nanaimo Documentation

Release 0.2.5

Amazon.com

Mar 15, 2020

Contents

To illustrate how to use Nanaimo we are going to work with a specific scenario. We'll be writing tests to verify an IMU using a pan-tilt test rig.

Fig. 1: Example scenario using Nanaimo to test an IMU.

Our test will run the following steps:

1. Upload the firmware using a serial bootloader.
2. Start the pan/tilt test rig,
3. Start capturing IMU readings.
4. Stop the pan/tilt test rig.
5. Validate the data.

As you read this document you might wonder if the amount of rigor is necessary given the sometimes trivial nature of tests for components and sub-systems. One might be tempted to write a bash script or a simpler python script with no formal structure.

Don't normalize deviance....

The authors of Nanaimo aver that the real-world is complicated and messy. When you include physical devices in software workflows you mix something that is inherently chaotic with something that assumes everything it does is absolutely deterministic. Dealing with edge cases and ensuring repeatability turns unstructured code into a complex mess. With unmanaged complexity comes unexpected behavior and when the unexpected behavior is tolerated a normalization of deviance sets in making the tests worse than nothing.

Nanaimo seeks to organize complexity within `Fixtures` presenting a simplified API to test cases and allowing for a verification syntax that is concise and readable. Furthermore, the architecture enables and encourages fixture reuse supporting DRY principles.

To summarize; as you read this guide keep the following goals in mind:

> **Encapsulate complexity in Fixtures. Write simple and clear test cases. Don't ignore failures.**

CHAPTER 1

Taxonomy

Let's take a moment to review some important concepts and terms we'll use in our demonstration below. There are two main things you will build using Nanaimo:

1. test cases
2. fixtures

We'll review both of these in the next two sections and will cover *nait (NanAimo Interactive Terminal)* later in this document.

1.1 Test Fixtures

Mechanical, material, and electrical engineers may be confused by the term "fixture" as used by software engineers. The former group uses the term to describe a physical adapter holding test material in a precise and repeatable position within a test instrument. For software engineers the term is similar but purely logical. For example, JUnit describes a fixture as something that ensures "there is a well known and fixed environment in which tests are run so that results are repeatable[2]." The term "environment" is the key here. Tests must be run in a well-known and repeatable environment to be valid. The job of Nanaimo fixtures are to validate and adapt the software testing environment to promote consistency and reliability.

Note: A quick note on "instruments" which is another taxonomic term found in Nanaimo. A Nanaimo instrument is simply a flavor of Fixture that either manipulates or measures a test subject. Like the UTM in figure 1.2, instruments sometimes need other fixtures which is why Nanaimo `Fixtures` are composable.

---

[1] Wikipedia Article, "Test Fixture": https://en.wikipedia.org/wiki/Test_fixture
[2] The JUnit 4 wiki

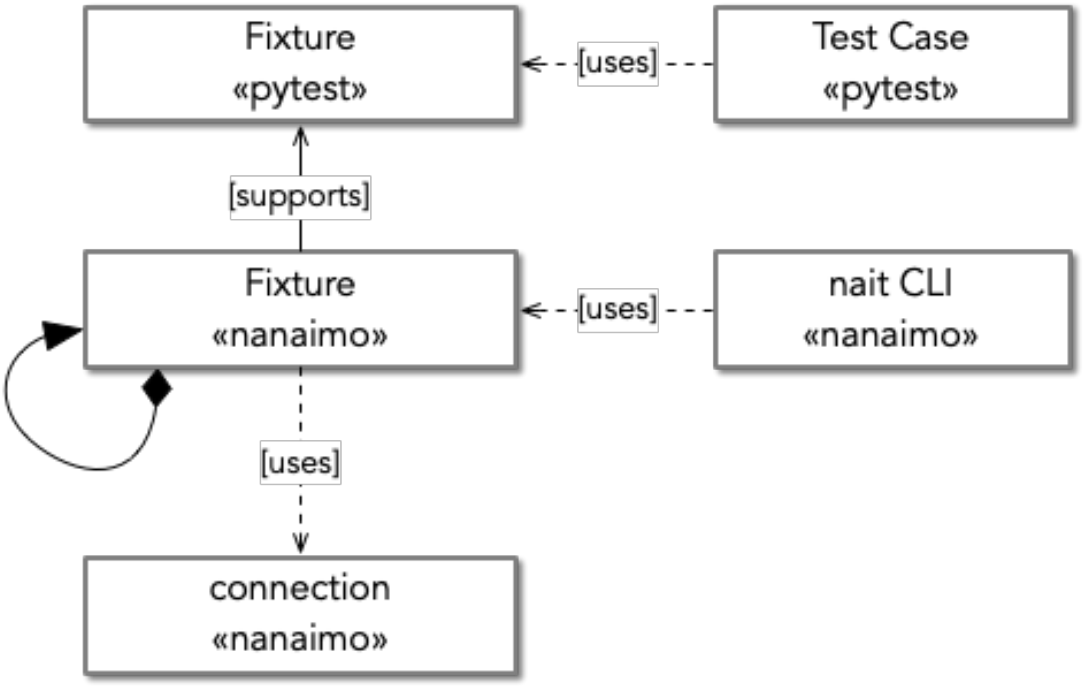

Fig. 1.1: The major concepts in Nanaimo.

Fig. 1.2: A test fixture used to hold material under test in a UTM (Universal Testing Machine)[1].

1.2 Test Cases

A test case is the xunit[3] term for a single pytest test. For example:

```
import pytest
from nanaimo import assert_success

@pytest.mark.asyncio
async def test_eating_dessert(nanaimo_bar):
    """
    Get a Nanaimo bar and eat it!
    """
    assert_success(await nanaimo_bar.gather()).eat()
```

In this example the "test case" is the `test_eating_dessert` method. Note the use of *`nanaimo_bar`*. This is a *`Nanaimo Fixture`* acting as a Pytest fixture. If you are unfamiliar with Pytest fixtures you should review their documentation at this point.

Nanaimo defines HIL (Hardware-In-the-Loop) test cases as pytests that use *`Fixtures`* to modify, manipulate, and observe hardware available in the test environment. Figure 1.3 shows the typical structure of this type of test case. We'll discuss each phase in this section of the document and then provide a full example in a following section.

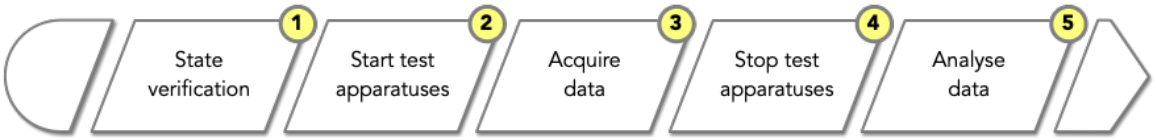

Fig. 1.3: Typical phases of a HIL unit test.

1.2.1 Phase 1. State Verification

The first thing most HIL (Hardware-In-the-Loop) tests will do is to verify the state of the required *`fixtures`*. Be careful with this phase and remember the opening admonition, "Encapsulate complexity in fixtures. Write simple test cases. Don't ignore failures". Well designed fixtures should provide internal consistency checks and fail when the fixture is improperly used rather than requiring each test case to perform fixture setup. Still, not all required state is generic so there are sometimes checks like, "Is this fixture already powered up? If so reboot it." or "Is this instrument calibrated? If not then run the calibration procedure." As both examples provided suggest, this phase is often an optimization used to skip lengthy steps that may be unnecessary. Where initialization steps can take several minutes or more it becomes a necessary optimization to enable a large number of small and simple tests to be written without creating test suites that take hours to complete.

Note: Software unittest best practices generally posit that each test should be as narrow as possible ideally testing everything in the most independent and granular manner possible. Nanaimo seeks to enable this methodology when using hardware test apparatuses by allowing for highly intelligent *`Fixtures`* to optimize the test rig turn around time and to ensure all tests can begin in a known state. For those of you interested in writing more "sociable" tests keep an eye on Issue 74 which would enable this pattern.

---

[3] Wikipedia Article, "xUnit": https://en.wikipedia.org/wiki/XUnit

1.2.2 Phases 2, 3, & 4. Start/Acquire/Stop

The next three steps are sometimes carefully sequenced and sometimes run concurrently. The logic here should be obvious: first you start the flow of data, you sample this data, and then you stop the flow of data. Subtleties do arise especially when you need to capture data that is only available as part of the startup sequence of a device (e.g. when you first power it on). When this data is required you typically modify the sequence to be:

> start acquisition > start the fixture > stop the fixture > stop acquisition

You should prefer the form show in in Figure 1.3 since startup and shutdown are typically special cases.

1.2.3 Phase 5. Analyse

Now we get to the pure software part of the test. Nanaimo considers it good form to defer analysing the data until after all fixtures have completed. This makes for two classes of failure:

1. Data acquisition failure
2. System performance failure

Data acquisition failures should always be treated as unknown failures which are bugs in the test cases and/or test fixtures themselves. Well designed tests should be deterministic and you should always expect to acquire the data you need to analyse a system under test. If you really want to argue the point you can simply reclassify induced hardware failures as the data itself to understand our argument. For example:

```
# Acquire data
assert_success(await my_fixture.gather(cmd='ping'))
assert_success(await my_fixture.gather(cmd='halt'))
with pytest.raises(DeviceUnavailableError):
    await my_fixture.gather(cmd='ping')
```

...where the "data" is "did the device become unreachable?" which is automatically analysed by `pytest.raises()`.

It's more typical, however, to acquire a data-set like logs or sensor data that is then processed comparing it to the required performance of the system under test. Failed assertions here are bugs or regressions in the system itself. It is just this scenario that our example project will explore so let's get started.

CHAPTER 2

Setting up your Project

We won't go into detail on how to setup a Python project but we'll add a few things to make pytest happy by default (none of this setup is specific to Nanaimo). Start by creating the following directories and files:

```
+ myproject
|
|   + test
|   |   test_stuff.py
|   |   conftest.py
|
| tox.ini
```

In `tox.ini` add the following:

```
[pytest]
log_cli = true
log_cli_level = DEBUG
log_format = %(asctime)s %(levelname)s %(name)s: %(message)s
log_date_format = %Y-%m-%d %H:%M:%S
```

This will make any use of `logging.Logger` log to the console when running tests.

Next setup a virtual environment. Again, this isn't a requirement for Nanaimo but it is a best practice especially when playing around with a package like we're doing here:

```
cd myproject
virtualenv .pyenv
source .pyenv/bin/activate
```

Finally, add Nanaimo:

```
pip install nanaimo
```

If you want to run Nanaimo from source you can also do:

```
pip install -e /path/to/nanaimo/
```

You may also want to use instruments that have other dependencies on the test environment but we'll discuss this more in the section on writing your own *Nanaimo Fixture*.

Finally, let's add the "hello world" of Nanaimo, the nanaimo-bar fixture test, to `test_stuff.py`. See *the nanaimo-bar example* above for this example. You should be able to run this test now:

```
pytest
```

If you configured the tox pytest section for logging you'll see this output:

```
------------------- live log sessionstart -------------------
collected 1 item

test/test_foo.py::test_eating_dessert
----------------------- live log setup -----------------------
2019-11-18 10:28:58 DEBUG asyncio: Using selector: KqueueSelector
2019-11-18 10:28:58 DEBUG asyncio: Using selector: KqueueSelector
----------------------- live log call ------------------------
2019-11-18 10:28:58 INFO nanaimo_bar: don't forget to eat your dessert.
2019-11-18 10:28:58 INFO nanaimo_bar: Nanaimo bars are yummy.
PASSED
```

Now list your available pytest fixtures:

```
pytest --fixtures
```

You'll see sections with titles like `fixtures defined from nanaimo...`. For example:

```
----------------------- fixtures defined from nanaimo.pytest.plugin -------------------
↪------
nanaimo_fixture_manager
    Provides a default :class:`FixtureManager <nanaimo.fixtures.FixtureManager>` to a␣
↪test.

    .. invisible-code-block: python

        import nanaimo
        import nanaimo.fixtures

    .. code-block:: python

        def test_example(nanaimo_fixture_manager: nanaimo.Namespace) -> None:
            common_loop = nanaimo_fixture_manager.loop

    :param pytest_request: The request object passed into the pytest fixture factory.
    :type pytest_request: _pytest.fixtures.FixtureRequest
    :return: A new fixture manager.
    :rtype: nanaimo.fixtures.FixtureManager
```

If you do `pytest --help` you'll see the arguments listed for your Nanaimo fixtures. For example

```
nanaimo_instr_bk_precision:
--bk-port=BK_PORT     The port the BK Precision power supply is connected to. Set
                        NANAIMO_BK_PORT in the environment to override default.
--bk-command=BK_COMMAND, --BC=BK_COMMAND
```

(continues on next page)

(continued from previous page)

```
                        command
--bk-command-timeout=BK_COMMAND_TIMEOUT
                        time out for individual commands. Set NANAIMO_BK_COMMAND_
↪TIMEOUT in the
                        environment to override default.
--bk-target-voltage=BK_TARGET_VOLTAGE
                        The target voltage Set NANAIMO_BK_TARGET_VOLTAGE in the␣
↪environment to
                        override default.
--bk-target-voltage-threshold-rising=BK_TARGET_VOLTAGE_THRESHOLD_RISING
                        Voltage offset from the target voltage to trigger on when the␣
↪voltage is
                        rising. Set NANAIMO_BK_TARGET_VOLTAGE_THRESHOLD_RISING in the
                        environment to override default.
--bk-target-voltage-threshold-falling=BK_TARGET_VOLTAGE_THRESHOLD_FALLING
                        Voltage offset from the target voltage to trigger on when the␣
↪voltage is
                        falling. Set NANAIMO_BK_TARGET_VOLTAGE_THRESHOLD_FALLING in␣
↪the
                        environment to override default.
```

These are defined by the Nanaimo fixture itself in the *on_visit_test_arguments* hook and can be overridden using explicit commandline parameters to pytest or by passing in overrides in your pytests to the *Fixture.gather()* method. The base configuration should come from defaults either in `etc/nanaimo.cfg` or in a config file specified by `--rcfile`.

We'll cover configuration in a later section. For now we'll pretend we configured everything already so we can jump into the code and work back to the configuration.

CHAPTER 3

Writing Pytests with Hardware-In-the-Loop

Let's get started with the simplest test to analyse but, perhaps, the most complex to automate; The firmware update.

3.1 Firmware Update Pt.1

```
from nanaimo import assert_success

@pytest.mark.asyncio
async def test_upload_firmware(nanaimo_arguments, nanaimo_cmd):
    """
    This test requires that a (fictitious) utility 'upload_firmware' is available in
    the environment and that it takes the arguments 'firmware path' and 'serial port'
    as its arguments.
    """
    upload_command = 'upload_firmware {imu_firmware} {imu_port}'.format(
                        **vars(nanaimo_arguments)
                    )
    assert_success(await nanaimo_cmd.gather(cmd_shell=upload_command))
```

So everything that is interesting (read: complex) about this test is hidden down in our fictitious "upload_firmware" program. Our earlier assertion that this was difficult to automate seems bogus. Regardless we've verified that the device is present and can have a new firmware loaded on it. We'll come back to this test later to explain why we warned about the complexity. For now let's move forward to capturing and analysing some IMU data.

3.2 IMU Data Test

This test will use the following Nanaimo fixtures:

pytest fixture name	Role
nanaimo_serial	Attached to the IMU to capture data.
nanaimo_serial_watch	Attached to the IMU validate that it started up normally.
nanaimo_yepkit	USB3 hub with controllable power output.
nanaimo_gather	Run the serial watcher at the same time we turn on the IMU power

```
from nanaimo import assert_success

@pytest.mark.asyncio
async def test_imu(nanaimo_arguments, nanaimo_serial_watch, nanaimo_serial, nanaimo_
↪yepkit, nanaimo_gather):
    """
    A test that verifies that our IMU is returning sensible data.
    """

    yepkit_port_for_pantilt = nanaimo_arguments.yep_pantilt_port
    yepkit_port_for_imu = nanaimo_arguments.yep_imu_port

    # Enable the IMU and ensure we see the expected "I'm up" message
    coroutines = [
        nanaimo_serial_watch.gather(lw_pattern=r"I'm\s+up"),
        nanaimo_yepkit.gather(yep_port=yepkit_port_for_imu, yep_command='u')
    ]

    assert_success(await nanaimo_gather.gather(gather_coroutine=coroutines))

    # Start the pan-tilt fixture.
    assert_success(await nanaimo_yepkit.gather(yep_port=yepkit_port_for_pantilt,
                                               yep_command='u'))

    # We're going to wait 3-seconds to let the IMU warm up a bit, to let the
    # pan-tilt hardware to work out any resonances from startup impulses, and
    # to let any filters flatten out in the IMU.
    await nanaimo_gather.countdown_sleep(3)

    # We'll capture 10-seconds of data.
    artifacts = assert_success(await nanaimo_serial.gather(ser_memory_capture=10))

    # Shutdown the pan-tilt fixture and the IMU.
    assert_success(await nanaimo_yepkit.gather(yep_port=[yepkit_port_for_pantilt,
                                                         yepkit_port_for_imu],
                                               yep_command='d'))

    # ######################################################################
    #  At this point we're done acquiring data. Next use this data to
    #  evaluate the performance of the IMU
    # ######################################################################

    # for ypr_tuple in artifacts.data
```

We're going to end this example here to avoid digressing into a discussion on analysing sensor data and stay focused on the mechanics of Nanaimo. Before we move on though we offer Figure 3.1 which is real data captured from the test rig shown in Figure 1. Seeing this data one can see how imprecise the test rig is but also that there are obvious strategies for sanity testing this data-set.

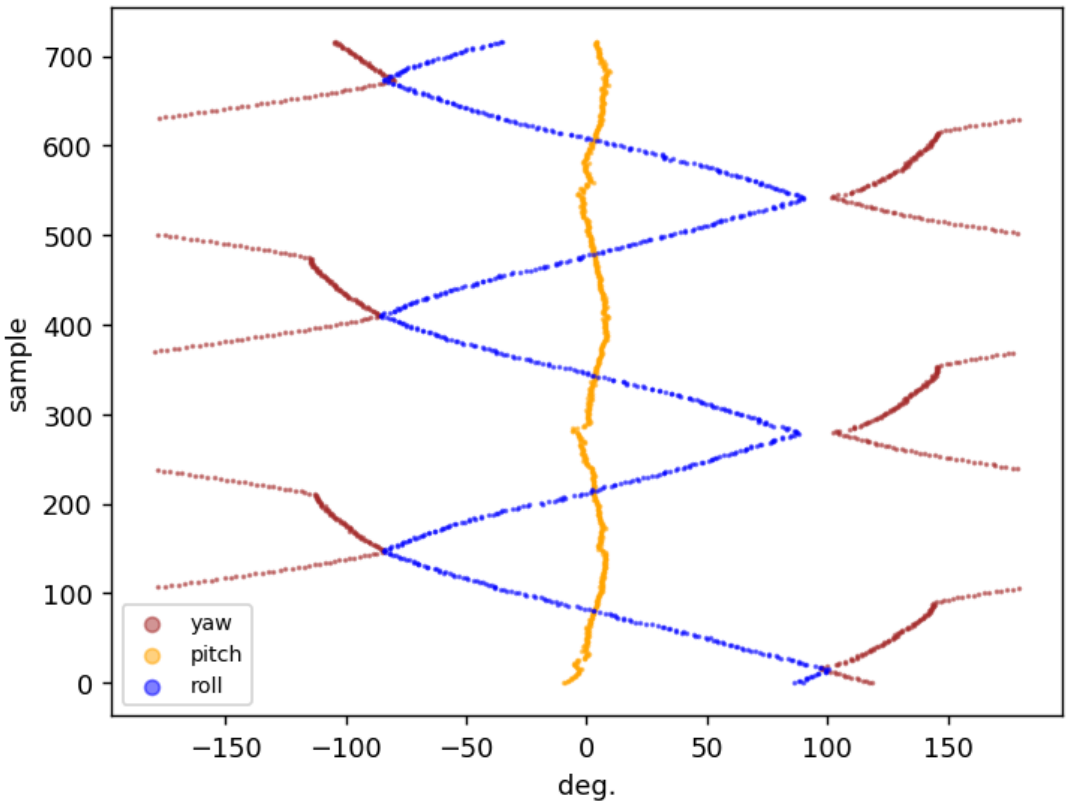

Fig. 3.1: Plot of Yaw Pitch and Roll data acquired as part of the example test.

3.3 Firmware Update Pt.2

Let's revisit the firmware update test again. The complexity we hinted at arises from two things:

1. Making sure the firmware is loaded before each test.
2. Reducing test time and decreasing part wear by skipping the firmware update if the right firmware was already on the device.

Item 2 is sometimes handled automatically by the underlying tooling (for example, Segger JLink is very good at eliding unnecessary writes) and the amount of time it takes to "upload" or "flash" a firmware to a target device varies significantly depending on the device's capabilities, the size of the binary, the efficiency of the programmer, and upload protocol in use. Where the test itself must contain logic to avoid unnecessary programming we need to make some decisions. We either need to check and potentially update the firmware before each test or we need to be sure we always run the firmware update test first and successfully.

As Item 2 details, before we can make any decisions on how to ensure we have the correct firmware we need a way to check this. Again, the underlying tool can often handle this for us automatically but for our example we'll have to ask:

```
from nanaimo import assert_success

@pytest.mark.asyncio
async def test_upload_firmware_if_needed(nanaimo_arguments, nanaimo_cmd, nanaimo_log):
    """
    We'll expand our use of the fictitious 'upload_firmware' binary to suppose it
    has a '--query' parameter that returns the version of the firmware found on the
    device.
    """

    class VersionFilter(logging.Filter):
        '''
        Very naive filter that finds and stores only the structured version number
        found in the subprocess output.
        '''

        def __init__(self):
            self.result = None

        def filter(self, record: logging.LogRecord) -> bool:
            if record.getMessage().startswith('version='):
                self.result = [int(x) for x in record.getMessage()[8:].split('.')]

    query_command = 'upload_firmware --query-version {imu_port}'.format(
                        **vars(nanaimo_arguments)
                    )
    upload_command = 'upload_firmware {imu_firmware} {imu_port}'.format(
                        **vars(nanaimo_arguments)
                    )

    nanaimo_cmd.stdout_filter = VersionFilter()
    assert_success(await nanaimo_cmd.gather(cmd_shell=query_command))

    version_triplet = nanaimo_cmd.stdout_filter.result

    # We'll be reusing this fixture so let's unset the stdout filter.
    nanaimo_cmd.stdout_filter = None

    if nanaimo_arguments.imu_firmware_version_major != version_triplet[0] or \
```

(continues on next page)

(continued from previous page)

```
        nanaimo_arguments.imu_firmware_version_minor != version_triplet[1] or \
        nanaimo_arguments.imu_firmware_version_patch != version_triplet[2]:

            nanaimo_log.info('Required firmware version %d.%d.%d - Found %d.%d.%d',
                nanaimo_arguments.imu_firmware_version_major,
                nanaimo_arguments.imu_firmware_version_minor,
                nanaimo_arguments.imu_firmware_version_patch,
                version_triplet[0],
                version_triplet[1],
                version_triplet[2]
            )

            # Okay, NOW we know we need to upload. So do that.
            assert_success(await nanaimo_cmd.gather(cmd_shell=upload_command))

    else:

            nanaimo_log.info('Required firmware version %d.%d.%d Found. Skipping upload.',
                nanaimo_arguments.imu_firmware_version_major,
                nanaimo_arguments.imu_firmware_version_minor,
                nanaimo_arguments.imu_firmware_version_patch
            )
```

So, there are a few problems with our `test_upload_firmware_if_needed` method above. First, it's a lot of logic that isn't actually testing anything and there's no guarantee that this test will run before any other test that is implicitly testing a specific firmware version. Remembering the first part of our mantra "Encapsulate complexity in Fixtures" it looks like we need a fixture.

While you can create a fixture as part of a redistributable python package we're going to keep it simple and just use the `conftest.py`[4] file we created next to our `test_stuff.py` file. Open that file in your favorite editor and add the following:

Note: Unfortunately, this example won't actually do anything because your system doesn't have an "upload_firmware" binary that behaves as our tests assume. You could create a dummy program or you can modify the shell command to use a programmer that you do have available on your system.

```
#
# conftest.py
#

import logging
import pytest
import typing
import sys
from unittest.mock import MagicMock

import nanaimo
import nanaimo.config
import nanaimo.fixtures
import nanaimo.pytest
import nanaimo.pytest.plugin
from nanaimo.fixtures import FixtureManager
from nanaimo import assert_success
```

(continues on next page)

[4] Conftest.py in pytest docs: https://docs.pytest.org/en/latest/fixture.html#conftest-py

(continued from previous page)

```
class FirmwareUpdateFixture(nanaimo.builtin.nanaimo_cmd.Fixture):
    """
    You'll want to give your fixture a good class docstring since this is used by␣
↪Nanaimo
    as the help text in the --fixtures output (when using the redistributable form of␣
↪a
    fixture, see below)
    """

    fixture_name = 'firmware_update'
    argument_prefix = 'fwr'

    class VersionFilter(logging.Filter):
        '''
        Very naive filter that finds and stores only the structured version number
        found in the subprocess output.
        '''

        def __init__(self):
            self.result = None

        def filter(self, record: logging.LogRecord) -> bool:
            if record.getMessage().startswith('version='):
                self.result = [int(x) for x in record.getMessage()[8:].split('.')]
            return True

    query_command = 'upload_firmware --query-version {port}'
    upload_command = 'upload_firmware {firmware} {port}'

    def __init__(self,
                 manager: 'FixtureManager',
                 args: typing.Optional[nanaimo.Namespace] = None,
                 **kwargs: typing.Any):
        super().__init__(manager, args, **kwargs)
        self._cmd = nanaimo.builtin.nanaimo_cmd.Fixture(manager, args, **kwargs)

    @classmethod
    def on_visit_test_arguments(cls, arguments: nanaimo.Arguments) -> None:
        nanaimo.builtin.nanaimo_cmd.Fixture.on_visit_test_arguments(arguments)
        arguments.add_argument('force', action='store_true', help='Always upload␣
↪firmware.')
        arguments.add_argument('firmware_version_major', type=int, help='Required␣
↪major firmware version.')
        arguments.add_argument('firmware_version_minor', type=int, help='Required␣
↪minor firmware version.')
        arguments.add_argument('firmware_version_patch', type=int, help='Required␣
↪patch firmware version.')
        arguments.add_argument('port', help='The serial port to provide to upload_
↪firmware')
        arguments.add_argument('firmware', help='The firmware file to upload.')

    async def on_gather(self, args: nanaimo.Namespace) -> nanaimo.Artifacts:

        if args.fwr_force:
```

(continues on next page)

(continued from previous page)

```
            self._logger.info('Forced firmware upload...')

            # We are forcing the upload to be sure it works regardless of what exists␣
↪on the
            # target right now.
            return await self._do_upload(args)

        else:
            version_triplet = await self._query_version(args)

            if args.fwr_firmware_version_major != version_triplet[0] or \
                    args.fwr_firmware_version_minor != version_triplet[1] or \
                    args.fwr_firmware_version_patch != version_triplet[2]:

                self._logger.info('Required firmware version %d.%d.%d - Found %d.%d.%d
↪',
                                  args.fwr_firmware_version_major,
                                  args.fwr_firmware_version_minor,
                                  args.fwr_firmware_version_patch,
                                  version_triplet[0],
                                  version_triplet[1],
                                  version_triplet[2]
                                  )

                # Okay, NOW we know we need to upload. So do that.
                return await self._do_upload(args)

            else:

                self._logger.info('Required firmware version %d.%d.%d Found. Skipping␣
↪upload.',
                                  args.fwr_firmware_version_major,
                                  args.fwr_firmware_version_minor,
                                  args.fwr_firmware_version_patch
                                  )

    async def _query_version(self, args: nanaimo.Namespace) -> typing.Tuple[float,␣
↪float, float]:
        self._cmd.stdout_filter = self.VersionFilter()

        try:
            query = self.query_command.format(port=args.fwr_port)
            await self._cmd.gather(cmd_shell=query)

            return self._cmd.stdout_filter.result

        finally:
            self._cmd.stdout_filter = None

    async def _do_upload(self, args: nanaimo.Namespace) -> nanaimo.Artifacts:
        upload = self.upload_command.format(firmware=args.fwr_firmware,
                                            port=args.fwr_port)
        return await self._cmd.gather(cmd_shell=upload)


@pytest.fixture
def firmware_update(nanaimo_fixture_manager, nanaimo_arguments) -> nanaimo.fixtures.
↪Fixture:
```

(continues on next page)

(continued from previous page)

```
    return FirmwareUpdateFixture(nanaimo_fixture_manager, nanaimo_arguments)
```

Now go back to your test file and change the `test_upload_firmware_if_needed` method to:

```
from nanaimo import assert_success

@pytest.mark.asyncio
async def test_upload_firmware(firmware_update):
    assert_success(await firmware_update.gather(fwr_force=True))
```

Note: Note that the pytest fixture name should be the canonical name or `fixture_name` you defined for your fixture. When using setup.cfg to register redistributable plugins this is done automatically. It's good form to follow this convention when defining the plugin "manually" in a conftest.py. See the documentation for *nanaimo.pytest.plugin* for more details on creating and registering your own *Fixture* types.

This is now a test that always runs to verify that the firmware update works. For all other tests we can reuse this fixture to ensure we are testing with the current firmware. For example:

```
@pytest.mark.asyncio
async def test_imu(firmware_update,
                   nanaimo_arguments,
                   nanaimo_serial_watch,
                   nanaimo_serial,
                   nanaimo_yepkit,
                   nanaimo_gather):
    """
    A test that verifies that our IMU is returning sensible data.
    """

    assert_success(await firmware_update.gather())

    yepkit_port_for_pantilt = nanaimo_arguments.yep_pantilt_port
    yepkit_port_for_imu = nanaimo_arguments.yep_imu_port
    ...
```

3.4 Configuration

Finally, let's look at how the configuration works. In our example so far we've assumed several properties were available in the fixture arguments. There are several ways to provide these values. For a Linux-like system that is dedicated as a test host you may want to create an `/etc/nanaimo.cfg` file. For user overrides you can provide `~/nanaimo.cfg`. For your test you can supply these arguments either in `setup.cfg` or in `tox.ini`. For example:

```
# In tox.ini

[nanaimo]
fwr_firmware_version_major = 2
fwr_firmware_version_minor = 1
fwr_firmware_version_patch = 0
fwr_port = /dev/serial/by-id/usb-some-uart-port0
fwr_firmware = myfirmware.bin
```

A few things to note about Nanaimo ini syntax; first it uses `configparser.ConfigParser` to parse the configuration files and it also uses `configparser.ExtendedInterpolation` to allow referencing other nanaimo ini sections. Finally, underscores are treated as namespaces so the following tox configuration is equivalent to the previous example:

```
# In tox.ini

[nanaimo:fwr]
port = /dev/serial/by-id/usb-some-uart-port0
firmware = myfirmware.bin

[nanaimo:fwr_firmware_version]
major = 2
minor = 1
patch = 0
```

Also note in our previous fixture example that the prefix `fwr` was omitted from the argument names:

```
@classmethod
def on_visit_test_arguments(cls, arguments: nanaimo.Arguments) -> None:
    nanaimo.builtin.nanaimo_cmd.Fixture.on_visit_test_arguments(arguments)
    arguments.add_argument('force', action='store_true', help='Always upload firmware.
↪')
    arguments.add_argument('firmware_version_major', type=int, help='Required major␣
↪firmware version.')
    arguments.add_argument('firmware_version_minor', type=int, help='Required minor␣
↪firmware version.')
    arguments.add_argument('firmware_version_patch', type=int, help='Required patch␣
↪firmware version.')
    arguments.add_argument('port', help='The serial port to provide to upload_firmware
↪')
    arguments.add_argument('firmware', help='The firmware file to upload.')
```

This is done on purpose. Nanaimo will prepend the prefix based on the value of `argument_prefix` for a given fixture. What's useful about this is it allows fixtures to be composed out of other fixtures. For example, if you compose `MyOtherUpdateFixture` fixture out of `FirmwareUpdateFixture` like this:

```
class MyOtherUpdateFixture(FirmwareUpdateFixture):
    ...
    argument_prefix = 'mou'
    ...
    @classmethod
    def on_visit_test_arguments(cls, arguments: nanaimo.Arguments) -> None:
        super().on_visit_test_arguments(arguments)
```

. . . then your new fixture's arguments will be prefixed with *mou* and won't conflict with *fwr* arguments. If you aggregate in a fixture instead (like `FirmwareUpdateFixture` does with `nanaimo_cmd.Fixture` then the arguments will get their prefix from the aggregate.

3.5 Redistributable

Let's say we like our firmware update fixture so much we want to package it up and make it available to other pytests using our package. To do this we'd change four things from our previous example:

1. We need to add a setup.py and setup.cfg to allow setuptools to package up our python project for redistribution

```
+ myproject
|
|   + src
|   |
|   + test
|   |   test_stuff.py
|   |   conftest.py
|
| tox.ini
| setup.py
| setup.cfg
```

Feel free to use the nanaimo github repository as an example if you like.

2. Move our `FirmwareUpdateFixture` to its own module. Let's say we created a `my_module.py` file and moved our fixture out of conftest.py to this file

```
+ myproject
|
|   + src
|   |   my_module.py
|   |
|   + test
|   |   test_stuff.py
|   |   conftest.py
|
| tox.ini
| setup.py
| setup.cfg
```

3. In `my_module.py` change these lines

```
# This is a quick-and-dirty way to create our module that works for conftest.py
# without any special configuration.
@pytest.fixture
def firmware_update(nanaimo_fixture_manager, nanaimo_arguments) -> nanaimo.
↪fixtures.Fixture:
    return FirmwareUpdateFixture(nanaimo_fixture_manager, nanaimo_arguments)
```

to this

```
# This extends the core Nanaimo pytest plugin with our own but we need to␣
↪tell
# Nanaimo about it in our setup.cfg.
def pytest_nanaimo_fixture_type() -> typing.Type['nanaimo.fixtures.Fixture']:
    return FirmwareUpdateFixture
```

4. Finally list our new module in the pytest11 section of your setup.cfg

```
[options.entry_points]
pytest11 =
    pytest_nanaimo = nanaimo.pytest.plugin
    pytest_nanaimo_plugin_firmware_update = my_module
```

Now your fixture is available in the same way it was before to tests but will also be available to any packages that depend on your package and the fixture will be available to *nait (NanAimo Interactive Terminal)*. Furthermore, `pytest --help` will now show your fixture arguments and `pytest --fixtures` will include the docstring for your fixture class. Finally, you can now distribute your fixture to your tests using pypi or a similar distribution service.

CHAPTER 4

nait (NanAimo Interactive Terminal)

Nanaimo comes with a CLI designed to allow direct interaction with one or more fixtures installed in your environment. This allows you to interactively test configuration or reset the state of a failed fixture. These fixtures will only be available to nait if they are specified in your setup.cfg (as we discussed in the previous section and as detailed in *Nanaimo's pytest plugin documentation*). Using this guide's firmware update fixture example one might use nait to manually update a firmware like this

```
nait firmware_update --fwr-firmware path/to/my/firmware.bin
```

Nait will reuse all our configuration and will provide the exact same environment to the fixture as pytest since it's actually just a thin wrapper around `pytest.main()`.

See the *nait reference section* of this documentation for more detail.

CHAPTER 5

Fixtures Reference

This page provides a reference for the available fixtures in Nanaimo. While the *Nanaimo (library)* guide is comprehensive it is useful only if you are writing your own fixtures. This page provides a more concise reference when writing tests using only Nanaimo's built-in fixtures and pytest plugins.

5.1 Builtin Pytest Fixtures

A set of pytest fixtures that come with nanaimo.

5.1.1  nanaimo_arguments

nanaimo.pytest.plugin.**nanaimo_arguments**(*request: Any*) → nanaimo.Namespace

Exposes the commandline arguments and defaults provided to a test.

```
def test_example(nanaimo_arguments: nanaimo.Namespace) -> None:
    an_argument = nanaimo_arguments.some_arg
```

Parameters **pytest_request** (*_pytest.fixtures.FixtureRequest*) – The request object passed into the pytest fixture factory.

Returns A namespace with the pytest commandline args added per the documented rules.

Return type *nanaimo.Namespace*

5.1.2 nanaimo_log

nanaimo.pytest.plugin.**nanaimo_log**(*request: Any*) → logging.Logger
Provides the unit tests with a logger configured for use with the Nanaimo framework.

Note: For now this is just a Python logger. Future revisions may add capabilities like the ability to log to a display or otherwise provide feedback to humans about the status of a test.

```
def test_example(nanaimo_log: logging.Logger) -> None:
    nanaimo_log.info('Hiya')
```

It's recommended that all Nanaimo tests configure logging in a tox.ini, pytest.ini, or pyproject.toml (when this is supported). For example, the following section in tox.ini would enable cli logging for nanaimo tests:

```
[pytest]
log_cli = true
log_cli_level = DEBUG
log_format = %(asctime)s %(levelname)s %(name)s: %(message)s
log_date_format = %Y-%m-%d %H:%M:%S
```

Parameters **pytest_request** (*_pytest.fixtures.FixtureRequest*) – The request object passed into the pytest fixture factory.

Returns A logger for use by Nanaimo tests.

Return type logging.Logger

5.1.3 nanaimo_fixture_manager

nanaimo.pytest.plugin.**nanaimo_fixture_manager**(*request: Any*) → nanaimo.fixtures.FixtureManager
Provides a default *FixtureManager* to a test.

```
def test_example(nanaimo_fixture_manager: nanaimo.Namespace) -> None:
    common_loop = nanaimo_fixture_manager.loop
```

Parameters **pytest_request** (*_pytest.fixtures.FixtureRequest*) – The request object passed into the pytest fixture factory.

Returns A new fixture manager.

Return type *nanaimo.fixtures.FixtureManager*

5.2 Builtin Nanaimo Fixtures

A set of predefined *nanaimo.fixtures.Fixture* types that come with nanaimo.

5.2.1 nanaimo_gather (gather)

class nanaimo.builtin.nanaimo_gather.**Fixture**(*manager: nanaimo.fixtures.FixtureManager*, *args: Optional[nanaimo.Namespace] = None*, ***kwargs*)

Bases: *nanaimo.fixtures.Fixture*

This fixture takes a list of other fixtures and runs them concurrently returning a *nanaimo.Artifacts.combine()* ed set of *nanaimo.Artifacts*.

You can use this fixture directly in your unit tests:

```
async def example1() -> Artifacts:

    bar_one = nanaimo_bar.Fixture(manager)
    bar_two = nanaimo_bar.Fixture(manager)
    gather_fixture = nanaimo_gather.Fixture(manager)

    return await gather_fixture.gather(
        gather_coroutine=[
            bar_one.gather(bar_number=1),
            bar_two.gather(bar_number=2)
        ]
    )
```

You can also use the –gather-coroutines argument to specify fixtures by name:

```
nait nanaimo_gather --gather-coroutine nanaimo_bar --gather-coroutine nanaimo_bar
```

classmethod on_visit_test_arguments(*arguments: nanaimo.Arguments*) → None
Called by the environment before instantiating any *nanaimo.fixtures.Fixture* instances to register arguments supported by each type. These arguments should be portable between both `argparse` and `pytest`. The fixture is registered for this callback by returning a reference to its type from a `pytest_nanaimo_fixture_type` hook in your fixture's pytest plugin module.

on_gather(*args: nanaimo.Namespace*) → nanaimo.Artifacts
Multi-plex fixture

5.2.2 nanaimo_serial_watch (lw)

class nanaimo.builtin.nanaimo_serial_watch.**Fixture**(*manager: nanaimo.fixtures.FixtureManager*, *args: nanaimo.Namespace*, ***kwargs*)

Bases: *nanaimo.fixtures.Fixture*

Gathers a log over a serial connection until a given pattern is matched.

classmethod on_visit_test_arguments(*arguments: nanaimo.Arguments*) → None
Called by the environment before instantiating any *nanaimo.fixtures.Fixture* instances to register arguments supported by each type. These arguments should be portable between both `argparse`

and `pytest`. The fixture is registered for this callback by returning a reference to its type from a `pytest_nanaimo_fixture_type` hook in your fixture's pytest plugin module.

on_gather(*args: nanaimo.Namespace*) → nanaimo.Artifacts
Watch the logs until the pattern matches.

Returned Artifacts		
key	type	Notes
match	re.MatchObject	The match if result_code is 0
matched_line	str	The full line matched if result_code is 0

5.2.3 Fixtures based on `nanaimo.fixture.SubprocessFixture`

class nanaimo.fixtures.**SubprocessFixture**(*manager: nanaimo.fixtures.FixtureManager*, *args: Optional[nanaimo.Namespace] = None*, ***kwargs*)

Bases: *nanaimo.fixtures.Fixture*

Fixture base type that accepts a string argument `cmd` and executes it as a subprocess. Because some subprocess commands might result in huge amounts of data being sent to stdout and/or stderr this class does not capture this data by default. Instead, tests are encouraged to either filter the subprocess pipes or use the `--logfile` argument to write the output to a file in persistent storage.

Filtering is accomplished using the *stdout_filter* or *stderr_filter* property of this class. For example:

```
class MySubprocessFixture(nanaimo.fixtures.SubprocessFixture):
    '''
    Subprocess test fixture that simply calls "nait --version"
    '''

    @classmethod
    def on_visit_test_arguments(cls, arguments: nanaimo.Arguments) -> None:
        pass

    def on_construct_command(self, arguments: nanaimo.Namespace, inout_artifacts:␣
↪nanaimo.Artifacts) -> str:
        return 'nait --version'


async def example(manager):

    subject = MySubprocessFixture(manager)

    # The accumulator does capture all stdout. Only use this if you know
    # the subprocess will produce a managable and bounded amount of output.
    filter = nanaimo.fixtures.SubprocessFixture.SubprocessMessageAccumulator()
    subject.stdout_filter = filter

    artifacts = await subject.gather()

    # In our example the subprocess produces only and exactly the Nanaimo
    # version number
    assert filter.getvalue() == nanaimo.version.__version__
```

Parameters

- **stdout_filter** – A `logging.Filter` used when gathering the subprocess.
- **stderr_filter** – A `logging.Filter` used when gathering the subprocess.

class SubprocessMessageAccumulator (*minimum_level: int = 20*)
Bases: `logging.Filter`, `_io.StringIO`

Helper class for working with *SubprocessFixture.stdout_filter()* or *SubprocessFixture.stderr_filter()*. This implementation will simply write all log messages (i.e. `logging.LogRecord.getMessage()`) to its internal buffer. Use `getvalue()` to get a reference to the buffer.

You can also subclass this method and override its `logging.Filter.filter()` method to customize your filter criteria.

Parameters **minimum_level** – The minimum loglevel to accumulate messages for.

filter (*record: logging.LogRecord*) → bool
Determine if the specified record is to be logged.

Is the specified record to be logged? Returns 0 for no, nonzero for yes. If deemed appropriate, the record may be modified in-place.

class SubprocessMessageMatcher (*pattern: Any*, *minimum_level: int = 20*)
Bases: `logging.Filter`

Helper class for working with *SubprocessFixture.stdout_filter()* or *SubprocessFixture.stderr_filter()*. This implementation will watch every log message and store any that match the provided pattern.

This matcher does not buffer all logged messages.

Parameters

- **pattern** – A regular expression to match messages on.
- **minimum_level** – The minimum loglevel to accumulate messages for.

match_count
The number of messages that matched the provided pattern.

filter (*record: logging.LogRecord*) → bool
Determine if the specified record is to be logged.

Is the specified record to be logged? Returns 0 for no, nonzero for yes. If deemed appropriate, the record may be modified in-place.

classmethod on_visit_test_arguments (*arguments: nanaimo.Arguments*) → None
Called by the environment before instantiating any *nanaimo.fixtures.Fixture* instances to register arguments supported by each type. These arguments should be portable between both `argparse` and `pytest`. The fixture is registered for this callback by returning a reference to its type from a `pytest_nanaimo_fixture_type` hook in your fixture's pytest plugin module.

on_gather (*args: nanaimo.Namespace*) → nanaimo.Artifacts

Artifacts		
key	type	Notes
`cmd`	str	The command used to execute the subprocess.
`logfile`	Optional[pathlib.Path]	A file containing stdout, stderr, and test logs

stdout_filter
A filter used when logging the stdout stream with the subprocess.

stderr_filter
A filter used when logging the stderr stream with the subprocess.

on_construct_command(*arguments: nanaimo.Namespace*, *inout_artifacts: nanaimo.Artifacts*) → str
Called by the subprocess fixture to ask the specialization to form a command given a set of arguments.

Parameters

- **arguments** (nanaimo.Arguments) – The arguments passed into *Fixture.on_gather()*.
- **inout_artifacts** (nanaimo.Artifacts) – A set of artifacts the superclass is assembling This is provided to the subclass to allow it to optionally contribute artifacts.

Returns The command to run in a subprocess shell.

5.2.3.1 nanaimo_cmd (cmd)

class nanaimo.builtin.nanaimo_cmd.**Fixture**(*manager: nanaimo.fixtures.FixtureManager*, *args: Optional[nanaimo.Namespace] = None*, ***kwargs*)
Bases: *nanaimo.fixtures.SubprocessFixture*

Fixture that accepts a string argument cmd and executes it as a subprocess.

```
async def list_directory() -> Artifacts:

    cmd = nanaimo_cmd.Fixture(manager)

    return await cmd.gather(
        cmd_shell = ('dir' if os.name == 'nt' else 'ls -la')
    )
```

classmethod on_visit_test_arguments(*arguments: nanaimo.Arguments*) → None
Called by the environment before instantiating any *nanaimo.fixtures.Fixture* instances to register arguments supported by each type. These arguments should be portable between both argparse and pytest. The fixture is registered for this callback by returning a reference to its type from a pytest_nanaimo_fixture_type hook in your fixture's pytest plugin module.

on_construct_command(*arguments: nanaimo.Namespace*, *inout_artifacts: nanaimo.Artifacts*) → str
Called by the subprocess fixture to ask the specialization to form a command given a set of arguments.

Parameters

- **arguments** (nanaimo.Arguments) – The arguments passed into Fixture.on_gather().
- **inout_artifacts** (nanaimo.Artifacts) – A set of artifacts the superclass is assembling This is provided to the subclass to allow it to optionally contribute artifacts.

Returns The command to run in a subprocess shell.

5.2.3.2 nanaimo_scp (scp)

class nanaimo.builtin.nanaimo_scp.**Fixture**(*manager: nanaimo.fixtures.FixtureManager*, *args: Optional[nanaimo.Namespace] = None*, ***kwargs*)

Bases: *nanaimo.fixtures.SubprocessFixture*

This fixture assumes that scp is available and functional on the system.

classmethod on_visit_test_arguments(*arguments: nanaimo.Arguments*) → None
Called by the environment before instantiating any *nanaimo.fixtures.Fixture* instances to register arguments supported by each type. These arguments should be portable between both argparse and pytest. The fixture is registered for this callback by returning a reference to its type from a pytest_nanaimo_fixture_type hook in your fixture's pytest plugin module.

on_construct_command(*args: nanaimo.Namespace*, *inout_artifacts: nanaimo.Artifacts*) → str
Form the upload command.

5.2.3.3 nanaimo_ssh (ssh)

class nanaimo.builtin.nanaimo_ssh.**Fixture**(*manager: nanaimo.fixtures.FixtureManager*, *args: Optional[nanaimo.Namespace] = None*, ***kwargs*)

Bases: *nanaimo.fixtures.SubprocessFixture*

This fixture assumes that ssh is available and functional on the system.

classmethod on_visit_test_arguments(*arguments: nanaimo.Arguments*) → None
Called by the environment before instantiating any *nanaimo.fixtures.Fixture* instances to register arguments supported by each type. These arguments should be portable between both argparse and pytest. The fixture is registered for this callback by returning a reference to its type from a pytest_nanaimo_fixture_type hook in your fixture's pytest plugin module.

on_construct_command(*args: nanaimo.Namespace*, *inout_artifacts: nanaimo.Artifacts*) → str
Called by the subprocess fixture to ask the specialization to form a command given a set of arguments.

Parameters

- **arguments** (nanaimo.Arguments) – The arguments passed into Fixture.on_gather().
- **inout_artifacts** (nanaimo.Artifacts) – A set of artifacts the superclass is assembling This is provided to the subclass to allow it to optionally contribute artifacts.

Returns The command to run in a subprocess shell.

5.3 Instrument Fixtures

Fixtures for popular measurement, control, and test gear.

5.3.1 nanaimo_instr_bk_precision (bk)

class nanaimo.instruments.bkprecision.**Series1900BUart**(*manager: nanaimo.fixtures.FixtureManager*, *args: nanaimo.Namespace*, ***kwargs*)

Bases: *nanaimo.fixtures.Fixture*

Control of a 1900B series BK Precision power supply via UART.

classmethod mode_to_text(*mode: int*) → str
Get a two-character textual representation for a given power supply mode.

UartFactoryType = typing.Callable[[typing.Union[str, pathlib.Path]], typing.Any]
The serial port factory type for this instrument.

classmethod default_serial_port(*port: Union[str, pathlib.Path]*) → Generator[nanaimo.connections.AbstractAsyncSerial, None, None]
Creates a serial connection to the given port using the default settings for a BK Precision Series 1900B power supply.

classmethod on_visit_test_arguments(*arguments: nanaimo.Arguments*) → None
Called by the environment before instantiating any *nanaimo.fixtures.Fixture* instances to register arguments supported by each type. These arguments should be portable between both argparse and pytest. The fixture is registered for this callback by returning a reference to its type from a pytest_nanaimo_fixture_type hook in your fixture's pytest plugin module.

is_volage_above_on_threshold(*voltage: float*) → bool
Deprecated misspelling. See *is_voltage_above_on_threshold()* for correct method.

is_voltage_above_on_threshold(*voltage: float*) → bool
Return if a given voltage is above the configured threshold for the high/on/rising voltage for this fixture.

Raises **ValueError** – if no target voltage could be determined.

is_volage_below_off_threshold(*voltage: float*) → bool
Deprecated misspelling. See *is_voltage_below_off_threshold()* for correct method.

is_voltage_below_off_threshold(*voltage: float*) → bool
Return if a given voltage is below the configured threshold for the low/off/falling voltage for this fixture.

on_gather(*args: nanaimo.Namespace*) → nanaimo.Artifacts
Send a command to the instrument and return the result. :param str command: Send one of the following commands:

Command	Action	Returns
'1'	Turn on output voltage	'OK' or error text.
'0'	Turn off output voltage	'OK' or error text
'r'	Send a stream of <cr> characters	(NA)
'?'	Read the front panel display	Display voltage, current, and status (ON or OFF)

5.3.2 nanaimo_instr_ykush (wk)

class nanaimo.instruments.ykush.**Fixture**(*manager: nanaimo.fixtures.FixtureManager*, *args: Optional[nanaimo.Namespace] = None*, ***kwargs*)

Bases: *nanaimo.fixtures.SubprocessFixture*

Fixture for controlling Yepkit USB hubs with switchable power. For example the YKUSH3 is a 3-port USB-3 hub that allows individual control of the power rails for each port.

This is a subprocess fixture that requires the `ykushcmd` program is available in the subprocess environment (see Yepkit's documentation for how to build this from source). All arguments can be overridden via the fixtures gather method. The supported commands are:

command	example	Description
`yku-all-on`	`await nanaimo_instr_ykush.gather(yku_all_on=True)`	Turn on power to all ports on the YKUSH.
`yku-all-off`	`await nanaimo_instr_ykush.gather(yku_all_off=True)`	Turn off power to all ports on the YKUSH.
`yku-command`	`await nanaimo_instr_ykush.gather(yku_command='-l')`	Pass-through any command to ykushcmd.

classmethod on_visit_test_arguments(*arguments: nanaimo.Arguments*) → None

Called by the environment before instantiating any *nanaimo.fixtures.Fixture* instances to register arguments supported by each type. These arguments should be portable between both `argparse` and `pytest`. The fixture is registered for this callback by returning a reference to its type from a `pytest_nanaimo_fixture_type` hook in your fixture's pytest plugin module.

on_construct_command(*arguments: nanaimo.Namespace*, *inout_artifacts: nanaimo.Artifacts*) → str

CHAPTER 6

nait (NanAimo Interactive Terminal)

Usage:

```
nait --help
nait --fixtures
```

The commandline `nait` is installed with the nanaimo package and provides direct access to nanaimo fixtures without having to write pytests. The CLI still uses pytest but synthesizes tests based on the fixtures provided. See *Fixtures Reference* for fixture names and arguments.

Example:

```
> nait nanaimo_bar

----------------------------------- live log call -----------------------------------
2019-12-18 16:49:34 DEBUG nanaimo.config: Configuration read from ['/etc/nanaimo.cfg',
 'setup.cfg', 'tox.ini']
2019-12-18 16:49:34 INFO nanaimo_bar: don't forget to eat your dessert.
PASSED                                                                        [100
%]

--------------------------------------- nanaimo ---------------------------------------
Fixture(s) "nanaimo_bar" result = 0
.............................. Artifacts for nanaimo_bar ..............................
eat=functools.partial(<bound method Logger.info of <Logger nanaimo_bar (DEBUG)>>,
'Nanaimo bars are yummy.')
bar_None=0.42314194
================================== 1 passed in 0.02s␣
==================================

> nait nanaimo_cmd -C 'nait --help'
```

(continues on next page)

(continued from previous page)

```
---------------------------------- live log call -----------------------------------
↪--
2019-12-18 16:51:35 DEBUG nanaimo.config: Configuration read from ['/etc/nanaimo.cfg',
↪ 'setup.cfg', 'tox.ini']
2019-12-18 16:51:35 DEBUG nanaimo_cmd: About to execute command "nait --help" in a␣
↪subprocess shell
2019-12-18 16:51:36 INFO nanaimo_cmd: usage: nait [options] [nanaimo_fixture]␣
↪[nanaimo_fixture] [...]
2019-12-18 16:51:36 DEBUG nanaimo_cmd: command "nait --help" exited with 0
PASSED                                                                    [100
↪%]

-------------------------------------- nanaimo --------------------------------------
↪--
Fixture(s) "nanaimo_cmd" result = 0
.............................. Artifacts for nanaimo_cmd ..............................
↪..
================================== 1 passed in 0.47s␣
↪==================================
```

You are also invoking pytest so all pytest arguments apply. Use `nait --help` to see a listing of arguments and `nait --fixtures` to see all available fixtures.

6.1 Special Arguments

The following arguments are specific to nait and are not the same as when invoking `pytest` directly:

- `--version` – List the version of Nanaimo (*not* pytest).
- `-S` or `--environ-shell` – List all environment variables provided to subprocess shells in shell syntax. This allows applying the subprocess environment to your current session by doing (in bash) `eval $(nait -S)`
- `--environ` – A key=value argument to define environment variables in a subprocess shell. You can provide multiple arguments for example

```
nait --environ foo=bar --environ baz=biz -S
```

- `--concurrent` – Runs multiple fixtures concurrently

```
# Runs two nanaimo_bar fixtures one after the other
nait nanaimo_bar nanaimo_bar

# Runs two nanaimo_bar fixtures concurrently
nait --concurrent nanaimo_bar nanaimo_bar
```

CHAPTER 7

Nanaimo (library)

7.1 nanaimo

This module contains the common types used by Nanaimo.

exception nanaimo.**AssertionError**
Bases: RuntimeError

Thrown by Nanaimo tests when an assertion has failed.

Note: This exception should be used only when the state of a *nanaimo.fixtures.Fixture* was invalid. You should use pytest tests and assertions when writing validation cases for fixture output like log files or sensor data.

---

class nanaimo.**Arguments**(*inner_arguments: Any*, *defaults: Optional[nanaimo.config.ArgumentDefaults] = None*, *required_prefix: Optional[str] = None*, *filter_duplicates: bool = False*)
Bases: object

Adapter for pytest and argparse parser arguments.

Parameters

- **inner_arguments** – Either a pytest group (unpublished type returned from pytest.Parser.getgroup()) or a argparse.ArgumentParser
- **defaults** (*typing.Optional[ArgumentDefaults]*) – Optional provider of default values for arguments.
- **required_prefix** (*str*) – If provided *add_argument()* will rewrite arguments to enure they have the required prefix.
- **filter_duplicates** (*bool*) – If true then this class will track keys provided to the *add_argument()* method and will not call the inner object if duplicates are detected. If

false then all calls to *add_argument()* are always forwarded to the inner object. This filter is tracked per instance so duplicates provided to different instances are not filtered.

set_inner_arguments (*inner_arguments: Any*) → None

Reset the inner argument object this object wraps. This method allows a single instance to filter all arguments preventing duplicates from reading the inner objects.

```
a = Arguments(parser, filter_duplicates=True)
a.add_argument('--foo')

# This second call will not make it to the parser
# object we set above.
a.add_argument('--foo')

# If we set another parser on the same Arguments instance...
a.inner_arguments = my_other_parser

# then the same filter will continue to apply for this new
# inner argument object.
a.add_argument('--foo')
```

add_argument (**args*, ***kwargs*) → None

This method invokes `argparse.ArgumentParser.add_argument()` but with one additional argument: `enable_default_from_environ`. If this is provided as True then a default value will be taken from an environment variable derived from the long form of the argument:

```
# Using...
long_arg = '--baud-rate'

# ...the environment variable looked for will be:
environment_var_name = 'NANAIMO_BAUD_RATE'

# If we set the environment variable...
os.environ[environment_var_name] = '115200'

a = Arguments(parser, config)

# ...and provide a default...
a.add_argument('--baud-rate',
               default=9600,
               type=int,
               enable_default_from_environ=True,
               help='Will be 9600 unless argument is provided.')

# ...the actual default value will be 115200
```

```
assert add_argument_call_args['default'] == 115200
```

```
# Using a required prefix...
a = Arguments(parser, config, required_prefix='ad')

# ...and adding an argument...
a.add_argument('--baud-rate')

# ...the actual argument added will be
actual_long_arg = '--ad-baud-rate'
```

class nanaimo.**Namespace**(*parent: Optional[Any] = None*, *defaults: Optional[nanaimo.config.ArgumentDefaults] = None*, *allow_none_values: bool = True*)

Bases: `object`

Generic object that acts like `argparse.Namespace` but can be created using pytest plugin arguments as well.

If *`nanaimo.config.ArgumentDefaults`* are used with the *`Arguments`* and this class then a given argument's value will be resolved in the following order:

1. provided value
2. config file specified by –rcfile argument.
3. nanaimo.cfg in user directory
4. nanaimo.cfg in system directory
5. default from environment (if `enable_default_from_environ` was set for the argument)
6. default specified for the argument.

This is accomplished by first rewriting the defaults when attributes are defined on the *`Arguments`* class and then capturing missing attributes on this class and looking up default values from configuration files.

For lookup steps involving configuration files (where `configparser.ConfigParser` is used internally) the lookup will search the configuration space using underscores _ as namespace separators. this search will proceed as follows:

```
# given
key = 'a_b_c_d'

# the following lookups will occur
config_lookups = {
    'nanaimo:a_b_c': 'd',
    'nanaimo:a_b': 'c_d',
    'nanaimo:a': 'b_c_d',
    'a_b_c': 'd',
    'a_b': 'c_d',
    'a': 'b_c_d',
    'nanaimo': 'a_b_c_d'
}

# when using an ArgumentDefaults instance
_ = argument_defaults[key]
```

So for a given configuration file:

```
[nanaimo]
a_b_c_d = 1

[a]
b_c_d = 2
```

the value `2` under the `a` group will override (i.e. mask) the value `1` under the `nanaimo` group.

Note: A specific example:

- `--bk-port <value>` – if provided on the commandline will always override everything.

- `[bk] port = <value>` – in a config file will be found next if no argument was given on the commandline.
- `NANAIMO_BK_PORT` - set in the environment will be used if no configuration was provided because the *nanaimo.instruments.bkprecision* module defines the `bk-port` argument with `enable_default_from_environ` set.

This object has a somewhat peculiar behavior for Python. All attributes will be reported either as a found value or as `None`. That is, any arbitrary attribute requested from this object will be `None`. To differentiate between `None` and "not set" you must using `in`:

```
ns = nanaimo.Namespace()
assert ns.foo is None
assert 'foo' not in ns
```

The behavior was designed to simplify argument handling code since argparse Namespaces will have `None` values for all arguments even if the were not provided and had no default value.

Parameters

- **parent** (*typing.Optional[typing.Any]*) – A namespace-like object to inherit attributes from.
- **defaults** (*typing.Optional[ArgumentDefaults]*) – Defaults to use if a requested attribute is not available on this object.
- **allow_none_values** (*bool*) – If True then an attribute with a None value is considered valid otherwise any attribute that is None will cause the Namespace to search for a non-None value in the defaults.

get_as_merged_dict(*key: str*) → Mapping[str, Any]

Expect the value to be a dictionary. In this case also load the defaults into the dictionary.

Parameters **key** – The key to load the dictionary from.

merge(***kwargs*) → T

Merges a list of keyword arguments with this namespace and returns a new, merged Namespace. This does not modify the instance that merge is called on.

Example:

```
original = Namespace()
setattr(original, 'foo', 1)

assert 1 == original.foo

merged = original.merge(foo=2, bar='hello')

assert 1 == original.foo
assert 2 == merged.foo
assert 'hello' == merged.bar
```

Returns A new namespace with the contents of this object and any values provided as kwargs overwriting the values in this instance where the keys are the same.

class nanaimo.**Artifacts**(*result_code: int = 0*, *parent: Optional[Any] = None*, *defaults: Optional[nanaimo.config.ArgumentDefaults] = None*, *allow_none_values: bool = True*)

Bases: *nanaimo.Namespace*

Namespace returned by *nanaimo.fixtures.Fixture* objects when invoked that contains the artifacts collected from the fixture's activities.

Parameters

- **result_code** – The value to report as the status of the activity that gathered the artifacts.
- **parent** (*typing.Optional[typing.Any]*) – A namespace-like object to inherit attributes from.
- **defaults** (*typing.Optional[ArgumentDefaults]*) – Defaults to use if a requested attribute is not available on this object.
- **allow_none_values** (*bool*) – If True then an attribute with a None value is considered valid otherwise any attribute that is None will cause the Artifacts to search for a non-None value in the defaults.

classmethod combine(**artifacts*) → nanaimo.Artifacts

Combine a series of artifacts into a single instance. This method uses *Namespace.merge()* but adds additional semantics including:

Note: While this method does not modify the original objects it also does not do a deep copy of artifact values.

Given two *Artifacts* objects with the same attribute the right-most item in the combine list will overwrite the previous values and become the only value:

```
setattr(first, 'foo', 1)
setattr(second, 'foo', 2)

assert Artifacts.combine(first, second).foo == 2
assert Artifacts.combine(second, first).foo == 1
```

The *result_code* of the combined value will be either 0 iff all combined Artifact objects have a result_code of 0:

```
first.result_code = 0
second.result_code = 0

assert Artifacts.combine(first, second).result_code == 0
```

or will be non-zero if any instance had a non-zero result code:

```
first.result_code = 0
second.result_code = 1

assert Artifacts.combine(first, second).result_code != 0
```

Parameters **artifacts** – A list of artifacts to combine into a single *Artifacts* instance.

Raises **ValueError** – if no artifact objects were provided or if the method was otherwise unable to create a new object from the provided ones.

result_code

0 if the artifacts were retrieved without error. Non-zero if some error occurred. The contents of this *Namespace* is undefined for non-zero result codes.

dump(*logger: logging.Logger*, *log_level: int = 10*) → None
Dump a human readable representation of this object to the given logger. :param logger: The logger to use. :param log_level: The log level to dump the object as.

nanaimo.**assert_success**(*artifacts: nanaimo.Artifacts*) → nanaimo.Artifacts
Syntactic sugar to allow more fluent handling of *fixtures.Fixture.gather()* artifacts. For example:

```
async def test_my_fixture():

    artifacts = assert_success(await fixture.gather())

    # Now we can use the artifacts. If the gather had returned
    # non-zero for the result_code an assertion error would have
    # been raised.
```

Parameters **artifacts** (nanaimo.Artifacts) – The artifacts to assert on.

Returns artifacts (for convenience).

Return type *nanaimo.Artifacts*()

nanaimo.**assert_success_if**(*artifacts: nanaimo.Artifacts*, *conditional: Callable[[nanaimo.Artifacts], bool]*) → nanaimo.Artifacts
Syntactic sugar to allow more fluent handling of *fixtures.Fixture.gather()* artifacts but with a user-supplied conditional.

```
async def test_my_fixture():

    def fail_if_no_foo(artifacts: Artifacts) -> bool:
        return 'foo' in artifacts

    artifacts = assert_success_if(await fixture.gather(), fail_if_no_foo)

    print('artifacts have foo. It's value is {}'.format(artifacts.foo))
```

Parameters

- **artifacts** (nanaimo.Artifacts) – The artifacts to assert on.
- **conditiona** – A method called to evaluate gathered artifacts iff *Artifacts.result_code* is 0. Return False to trigger an assertion, True to pass.

Returns artifacts (for convenience).

Return type *nanaimo.Artifacts*()

nanaimo.**set_subprocess_environment**(*args: nanaimo.Namespace*) → None
Updates os.environ from values set as environ in the provided arguments.

Parameters **args** – A namespace to load the environment from. The map of values in this key are added to any subsequent subprocess started but can be overridden by env arguments to subprocess constructors like subprocess.Popen

7.2 nanaimo.fixtures

Almost everything in Nanaimo is a *nanaimo.fixtures.Fixture*. Fixtures can be pytest fixtures, instrument abstractions, aggregates of other fixtures, or anything else that makes sense. The important thing is that any fixture

can be a pytest fixture or can be awaited directly using *nait (NanAimo Interactive Terminal)*.

class nanaimo.fixtures.**Fixture**(*manager: nanaimo.fixtures.FixtureManager*, *args: Optional[nanaimo.Namespace] = None*, ***kwargs*)

Bases: object

Common, abstract class for pytest fixtures based on Nanaimo. Nanaimo fixtures provide a visitor pattern for arguments that are common for both pytest extra arguments and for argparse commandline arguments. This allows a Nanaimo fixture to expose a direct invocation mode for debuging with the same arguments used by the fixture as a pytest plugin. Additionally all Nanaimo fixtures provide a *gather()* function that takes a *nanaimo.Namespace* containing the provided arguments and returns a set of *nanaimo.Artifacts* gathered by the fixture. The contents of these artifacts are documented by each concrete fixture.

```
class MyFixture(nanaimo.fixtures.Fixture):
    @classmethod
    def on_visit_test_arguments(cls, arguments: nanaimo.Arguments) -> None:
        arguments.add_argument('--foo', default='bar')

    async def on_gather(self, args: nanaimo.Namespace) -> nanaimo.Artifacts:
        artifacts = nanaimo.Artifacts(-1)
        # do something and then return
        artifacts.result_code = 0
        return artifacts
```

MyFixture can now be used from a commandline like:

```
python -m nanaimo MyFixture --foo baz
```

or as part of a pytest:

```
pytest --foo=baz
```

Parameters

- **manager** (FixtureManager) – The fixture manager that is the scope for this fixture. There must be a 1:1 relationship between a fixture instance and a fixture manager instance.
- **args** (nanaimo.Namespace) – A namespace containing the arguments for this fixture.
- **kwargs** – All fixtures can be given a asyncio.AbstractEventLoop instance to use as loop and an initial value for *gather_timeout_seconds* as gather_timeout_seconds (float). Other keyword arguments may used by fixture specializations.

classmethod get_canonical_name() → str

The name to use as a key for this *nanaimo.fixtures.Fixture* type. If a class defines a string *fixture_name* this will be used as the canonical name otherwise it will be the name of the fixture class itself.

```
class MyFixture(nanaimo.fixtures.Fixture):

    fixture_name = 'my_fixture'

assert 'my_fixture' == MyFixture.get_canonical_name()
```

classmethod get_argument_prefix() → str

The name to use as a prefix for arguments. This also becomes the configuration section that the fixture's arguments can be overridden from. If the fixture defines an argument_prefix class member this value is used otherwise the value returned from *get_canonical_name()* is used.

```
class MyFixture(nanaimo.fixtures.Fixture):

    argument_prefix = 'mf'
```

```
>>> MyFixture.get_argument_prefix()  # noqa : F821
'mf'
```

```
class MyOtherFixture(nanaimo.fixtures.Fixture):
    # this class doesn't define argument_prefix so
    # so the canonical name is used instead.
    fixture_name = 'my_outre_fixture'
```

```
>>> MyOtherFixture.get_argument_prefix()  # noqa : F821
'my-outre-fixture'
```

classmethod get_arg_covariant(*args: nanaimo.Namespace*, *base_name: str*, *default_value: Optional[Any] = None*) → Any

When called by a baseclass this method will return the most specalized argument value available.

Parameters

- **args** – The arguments to search.
- **base_name** – The base name. For example `foo` for `--prefix-bar`.
- **default_value** – The value to use if the argument could not be found.

classmethod get_arg_covariant_or_fail(*args: nanaimo.Namespace*, *base_name: str*) → Any

Calls *get_arg_covariant()* but raises `ValueError` if the result is *None*.

Raises **ValueError** – if no value could be found for the argument.

gather_until_complete(**args*, ***kwargs*) → nanaimo.Artifacts

helper function where this:

```
foo.gather_until_complete()
```

is equivalent to this:

```
foo.loop.run_until_complete(foo.gather())
```

name

The canonical name for the Fixture.

loop

The running asyncio EventLoop in use by this Fixture. This will be the loop provided to the fixture in the constructor if that loop is still running otherwise the loop will be a running loop retrieved by `asyncio.get_event_loop()`. :raises RuntimeError: if no running event loop could be found.

manager

The *FixtureManager* that owns this *nanaimo.fixtures.Fixture*.

logger

A logger for this *nanaimo.fixtures.Fixture* instance.

fixture_arguments

The Fixture-wide arguments. Can be overridden by kwargs for each *gather()* invocation.

gather_timeout_seconds
The timeout in fractional seconds to wait for *on_gather()* to complete before raising a asyncio.TimeoutError.

classmethod visit_test_arguments(*arguments: nanaimo.Arguments*) → None
Visit this fixture's *on_visit_test_arguments()* but with the proper nanaimo.Arguments.required_prefix set.

classmethod on_visit_test_arguments(*arguments: nanaimo.Arguments*) → None
Called by the environment before instantiating any *nanaimo.fixtures.Fixture* instances to register arguments supported by each type. These arguments should be portable between both argparse and pytest. The fixture is registered for this callback by returning a reference to its type from a pytest_nanaimo_fixture_type hook in your fixture's pytest plugin module.

on_test_teardown(*test_name: str*) → None
Invoked for each fixture after it was used in a test as part of the pytest _pytest.hookspec.pytest_runtest_teardown() protocol.

Parameters **test_name** – The name of the test the fixture was used with.

countdown_sleep(*sleep_time_seconds: float*) → None
Calls asyncio.sleep() for 1 second then emits an logging.Logger.info() of the time remaining until *sleep_time_seconds*. This is useful for long waits as an indication that the process is not deadlocked.

Parameters **sleep_time_seconds** (*float*) – The amount of time in seconds for this coroutine to wait before exiting. For each second that passes while waiting for this amount of time the coroutine will asyncio.sleep() for 1 second.

gate_tasks(*gate_co_or_f: Union[Coroutine[T_co, T_contra, V_co], _asyncio.Future], timeout_seconds: Optional[float], *gated_tasks*) → Tuple[_asyncio.Future, List[_asyncio.Future]]
Runs a set of tasks until a gate task completes then cancels the remaining tasks.

```
async def gated_task():
    while True:
        await asyncio.sleep(.1)

async def gate_task():
    await asyncio.sleep(1)
    return 'gate passed'

async def example():

    any_fixture = nanaimo_bar.Fixture(manager)

    gate_future, gated_futures = await any_fixture.gate_tasks(gate_task(),␣
↪None, gated_task())

    assert not gate_future.cancelled()
    assert 'gate passed' == gate_future.result()
    assert len(gated_futures) == 1
    assert gated_futures[0].cancelled()
```

Parameters

- **gate_co_or_f** (*typing.Union[typing.Coroutine, asyncio.Future]*) – The task that is expected to complete in less than timeout_seconds.

- **timeout_seconds** (*float*) – Time in seconds to wait for the gate for before raising asyncio.TimeoutError. Set to None to disable.
- **persistent_tasks** (*typing.Union[typing.Coroutine, asyncio.Future]*) – Iterable of tasks that may remain active.

Returns a tuple of the gate future and a set of the gated futures.

Return type typing.Tuple[asyncio.Future, typing.Set[asyncio.Future]]:

Raises

- ***AssertionError*** – if any of the persistent tasks exited.
- **asyncio.TimeoutError** – if the observer task does not complete within timeout_seconds.

gather(**args*, ***kwargs*) → nanaimo.Artifacts

Coroutine awaited to gather a new set of fixture artifacts.

Parameters **kwargs** – Optional arguments to override or augment the arguments provided to the *nanaimo.fixtures.Fixture* constructor

Returns A set of artifacts with the *nanaimo.Artifacts.result_code* set to indicate the success or failure of the fixture's artifact gathering activies.

Raises **asyncio.TimeoutError** – If *gather_timeout_seconds* is > 0 and *on_gather()* takes longer then this to complete or if on_gather itself raises a timeout error.

observe_tasks(*observer_co_or_f: Union[Coroutine[T_co, T_contra, V_co], _asyncio.Future], timeout_seconds: Optional[float], *persistent_tasks*) → Set[_asyncio.Future]

Allows running a set of tasks but returning when an observer task completes. This allows a pattern where a single task is evaluating the side-effects of other tasks as a gate to continuing the test or simply that a set of task should continue to run but a single task must complete.

Parameters

- **observer_co_or_f** (*typing.Union[typing.Coroutine, asyncio.Future]*) – The task that is expected to complete in less than timeout_seconds.
- **timeout_seconds** (*float*) – Time in seconds to observe for before raising asyncio.TimeoutError. Set to None to disable.
- **persistent_tasks** (*typing.Union[typing.Coroutine, asyncio.Future]*) – Iterable of tasks that may remain active.

Returns a list of the persistent tasks as futures.

Return type typing.Set[asyncio.Future]

Raises

- ***AssertionError*** – if any of the persistent tasks exited.
- **asyncio.TimeoutError** – if the observer task does not complete within timeout_seconds.

observe_tasks_assert_not_done(*observer_co_or_f: Union[Coroutine[T_co, T_contra, V_co], _asyncio.Future], timeout_seconds: Optional[float], *persistent_tasks*) → Set[_asyncio.Future]

Allows running a set of tasks but returning when an observer task completes. This allows a pattern where a single task is evaluating the side-effects of other tasks as a gate to continuing the test.

Parameters

- **observer_co_or_f** (*typing.Union[typing.Coroutine, asyncio.Future]*) – The task that is expected to complete in less than `timeout_seconds`.
- **timeout_seconds** (*float*) – Time in seconds to observe for before raising `asyncio.TimeoutError`. Set to None to disable.
- **persistent_tasks** (*typing.Union[typing.Coroutine, asyncio.Future]*) – Iterable of tasks that must remain active or `AssertionError` will be raised.

Returns a list of the persistent tasks as futures.

Return type typing.Set[asyncio.Future]

Raises

- ***AssertionError*** – if any of the persistent tasks exited.
- **asyncio.TimeoutError** – if the observer task does not complete within `timeout_seconds`.

on_gather(*args: nanaimo.Namespace*) → nanaimo.Artifacts

Coroutine awaited by a call to *gather()*. The fixture should always retrieve new artifacts when invoked leaving caching to the caller.

Parameters **args** (`nanaimo.Namespace`) – The arguments provided for the fixture instance merged with kwargs provided to the *gather()* method.

Returns A set of artifacts with the *nanaimo.Artifacts.result_code* set to indicate the success or failure of the fixture's artifact gathering activies.

Raises **asyncio.TimeoutError** – It is valid for a fixture to raise timeout errors from this method.

class nanaimo.fixtures.**SubprocessFixture**(*manager: nanaimo.fixtures.FixtureManager*, *args: Optional[nanaimo.Namespace] = None*, ***kwargs*)

Bases: *nanaimo.fixtures.Fixture*

Fixture base type that accepts a string argument `cmd` and executes it as a subprocess. Because some subprocess commands might result in huge amounts of data being sent to stdout and/or stderr this class does not capture this data by default. Instead, tests are encouraged to either filter the subprocess pipes or use the `--logfile` argument to write the output to a file in persistent storage.

Filtering is accomplished using the *stdout_filter* or *stderr_filter* property of this class. For example:

```
class MySubprocessFixture(nanaimo.fixtures.SubprocessFixture):
    '''
    Subprocess test fixture that simply calls "nait --version"
    '''

    @classmethod
    def on_visit_test_arguments(cls, arguments: nanaimo.Arguments) -> None:
        pass

    def on_construct_command(self, arguments: nanaimo.Namespace, inout_artifacts:␣
↪nanaimo.Artifacts) -> str:
        return 'nait --version'


async def example(manager):
```

(continues on next page)

(continued from previous page)

```
    subject = MySubprocessFixture(manager)

    # The accumulator does capture all stdout. Only use this if you know
    # the subprocess will produce a managable and bounded amount of output.
    filter = nanaimo.fixtures.SubprocessFixture.SubprocessMessageAccumulator()
    subject.stdout_filter = filter

    artifacts = await subject.gather()

    # In our example the subprocess produces only and exactly the Nanaimo
    # version number
    assert filter.getvalue() == nanaimo.version.__version__
```

Parameters

- **stdout_filter** – A `logging.Filter` used when gathering the subprocess.
- **stderr_filter** – A `logging.Filter` used when gathering the subprocess.

class SubprocessMessageAccumulator(*minimum_level: int = 20*)
Bases: `logging.Filter`, `_io.StringIO`

Helper class for working with *SubprocessFixture.stdout_filter()* or *SubprocessFixture.stderr_filter()*. This implementation will simply write all log messages (i.e. `logging.LogRecord.getMessage()`) to its internal buffer. Use `getvalue()` to get a reference to the buffer.

You can also subclass this method and override its `logging.Filter.filter()` method to customize your filter criteria.

Parameters **minimum_level** – The minimum loglevel to accumulate messages for.

filter(*record: logging.LogRecord*) → bool
Determine if the specified record is to be logged.

Is the specified record to be logged? Returns 0 for no, nonzero for yes. If deemed appropriate, the record may be modified in-place.

class SubprocessMessageMatcher(*pattern: Any*, *minimum_level: int = 20*)
Bases: `logging.Filter`

Helper class for working with *SubprocessFixture.stdout_filter()* or *SubprocessFixture.stderr_filter()*. This implementation will watch every log message and store any that match the provided pattern.

This matcher does not buffer all logged messages.

Parameters

- **pattern** – A regular expression to match messages on.
- **minimum_level** – The minimum loglevel to accumulate messages for.

match_count
The number of messages that matched the provided pattern.

filter(*record: logging.LogRecord*) → bool
Determine if the specified record is to be logged.

Is the specified record to be logged? Returns 0 for no, nonzero for yes. If deemed appropriate, the record may be modified in-place.

classmethod on_visit_test_arguments(*arguments: nanaimo.Arguments*) → None
Called by the environment before instantiating any *nanaimo.fixtures.Fixture* instances to register arguments supported by each type. These arguments should be portable between both `argparse` and `pytest`. The fixture is registered for this callback by returning a reference to its type from a `pytest_nanaimo_fixture_type` hook in your fixture's pytest plugin module.

on_gather(*args: nanaimo.Namespace*) → nanaimo.Artifacts

Artifacts		
key	type	Notes
`cmd`	str	The command used to execute the subprocess.
`logfile`	Optional[pathlib.Path]	A file containing stdout, stderr, and test logs

stdout_filter
A filter used when logging the stdout stream with the subprocess.

stderr_filter
A filter used when logging the stderr stream with the subprocess.

on_construct_command(*arguments: nanaimo.Namespace*, *inout_artifacts: nanaimo.Artifacts*) → str
Called by the subprocess fixture to ask the specialization to form a command given a set of arguments.

Parameters

- **arguments** (`nanaimo.Arguments`) – The arguments passed into *Fixture.on_gather()*.
- **inout_artifacts** (`nanaimo.Artifacts`) – A set of artifacts the superclass is assembling This is provided to the subclass to allow it to optionally contribute artifacts.

Returns The command to run in a subprocess shell.

class nanaimo.fixtures.**FixtureManager**(*loop: Optional[asyncio.events.AbstractEventLoop] = None*)
Bases: `object`

A simple fixture manager and a baseclass for specalized managers.

loop
The running asyncio EventLoop in use by all Fixtures. This will be the loop provided to the fixture manager in the constructor if that loop is still running otherwise the loop will be a running loop retrieved by `asyncio.get_event_loop()`. :raises RuntimeError: if no running event loop could be found.

create_fixture(*canonical_name: str*, *args: Optional[nanaimo.Namespace] = None*, *loop: Optional[asyncio.events.AbstractEventLoop] = None*) → nanaimo.fixtures.Fixture
Create a new *nanaimo.fixtures.Fixture* instance iff the `canonical_name` is a registered plugin for this process.

Parameters

- **canonical_name** (*str*) – The canonical name of the fixture to instantiate.
- **args** (`nanaimo.Namespace`) – The arguments to provide to the new instance.
- **loop** (*typing.Optional[asyncio.AbstractEventLoop]*) – An event loop to provide the fixture instance.

Raises **KeyError** – if `canonical_name` was not registered with this manager.

class nanaimo.fixtures.**PluggyFixtureManager**
Bases: object

DEPRECATED. Do not use.

7.3 nanaimo.config

Much of what Nanaimo does is to allow for intuitive, intelligent, and adaptable configuration of fixtures. This module contains internal implementations used by public types to facilitate this functionality.

class nanaimo.config.**ArgumentDefaults**(*args: Optional[Any] = None*)
Bases: object

Manages default values for Nanaimo arguments. Use in conjunction with *nanaimo.Arguments* and *nanaimo.Namespace* to populate default argument values when adding arguments to pytest or argparse and to replace missing attributes with values from configuration.

When using the Nanaimo CLI or pytest plugin you don't need to worry about this object as these modules wire up the defaults, arguments, and namespaces for you.

classmethod create_defaults_with_early_rc_config() → nanaimo.config.ArgumentDefaults
A special factory method that creates a *ArgumentDefaults* instance pulling the value of `--rcfile` directly from `sys.argv`. This allows defaults to be pulled from a config file before argument parsing is performed.

classmethod as_dict(*config_value: Union[str, List[str]]*) → Mapping[str, str]
Where a value is set as a map this method is used to parse the resulting string into a Python dictionary:

```
[nanaimo]
foo =
    a = 1
    b = 2
    c = 3
```

Given the above configuration the value of `foo` can be read as a dictionary as such:

```
foo_dict = ArgumentDefaults.as_dict(config['foo'])

assert foo_dict['b'] == '2'
```

7.4 nanaimo.pytest.plugin

Nanaimo provides a collection of pytest fixtures all defined in this module.

```
# In my_namespace/__init__.py

class MyTestFixture(nanaimo.fixtures.Fixture):

    fixture_name = 'my_fixture_name'
    argument_prefix = 'mfn'

    @classmethod
    def on_visit_test_arguments(cls, arguments: nanaimo.Arguments) -> None:
        pass
```

(continues on next page)

(continued from previous page)

```
    async def on_gather(self, args: nanaimo.Namespace) -> nanaimo.Artifacts:
        artifacts = nanaimo.Artifacts()
        # Do your on-target testing here and store results in nanaimo.Artifacts.
        return artifacts


def pytest_nanaimo_fixture_type() -> typing.Type['nanaimo.fixtures.Fixture']:
    '''
    This is required to provide the fixture to pytest as a fixture.
    '''
    return MyTestFixture
```

In your setup.cfg you'll first need to register nanaimo with pytest

```
[options]
    pytest11 =
        pytest_nanaimo = nanaimo.pytest.plugin
```

then you'll need to add your fixture's namespace:

```
[options]
    pytest11 =
        pytest_nanaimo = nanaimo.pytest.plugin
        pytest_nanaimo_plugin_my_fixture = my_namspace
```

Note: For Nanaimo the key for your plugin in the pytest11 map is unimportant. It must be unique but nothing else about it will be visible to your tests unless you integrate more deeply with pytest. The fixture you expose from your plugin via `pytest_nanaimo_fixture_type` will be named for the canonical name of your *`Fixture`* (in this case, "my_fixture_name").

You can add as many additional nanaimo plugins as you want but you can only have one *`Fixture`* in each module.

Where you don't need your Nanaimo fixture to be redistributable you may also use standard pytest fixture creation by using the *`nanaimo_fixture_manager`* and *`nanaimo_arguments`* fixtures supplied by the core nanaimo pytest plugin. For example, assuming you have `nanaimo.pytest.plugin` in your setup.cfg (see above) you can do something like this in a `conftest.py`

```
@pytest.fixture
def my_fixture_name(nanaimo_fixture_manager, nanaimo_arguments) -> 'nanaimo.fixtures.
↪Fixture':
    '''
    It is considered a best-practice to always name your pytest.fixture method the
    same as your Fixture's canonical name (i.e. fixture_name).
    '''
    return MyTestFixture(nanaimo_fixture_manager, nanaimo_arguments)
```

`nanaimo.pytest.plugin.`**`create_pytest_fixture`**(*request: Any*, *fixture_name: str*) → nanaimo.fixtures.Fixture

DEPRECATED. Use the appropriate *`FixtureManager`* and its *`FixtureManager.create_fixture`* method instead.

`nanaimo.pytest.plugin.`**`nanaimo_fixture_manager`**(*request: Any*) → nanaimo.fixtures.FixtureManager

Provides a default *`FixtureManager`* to a test.

```
def test_example(nanaimo_fixture_manager: nanaimo.Namespace) -> None:
    common_loop = nanaimo_fixture_manager.loop
```

Parameters **pytest_request** (*_pytest.fixtures.FixtureRequest*) – The request object passed into the pytest fixture factory.

Returns A new fixture manager.

Return type *nanaimo.fixtures.FixtureManager*

nanaimo.pytest.plugin.**nanaimo_arguments**(*request: Any*) → nanaimo.Namespace

Exposes the commandline arguments and defaults provided to a test.

```
def test_example(nanaimo_arguments: nanaimo.Namespace) -> None:
    an_argument = nanaimo_arguments.some_arg
```

Parameters **pytest_request** (*_pytest.fixtures.FixtureRequest*) – The request object passed into the pytest fixture factory.

Returns A namespace with the pytest commandline args added per the documented rules.

Return type *nanaimo.Namespace*

nanaimo.pytest.plugin.**nanaimo_log**(*request: Any*) → logging.Logger

Provides the unit tests with a logger configured for use with the Nanaimo framework.

Note: For now this is just a Python logger. Future revisions may add capabilities like the ability to log to a display or otherwise provide feedback to humans about the status of a test.

```
def test_example(nanaimo_log: logging.Logger) -> None:
    nanaimo_log.info('Hiya')
```

It's recommended that all Nanaimo tests configure logging in a tox.ini, pytest.ini, or pyproject.toml (when this is supported). For example, the following section in tox.ini would enable cli logging for nanaimo tests:

```
[pytest]
log_cli = true
log_cli_level = DEBUG
log_format = %(asctime)s %(levelname)s %(name)s: %(message)s
log_date_format = %Y-%m-%d %H:%M:%S
```

Parameters **pytest_request** (*_pytest.fixtures.FixtureRequest*) – The request object passed into the pytest fixture factory.

Returns A logger for use by Nanaimo tests.

Return type logging.Logger

nanaimo.pytest.plugin.**assert_success**(*artifacts: nanaimo.Artifacts*) → nanaimo.Artifacts

Syntactic sugar to allow more fluent handling of `fixtures.Fixture.gather()` artifacts. For example:

```
from nanaimo.pytest.plugin import assert_success

async def test_my_fixture():

```

(continues on next page)

(continued from previous page)

```
    artifacts = assert_success(await fixture.gather())

    # Now we can use the artifacts. If the gather had returned
    # non-zero for the result_code an assertion error would have
    # been raised.
```

Parameters **artifacts** (nanaimo.Artifacts) – The artifacts to assert on.

Returns artifacts (for convenience).

Return type *nanaimo.Artifacts*()

nanaimo.pytest.plugin.**assert_success_if**(*artifacts: nanaimo.Artifacts, conditional: Callable[[nanaimo.Artifacts], bool]*) → nanaimo.Artifacts

Syntactic sugar to allow more fluent handling of `fixtures.Fixture.gather()` artifacts but with a user-supplied conditional.

```
from nanaimo.pytest.plugin import assert_success_if

async def test_my_fixture():

    def fail_if_no_foo(artifacts: Artifacts) -> bool:
        return 'foo' in artifacts

    artifacts = assert_success_if(await fixture.gather(), fail_if_no_foo)

    print('artifacts have foo. It's value is {}'.format(artifacts.foo))
```

Parameters

- **artifacts** (nanaimo.Artifacts) – The artifacts to assert on.
- **conditiona** – A method called to evaluate gathered artifacts iff `Artifacts.result_code` is 0. Return False to trigger an assertion, True to pass.

Returns artifacts (for convenience).

Return type *nanaimo.Artifacts*()

class nanaimo.pytest.plugin.**PytestFixtureManager**(*pluginmanager: _pytest.config.PytestPluginManager, loop: Optional[asyncio.events.AbstractEventLoop] = None*)

Bases: *nanaimo.fixtures.FixtureManager*

FixtureManager implemented using pytest plugin APIs.

create_fixture(*canonical_name: str, args: Optional[nanaimo.Namespace] = None, loop: Optional[asyncio.events.AbstractEventLoop] = None*) → nanaimo.fixtures.Fixture

Create a new *nanaimo.fixtures.Fixture* instance iff the `canonical_name`` is a registered plugin for this process.

Parameters

- **canonical_name** (*str*) – The canonical name of the fixture to instantiate.
- **args** (nanaimo.Namespace) – The arguments to provide to the new instance.

- **loop** (*typing.Optional[asyncio.AbstractEventLoop]*) – An event loop to provide the fixture instance.

Raises **KeyError** – if `canonical_name` was not registered with this manager.

`nanaimo.pytest.plugin.`**`is_nait_mode`**`()` → bool
A special mode enabled by the 'nait' commandline that discards all collected tests and inserts a single test item driven by nait. This mode is used to interact with Nanaimo fixtures from a commandline.

`nanaimo.pytest.plugin.`**`pytest_addoption`**`(`*parser: _pytest.config.argparsing.Parser*, *pluginmanager: _pytest.config.PytestPluginManager*`)` → None
See `_pytest.hookspec.pytest_addoption()` for documentation. Also see the "Writing Plugins" guide.

`nanaimo.pytest.plugin.`**`pytest_addhooks`**`(`*pluginmanager*`)`
See `_pytest.hookspec.pytest_addhooks()` for documentation. Also see the "Writing Plugins" guide.

`nanaimo.pytest.plugin.`**`pytest_collection`**`(`*session: _pytest.main.Session*`)`
See `_pytest.hookspec.pytest_collection_modifyitems()` for documentation. Also see the "Writing Plugins" guide.

`nanaimo.pytest.plugin.`**`pytest_sessionstart`**`(`*session: _pytest.main.Session*`)` → None
See `_pytest.hookspec.pytest_sessionstart()` for documentation. Also see the "Writing Plugins" guide.

`nanaimo.pytest.plugin.`**`pytest_runtest_setup`**`(`*item: _pytest.nodes.Item*`)` → None
See `_pytest.hookspec.pytest_runtest_setup()` for documentation. Also see the "Writing Plugins" guide.

`nanaimo.pytest.plugin.`**`pytest_runtest_teardown`**`(`*item: _pytest.nodes.Item*, *nextitem: _pytest.nodes.Item*`)` → None
See `_pytest.hookspec.pytest_teardown()` for documentation. Also see the "Writing Plugins" guide.

`nanaimo.pytest.plugin.`**`pytest_sessionfinish`**`(`*session: _pytest.main.Session*, *exitstatus: int*`)` → None
See `_pytest.hookspec.pytest_sessionfinish()` for documentation. Also see the "Writing Plugins" guide.

`nanaimo.pytest.plugin.`**`pytest_report_header`**`(`*config: _pytest.config.Config*, *startdir*`)` → List[str]
See `_pytest.hookspec.pytest_sessionfinish()` for documentation. Also see the "Writing Plugins" guide.

`nanaimo.pytest.plugin.`**`pytest_terminal_summary`**`(`*terminalreporter: _pytest.terminal.TerminalReporter*, *exitstatus: int*, *config: _pytest.config.Config*`)` → None
See `_pytest.hookspec.pytest_sessionfinish()` for documentation. Also see the "Writing Plugins" guide.

7.5 nanaimo.builtin

Built-in *nanaimo.fixtures.Fixture* objects for common scenarios. See individual fixture documentation for use.

7.6 nanaimo.builtin.nanaimo_bar

class nanaimo.builtin.nanaimo_bar.**Fixture**(*manager: nanaimo.fixtures.FixtureManager*, *args: Optional[nanaimo.Namespace] = None*, ***kwargs*)

Bases: *nanaimo.fixtures.Fixture*

A trivial plugin. Returns an callable artifact named "eat" that logs a yummy info message when invoked.

classmethod on_visit_test_arguments(*arguments: nanaimo.Arguments*) → None

Called by the environment before instantiating any *nanaimo.fixtures.Fixture* instances to register arguments supported by each type. These arguments should be portable between both `argparse` and `pytest`. The fixture is registered for this callback by returning a reference to its type from a `pytest_nanaimo_fixture_type` hook in your fixture's pytest plugin module.

on_gather(*args: nanaimo.Namespace*) → nanaimo.Artifacts

Create a delicious function in the artifacts to eat.

Returned Artifacts		
key	type	Notes
eat	Callable[[],None]	A function that logs a message
bar_{number}	str	A string formed from an argument `bar_number`. This allows testing ordering of concurrent operations.

7.7 nanaimo.builtin.nanaimo_gather

class nanaimo.builtin.nanaimo_gather.**Fixture**(*manager: nanaimo.fixtures.FixtureManager*, *args: Optional[nanaimo.Namespace] = None*, ***kwargs*)

Bases: *nanaimo.fixtures.Fixture*

This fixture takes a list of other fixtures and runs them concurrently returning a *nanaimo.Artifacts.combine()* ed set of *nanaimo.Artifacts*.

You can use this fixture directly in your unit tests:

```
async def example1() -> Artifacts:

    bar_one = nanaimo_bar.Fixture(manager)
    bar_two = nanaimo_bar.Fixture(manager)
    gather_fixture = nanaimo_gather.Fixture(manager)

    return await gather_fixture.gather(
        gather_coroutine=[
            bar_one.gather(bar_number=1),
            bar_two.gather(bar_number=2)
        ]
    )
```

You can also use the –gather-coroutines argument to specify fixtures by name:

```
nait nanaimo_gather --gather-coroutine nanaimo_bar --gather-coroutine nanaimo_bar
```

classmethod on_visit_test_arguments(*arguments: nanaimo.Arguments*) → None

Called by the environment before instantiating any *nanaimo.fixtures.Fixture* instances to register arguments supported by each type. These arguments should be portable between both `argparse`

and pytest. The fixture is registered for this callback by returning a reference to its type from a pytest_nanaimo_fixture_type hook in your fixture's pytest plugin module.

on_gather(*args: nanaimo.Namespace*) → nanaimo.Artifacts
Multi-plex fixture

7.8 nanaimo.builtin.nanaimo_cmd

class nanaimo.builtin.nanaimo_cmd.**Fixture**(*manager: nanaimo.fixtures.FixtureManager*, *args: Optional[nanaimo.Namespace] = None*, ***kwargs*)

Bases: *nanaimo.fixtures.SubprocessFixture*

Fixture that accepts a string argument cmd and executes it as a subprocess.

```
async def list_directory() -> Artifacts:

    cmd = nanaimo_cmd.Fixture(manager)

    return await cmd.gather(
        cmd_shell = ('dir' if os.name == 'nt' else 'ls -la')
    )
```

classmethod on_visit_test_arguments(*arguments: nanaimo.Arguments*) → None
Called by the environment before instantiating any *nanaimo.fixtures.Fixture* instances to register arguments supported by each type. These arguments should be portable between both argparse and pytest. The fixture is registered for this callback by returning a reference to its type from a pytest_nanaimo_fixture_type hook in your fixture's pytest plugin module.

on_construct_command(*arguments: nanaimo.Namespace*, *inout_artifacts: nanaimo.Artifacts*) → str
Called by the subprocess fixture to ask the specialization to form a command given a set of arguments.

Parameters

- **arguments** (nanaimo.Arguments) – The arguments passed into Fixture.on_gather().
- **inout_artifacts** (nanaimo.Artifacts) – A set of artifacts the superclass is assembling This is provided to the subclass to allow it to optionally contribute artifacts.

Returns The command to run in a subprocess shell.

7.9 nanaimo.builtin.nanaimo_scp

class nanaimo.builtin.nanaimo_scp.**Fixture**(*manager: nanaimo.fixtures.FixtureManager*, *args: Optional[nanaimo.Namespace] = None*, ***kwargs*)

Bases: *nanaimo.fixtures.SubprocessFixture*

This fixture assumes that scp is available and functional on the system.

classmethod on_visit_test_arguments(*arguments: nanaimo.Arguments*) → None
Called by the environment before instantiating any *nanaimo.fixtures.Fixture* instances to register arguments supported by each type. These arguments should be portable between both argparse and pytest. The fixture is registered for this callback by returning a reference to its type from a pytest_nanaimo_fixture_type hook in your fixture's pytest plugin module.

on_construct_command(*args: nanaimo.Namespace*, *inout_artifacts: nanaimo.Artifacts*) → str
Form the upload command.

7.10 nanaimo.builtin.nanaimo_ssh

class nanaimo.builtin.nanaimo_ssh.**Fixture**(*manager: nanaimo.fixtures.FixtureManager*, *args: Optional[nanaimo.Namespace] = None*, ***kwargs*)

Bases: *nanaimo.fixtures.SubprocessFixture*

This fixture assumes that ssh is available and functional on the system.

classmethod on_visit_test_arguments(*arguments: nanaimo.Arguments*) → None
Called by the environment before instantiating any *nanaimo.fixtures.Fixture* instances to register arguments supported by each type. These arguments should be portable between both argparse and pytest. The fixture is registered for this callback by returning a reference to its type from a pytest_nanaimo_fixture_type hook in your fixture's pytest plugin module.

on_construct_command(*args: nanaimo.Namespace*, *inout_artifacts: nanaimo.Artifacts*) → str
Called by the subprocess fixture to ask the specialization to form a command given a set of arguments.

Parameters

- **arguments** (nanaimo.Arguments) – The arguments passed into Fixture.on_gather().
- **inout_artifacts** (nanaimo.Artifacts) – A set of artifacts the superclass is assembling This is provided to the subclass to allow it to optionally contribute artifacts.

Returns The command to run in a subprocess shell.

7.11 nanaimo.builtin.nanaimo_serial_watch

class nanaimo.builtin.nanaimo_serial_watch.**Fixture**(*manager: nanaimo.fixtures.FixtureManager*, *args: nanaimo.Namespace*, ***kwargs*)

Bases: *nanaimo.fixtures.Fixture*

Gathers a log over a serial connection until a given pattern is matched.

classmethod on_visit_test_arguments(*arguments: nanaimo.Arguments*) → None
Called by the environment before instantiating any *nanaimo.fixtures.Fixture* instances to register arguments supported by each type. These arguments should be portable between both argparse and pytest. The fixture is registered for this callback by returning a reference to its type from a pytest_nanaimo_fixture_type hook in your fixture's pytest plugin module.

on_gather(*args: nanaimo.Namespace*) → nanaimo.Artifacts
Watch the logs until the pattern matches.

Returned Artifacts		
key	type	Notes
match	re.MatchObject	The match if result_code is 0
matched_line	str	The full line matched if result_code is 0

7.12 nanaimo.connections

Connections are built-in async abstractions using standard communication protocols like UART, I2C, CAN, TCP/IP, etc. `Instrument` and *nanaimo.fixtures.Fixture* classes use connections to bind to physical hardware.

class nanaimo.connections.**TimestampedLine**(*line_text: object*)
Bases: str

A line of text with an associated timestamp. This is a subclass of string so the object may be treated as a string without conversion.

timestamp_seconds
The timestamp for the line in fractional seconds. For example, this would be the time the line of text was received when this type is returned for a getter.

class nanaimo.connections.**AbstractSerial**
Bases: object

Abstract base class for a serial communication channel.

class nanaimo.connections.**AbstractAsyncSerial**(*loop: Optional[asyncio.events.AbstractEventLoop] = None*)
Bases: *nanaimo.connections.AbstractSerial*

Abstract base class for a serial communication channel that provides asynchronous methods.

time() → float
Get the current, monotonic time, in fractional seconds, using the same clock used for receive timestamps.

get_line(*timeout_seconds: Optional[float] = None*) → nanaimo.connections.TimestampedLine
Get a line of text.

Parameters **timeout_seconds** (*float*) – Time in fractional seconds to wait for input.

Returns A line of text with the time it was received at.

Return type *TimestampedLine*

Raises **asyncio.TimeoutError** – If a full line of text was not received within the specified timeout period.

put_line(*input_line: str*, *timeout_seconds: Optional[float] = None*) → float
Put a line of text to the serial device.

Parameters

- **input_line** (*str*) – The line to put.
- **timeout_seconds** (*float*) – Fractional seconds to block for if the input buffer is full. If the buffer does not become available within this time then `asyncio.TimeoutError` is raised. Use 0 to block forever.

Returns The monotonic system time that the line was put into the serial buffers at (see *time()*).

Raises **asyncio.TimeoutError** – If an input buffer did not become available within the specified timeout.

7.13 nanaimo.connections.uart

class nanaimo.connections.uart.**ConcurrentUart**(*serial_port: serial.serialposix.Serial, loop: Optional[asyncio.events.AbstractEventLoop] = None, eol: str = 'rn', echo: bool = False*)

Bases: *nanaimo.connections.AbstractAsyncSerial*

Buffers serial input on another thread and provides line-oriented access to/from the tty via synchronized queues.

readline() → Optional[nanaimo.connections.TimestampedLine]

Get a line of text from the receive buffers.

Returns A line of text with the time it was received at.

Return type *TimestampedLine*

writeline(*input_line: str, end: Optional[str] = None*) → float

Enqueue a line of text to be transmitted on the serial device.

Parameters

- **input_line** (*str*) – The line to put.
- **end** (*typing.Optional[str]*) – Line ending (overrides the default `eol()`).

Returns The monotonic system time that the line was put into the serial buffers at (see `time()`).

7.14 nanaimo.instruments

This module contains control and communication objects for popular instruments useful to the type of testing Nanaimo is focused on. Where possible these async interfaces will use pure-python to communicate directly with an instrument using known protocols and well supported communication busses (e.g. ethernet, uart, or CAN), however, some instruments will require the installation of a vendor CLI for the Nanaimo automation to use.

Warning: Do not merge third-party code into the Nanaimo repo or provide Nanaimo abstractions that are incompatible with the licensing of this software.

Fig. 7.1 shows the Nanaimo jlink instrument using Segger's JLink Commander as a subprocess to upload a firmware image to a microcontroller.

7.15 nanaimo.instruments.jlink

class nanaimo.instruments.jlink.**ProgramUploader**(*manager: nanaimo.fixtures.FixtureManager, args: Optional[nanaimo.Namespace] = None, **kwargs*)

Bases: *nanaimo.fixtures.SubprocessFixture*

JLinkExe fixture that loads a given hexfile to a target device.

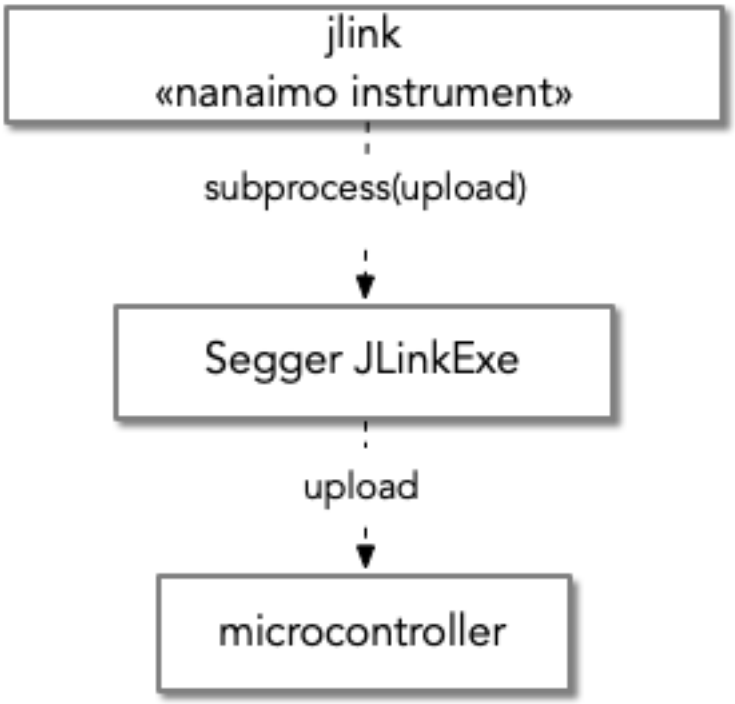

Fig. 7.1: Example of a Nanaimo Segger JLink instrument using the JLinkExe CLI as a Python subprocess.

classmethod on_visit_test_arguments(*arguments: nanaimo.Arguments*) → None
Called by the environment before instantiating any *nanaimo.fixtures.Fixture* instances to register arguments supported by each type. These arguments should be portable between both `argparse` and `pytest`. The fixture is registered for this callback by returning a reference to its type from a `pytest_nanaimo_fixture_type` hook in your fixture's pytest plugin module.

on_construct_command(*arguments: nanaimo.Namespace*, *inout_artifacts: nanaimo.Artifacts*) → str
Construct a command to upload (aka "flash") a firmware to a target device using Segger's JLink Commander program with the assumption that a Segger debug probe like the JLink is attached to the system.
+————————-+————————————+————————————————————+ | **Artifacts** | | + ————————-+————————————+————————————————————+ | key | type | Notes | +================+==+========================== | `tmpfile` | Optional[tempfile.NamedTemporaryFile] | A temporary file containing the jlink script to | | | | execute with expanded template parameters. | +————————- +————————————+————————————————————+ | `scriptfile` | Optional[str] | A path to a script file used to invoke jlink. | | | This file can contain template parameters. | +————————-+————————————+————————————————————+

7.16 nanaimo.instruments.bkprecision

class nanaimo.instruments.bkprecision.**Series1900BUart**(*manager: nanaimo.fixtures.FixtureManager*, *args: nanaimo.Namespace*, ***kwargs*)
Bases: *nanaimo.fixtures.Fixture*

Control of a 1900B series BK Precision power supply via UART.

classmethod mode_to_text(*mode: int*) → str
Get a two-character textual representation for a given power supply mode.

UartFactoryType = typing.Callable[[typing.Union[str, pathlib.Path]], typing.Any]
The serial port factory type for this instrument.

classmethod default_serial_port(*port: Union[str, pathlib.Path]*) → Generator[nanaimo.connections.AbstractAsyncSerial, None, None]
Creates a serial connection to the given port using the default settings for a BK Precision Series 1900B power supply.

classmethod on_visit_test_arguments(*arguments: nanaimo.Arguments*) → None
Called by the environment before instantiating any *nanaimo.fixtures.Fixture* instances to register arguments supported by each type. These arguments should be portable between both `argparse` and `pytest`. The fixture is registered for this callback by returning a reference to its type from a `pytest_nanaimo_fixture_type` hook in your fixture's pytest plugin module.

is_volage_above_on_threshold(*voltage: float*) → bool
Deprecated misspelling. See *is_voltage_above_on_threshold()* for correct method.

is_voltage_above_on_threshold(*voltage: float*) → bool
Return if a given voltage is above the configured threshold for the high/on/rising voltage for this fixture.

Raises **ValueError** – if no target voltage could be determined.

is_volage_below_off_threshold(*voltage: float*) → bool
Deprecated misspelling. See *is_voltage_below_off_threshold()* for correct method.

is_voltage_below_off_threshold(*voltage: float*) → bool
Return if a given voltage is below the configured threshold for the low/off/falling voltage for this fixture.

on_gather(*args: nanaimo.Namespace*) → nanaimo.Artifacts
Send a command to the instrument and return the result. :param str command: Send one of the following commands:

Command	Action	Returns
'1'	Turn on output voltage	'OK' or error text.
'0'	Turn off output voltage	'OK' or error text
'r'	Send a stream of <cr> characters	(NA)
'?'	Read the front panel display	Display voltage, current, and status (ON or OFF)

7.17 nanaimo.instruments.saleae

TODO: See https://github.com/ppannuto/python-saleae/blob/master/saleae/saleae.py for the command strings. They don't seem to be documented anywhere else.

class nanaimo.instruments.saleae.**Fixture**(*manager: nanaimo.fixtures.FixtureManager*, *args: Optional[nanaimo.Namespace] = None*, ***kwargs*)
Bases: *nanaimo.fixtures.Fixture*

This fixture controls a Saleae logic analyser attached to the system via USB.

Warning: Stubbed-out implementation This fixture doesn't do anything yet either then the most naive query possible of the Saleae. This would be a great first contribution to the Nanaimo project.

classmethod on_visit_test_arguments(*arguments: nanaimo.Arguments*) → None
Called by the environment before instantiating any *nanaimo.fixtures.Fixture* instances to register arguments supported by each type. These arguments should be portable between both argparse and pytest. The fixture is registered for this callback by returning a reference to its type from a pytest_nanaimo_fixture_type hook in your fixture's pytest plugin module.

on_gather(*args: nanaimo.Namespace*) → nanaimo.Artifacts
Coroutine awaited by a call to gather(). The fixture should always retrieve new artifacts when invoked leaving caching to the caller.

Parameters **args** (nanaimo.Namespace) – The arguments provided for the fixture instance merged with kwargs provided to the gather() method.

Returns A set of artifacts with the *nanaimo.Artifacts.result_code* set to indicate the success or failure of the fixture's artifact gathering activies.

Raises **asyncio.TimeoutError** – It is valid for a fixture to raise timeout errors from this method.

7.18 nanaimo.instruments.ykush

TODO: See https://www.learn.yepkit.com/reference/ykushcmd-reference-ykush/1/2 for the command strings.

class nanaimo.instruments.ykush.**Fixture**(*manager: nanaimo.fixtures.FixtureManager*, *args: Optional[nanaimo.Namespace] = None*, ***kwargs*)

Bases: *nanaimo.fixtures.SubprocessFixture*

Fixture for controlling Yepkit USB hubs with switchable power. For example the YKUSH3 is a 3-port USB-3 hub that allows individual control of the power rails for each port.

This is a subprocess fixture that requires the `ykushcmd` program is available in the subprocess environment (see Yepkit's documentation for how to build this from source). All arguments can be overridden via the fixtures gather method. The supported commands are:

command	example	Description
`yku-all-on`	`await nanaimo_instr_ykush.gather(yku_all_on=True)`	Turn on power to all ports on the YKUSH.
`yku-all-off`	`await nanaimo_instr_ykush.gather(yku_all_off=True)`	Turn off power to all ports on the YKUSH.
`yku-command`	`await nanaimo_instr_ykush.gather(yku_command='-l')`	Pass-through any command to ykushcmd.

classmethod on_visit_test_arguments(*arguments: nanaimo.Arguments*) → None

Called by the environment before instantiating any *nanaimo.fixtures.Fixture* instances to register arguments supported by each type. These arguments should be portable between both `argparse` and `pytest`. The fixture is registered for this callback by returning a reference to its type from a `pytest_nanaimo_fixture_type` hook in your fixture's pytest plugin module.

on_construct_command(*arguments: nanaimo.Namespace*, *inout_artifacts: nanaimo.Artifacts*) → str

7.19 nanaimo.parsers

The parser module contains parsers for common reporting formats used by instruments or test frameworks. These should be broken out from instruments where appropriate to ensure reuse accross instrument families and communication busses.

7.20 nanaimo.parsers.gtest

class nanaimo.parsers.gtest.**Parser**(*timeout_seconds: float*, *loop: Optional[asyncio.events.AbstractEventLoop] = None*)

Bases: `object`

Uses a given monitor to watch for google test results.

CHAPTER 8

Contributor Notes

Hi! Thanks for contributing. This page contains all the details about getting your dev environment setup.

Note: This is documentation for contributors developing nanaimo. If you are a user of this software you can ignore everything here.

8.1 Tools

8.1.1 tox

I highly recommend using a virtual environment when doing python development and we use tox to manage this for us. This'll save you hours of lost productivity the first time it keeps you from pulling in an unexpected dependency from your global python environment. You can install tox from brew on osx or apt-get on Linux. Here's how to use tox for local development:

```
tox -e local
source .tox/local/bin/activate
```

Note: Did your lint rule in tox fail? You can use `tox -e autoformat` to automatically correct formatting errors but **be careful!** This will modify your files in-place.

8.1.2 Visual Studio Code

Provided is a .vscode folder with recommended extensions and settings to get you started. Cd into the Nanaimo repository, source your virtualenv (see tox) and then launch vscode using *code* ..

8.2 Running The Tests

To run the full suite of tox tests locally you'll need docker. Once you have docker installed and running do:

```
docker pull uavcan/toxic:py35-py38-sq
docker run --rm -it -v $PWD:/repo uavcan/toxic:py35-py38-sq
tox
```

To run tests for a single version of python specify the environment as such

```
tox -e py36-test
```

And to run a single test for a single version of python do:

```
tox -e py36-test -- -k test_program_uploader_failure
```

8.2.1 Sybil Doctest

This project makes extensive use of Sybil doctests. These take the form of docstrings with a structure like thus:

```
.. invisible-code-block: python

    from fnmatch import fnmatch

.. code-block:: python

    # Use fnmatch to filter
    files = [
        'one.py',
        'two.rst',
        'three.png',
        'four.py'
    ]

>>> [fnmatch(f, '*.py') for f in files]  # noqa : F821
[True, False, False, True]
```

The invisible code block is executed but not displayed in the generated documentation and, conversely, `code-block` is both rendered using proper syntax formatting in the documentation and executed. REPL works the same as it does for `doctest` but `assert` is also a valid way to ensure the example is correct especially if used in a trailing `invisible-code-block`.

These tests are run as part of the regular pytest build. You can see the Sybil setup in the `conftest.py` found under the root directory but otherwise shouldn't need to worry about it. The simple rule is; if the docstring ends up in the rendered documentation then your `code-block` tests will be executed as unit tests.

8.3 Running Reports and Generating Docs

8.3.1 Documentation

We rely on read the docs to build our documentation from github but we also verify this build as part of our tox build. This means you can view a local copy after completing a full, successful test run (See *Running The Tests*) or do `docker run --rm -t -v $PWD:/repo uavcan/toxic:py35-py38-sq /bin/sh -c "tox`

`-e docs"` to build the docs target. You can open the index.html under .tox/docs/tmp/index.html or run a local web-server:

```
python -m http.server --directory .tox/docs/tmp &
open http://localhost:8000/index.html
```

8.3.2 Coverage

We generate a local html coverage report. You can open the index.html under .tox/report/tmp or run a local web-server:

```
python -m http.server --directory .tox/report/tmp &
open http://localhost:8000/index.html
```

8.3.3 Mypy

At the end of the mypy run we generate the following summaries:

- .tox/mypy/tmp/mypy-report-lib/index.txt
- .tox/mypy/tmp/mypy-report-script/index.txt

CHAPTER 9

Nanaimo: Hardware-In-the-Loop Unit Testing

Note: Nanaimo is alpha software and will remain so until we bump its version to 1.0.0 or greater. We will not knowingly break compatibility within a minor revision but we will break compatibility a few more times between minor revisions until beta is declared. Because of this you should depend on a minor version explicitly. For example

```
nanaimo ~= 0.1
```

Nanaimo is a set of utilities and plugins designed to enable integration of hardware test apparatuses with pytest. This can allow on-target tests to run as part of continuous integration pipelines like Buildkite, Bamboo, or Jenkins.

Nanaimo is designed to enable testing of software-defined, physical components in isolation to provide pre-integration verification of software interfaces and behavioral contracts. It adapts asynchronous control and monitoring of these components to fit familiar testing idioms (e.g. x-unit testing) using the popular python test framework, pytest.

Nanaimo is *not* a simulation framework and is not designed to support the complexity of a full hardware-in-the-loop platform. Instead it's focused on testing small integrations with a few hardware components and instruments using concepts, syntax, and frameworks familiar to software engineers. Examples of these small integrations might include verifying a SPI driver for a microcontroller or ensuring the upload time for a serial bootloader meets expected Key-Performance-Indicators (KPIs). To do this Nanaimo abstractions provide async interfaces to hardware either directly using available communication protocols (e.g. serial or IP networks) or by invoking a CLI provided by the instrument vendor. Because of this latter use case some instruments will require additional programs be available in a test environment.

This design is an amalgam of the TLYF (Test Like You Fly) methodology and the Swiss cheese model of failure analysis. Specifically; Nanaimo facilitates testing on actual or representative hardware for the first integration of software into a part or subassembly. Traditionally software engineers were responsible only for unit-testing and Software-In-the-Loop (SIL) simulation of their code. Nanaimo encourages software engineers to also provide hardware integration tests by enabling Hardware-In-the-Loop continuous-integration (HIL-CI, perhaps?).

Note: Nanaimo is named after Nanaimo bars which are about the best things humans have ever invented.

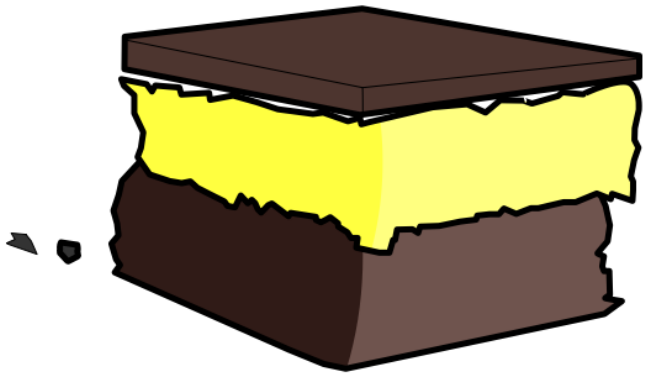

Fig. 9.1: A delicious Python treat that makes on-target testing sweet and satisfying.

Fig. 9.2: Example of S32K dev boards attached to Raspberry PI CI workers running the Buildkite agent and using Nanaimo.

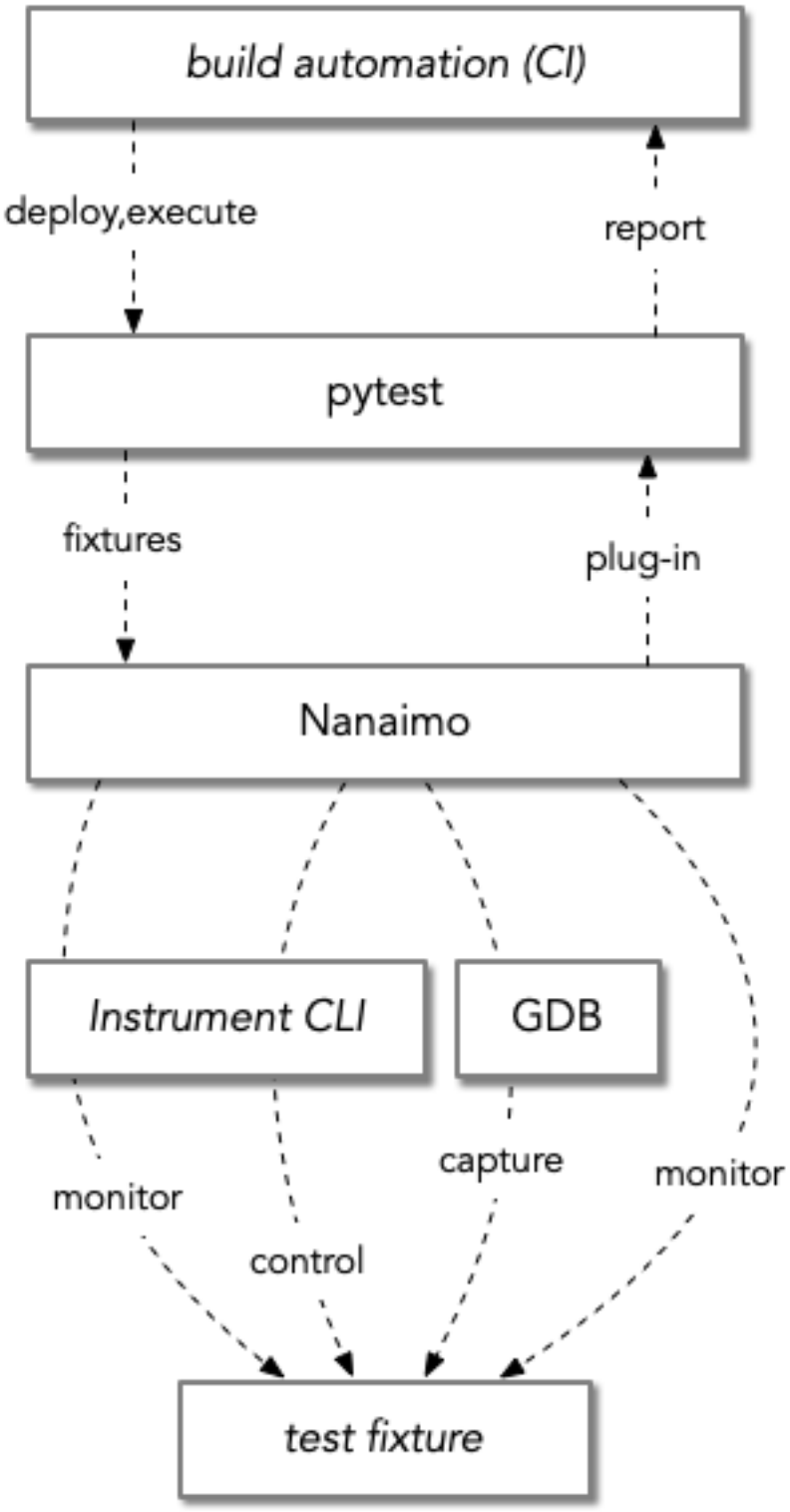

Fig. 9.3: Block diagram of Nanaimo's relationship to other components of a typical software build and test pipeline.

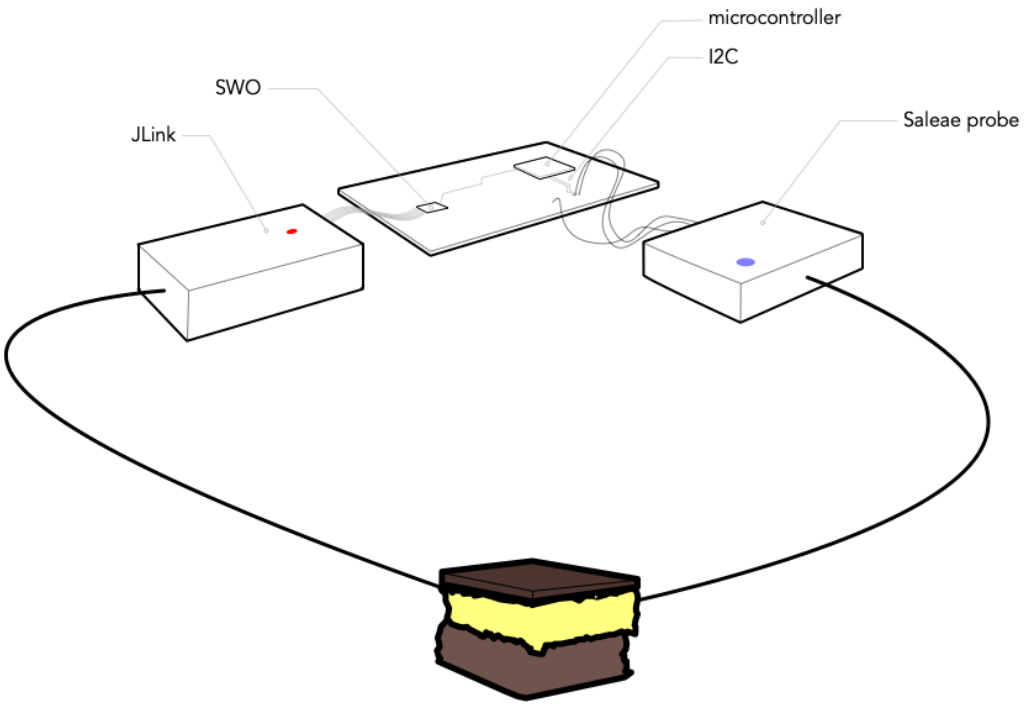

Fig. 9.4: Example scenario using Nanaimo to test an I2C driver for a microcontroller.

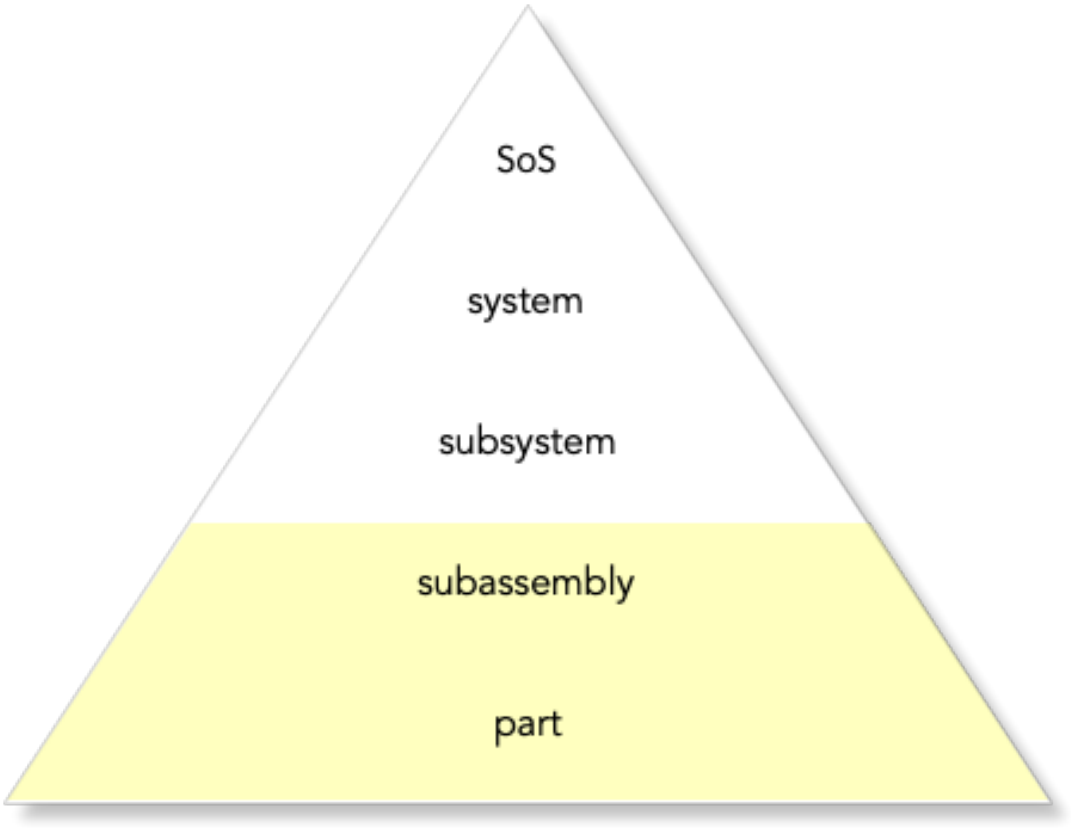

Fig. 9.5: Hierarchy of system testing. Nanaimo focuses on part and subassembly testing.

9.1 License

Python Module Index

n

Index

I

L

M

N

O

P

R

S

T

U

V

W