	Parameter 3	Parameter 4
send_uds_raw	Send uds raw	message	aux alias	timeout	
send_uds_config	Send uds config	message	aux alias	timeout	

Robot file:

```
*** Settings ***
Documentation   Robot framework Demo for uds auxiliary implementation
Library    pykiso.lib.robot_framework.uds_auxiliary.UdsAuxiliary    WITH NAME    UdsAux
Library    Collections
```

(continues on next page)

(continued from previous page)

```
*** Test Cases ***

Go in default session
    [Documentation]    Send diagnostic session control request session
    ...                default using Send uds raw

    ${response} =  Send uds raw    \x10\x01    uds_aux

    Log  ${response}

Use tester present sender
    [Documentation]    If no communication is exchanged with the client for more than 5
    ...                seconds the control unit automatically exits the current session and
    ...                returns to the "Default Session" back, and might go to sleep mode.
    ...
    ...                To avoid this issue, if test steps take too long between uds commands,
    ...                the tester present sender can be used. It will send
    ...                at a defined period a Tester Present, to signal to the device that
    ...                the client is still present.

    Start tester present with    1  seconds    uds_aux
    ${response} =  Send uds raw    \x10\x03    uds_aux
    Stop tester present
```

To run the available example:

```
cd examples
robot robot_test_suite/test_uds/
```

9.3 Robot Framework API Library Documentation

9.3.1 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

9.3.2 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

9.3.3 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.

`clear_buffer(aux_alias)`

Clear buffer from old stacked objects

Return type
None

`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** (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

start_recording_received_messages()

Start recording received com_aux messages

Return type
None

stop_recording_received_messages()

Stop recording received com_aux messages

Return type
None

9.3.4 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*)

Execute the corresponding test fixture using Test App communication protocol.

Parameters

- **command_type** (str) – message command sub-type
- **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

Raises

- **TypeError** – if the given command type doesn't exist

- **Assertion** – if an aknowledgment is not received or the report status is failed.

Return type
None

9.3.5 Proxy auxiliary plugin

module
proxy_auxiliary

synopsis
implementation of existing ProxyAuxiliary for Robot framework usage.

class pykiso.lib.robot_framework.proxy_auxiliary.**MpProxyAuxiliary**

Robot framework plugin for MpProxyAuxiliary.

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

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

9.3.6 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 type
str

return: list of off errors

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** (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

9.3.7 acroname auxiliary plugin

module
acroname_auxiliary

synopsis
implementation of existing AcronameAuxiliary for Robot framework usage.

class pykiso.lib.robot_framework.acroname_auxiliary.**AcronameAuxiliary**

Robot framework plugin for AcronameAuxiliary.

Initialize attributes.

get_port_current(*aux_alias*, *port*, *unit='A'*)

Get the current through the power line for selected usb port.

Parameters

- **aux_alias** (str) – auxiliary's alias
- **port** (int) – the USB port number
- **unit** (str) – unit of the result in "uA", "mA" or "A". Default "A"

Return type
Optional[float]

Returns
port current for given unit. None if unit is not supported.

get_port_current_limit(*aux_alias*, *port*, *unit='A'*)

Get the current limit for the port.

Parameters

- **aux_alias** (str) – auxiliary's alias
- **port** (int) – the USB port number
- **unit** (str) – unit of the result in "uA", "mA" or "A". Default "A"

Return type
Optional[float]

Returns
port current limit for given unit. None if unit is not supported.

get_port_voltage(*aux_alias*, *port*, *unit='V'*)

Get the voltage of the selected usb port.

Parameters

- **aux_alias** (str) – auxiliary's alias
- **port** (int) – the USB port number

- **unit** (str) – unit of the result in “uV”, “mV” or “V”. Default “V”

Return type
Optional[float]

Returns
port voltage for given unit. None if unit is not supported.

set_port_current_limit(*aux_alias*, *port*, *amps*, *unit='A'*)

Set the current limit for the port. If the set limit is not achievable, devices will round down to the nearest available current limit setting.

Parameters

- **aux_alias** (str) – auxiliary’s alias
- **port** (int) – the USB port number
- **amps** (float) – value for port current to set in “uA”, “mA” or “A”. Default “A”
- **unit** (str) – unit for the value to set. Default Ampere

Return type
None

set_port_disable(*aux_alias*, *port*)

Disable power and data lines for a USB port.

Parameters

- **aux_alias** (str) – auxiliary’s alias
- **port** (int) – the USB port number

Return type
int

Returns
brainstem error code. 0 if no error.

set_port_enable(*aux_alias*, *port*)

Enable power and data lines for a USB port.

Parameters

- **aux_alias** (str) – auxiliary’s alias
- **port** (int) – the USB port number

Return type
int

Returns
brainstem error code. 0 if no error.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

H

N

O

P

Q

R

S

T

U

W

X

Y