SILF Experiment API Documentation

Release 0.0

SILF Collaboration

Apr 18, 2018

Contents

CHAPTER 1

`device` Package

It packages api for single device.

CHAPTER 2

_const Module

silf.backend.commons.device._const.**DEVICE_STATES = ('off', 'stand-by', 'ready', 'running',**

Touple containing all allowable device states.

CHAPTER 3

_device Module

This is a api for a device.

class silf.backend.commons.device._device.**Device**(*device_id='default'*, *config_file=None*)

Bases: object

Defines plugin for a particular device in the experiment.

All methods are blocking, that is should block the current thread until finished.

Note: Instances of this object don't need to use any synchronization, they will always be called from single thread, This instance will be constructed used and destroyed on single process.

Warning: All methods should exit relatively fast.

Warning: Both method parameters and responses should be pickleable, these will be travelling between process boundaries.

MAIN_LOOP_INTERVAL = 0.1
Interval between invocations of main loop. Represents number of seconds as float.

apply_settings(*settings*)
Applies some set of settings to this device.

Parameters **settings** (*dict*) – Settings to be applied, it is already validated by the IDeviceManager.

Raises

- **InvalidStateException** – If device is in invalid state (that is not STAND_BY or RUNNING)

- **DeviceRuntimeException** – If any exception occours

Returns None

Return type None

logger

Returns Logger instance attached to this device. Utility method, you may use whatsoever logger you want

loop_iteration()
Perform an iteration of main experiment loop. Should terminate quickly,

Raises **DeviceRuntimeException** – If any exception occours

Returns If returned value is *False* or *None* next iteration of this method will be scheduled after *MAIN_LOOP_INTERVAL* seconds, it result is true it will be sheduled earlier (after at most one command from controller was performed);

perform_diagnostics(*diagnostics_level='short'*)
Performs diagnostics on the device. Can be ran if this device is OFF. or STAND_BY.

Parameters **diagnostics_level** (*str*) – Whether diagnostisc should be thororough or not, must be in DEVICE_STATES

Raises

- **DiagnosticsException** – If there is error in diagnostics.
- **InvalidStateException** – If device is in invalid state (that is not OFF)
- **DeviceRuntimeException** – If any exception occours.

pop_results()
This method returns list of recently acquired points, it should clear this list so next calls won't return the same result points.

Raises

- **InvalidStateException** – If device is in invalid state (that is not RUNNING
- **InvalidStateException** – If device is in invalid state (that is not READY
- **DeviceRuntimeException** – If any exception occours

Returns Returns results for (possibly) many points.

Return type list (or any other iterable) of *dict*.

post_power_up_diagnostics(*diagnostics_level='short'*)

power_down()
Call to this method moves this class to OFF state.

Raises

- **InvalidStateException** – If device is in invalid state (that is not STAND_BY
- **DeviceRuntimeException** – If any exception occours

power_up()
Call to this method enables consecutive *apply_settings()*.

It also should power up the device (if this action makes any sense for this particular device see also: *Power management*).

Raises

- **InvalidStateException** – If device is in invalid state (that is not OFF
- **DeviceRuntimeException** – If any exception occours

pre_power_up_diagnostics(*diagnostics_level='short'*)

start()

Starts the acquisituon on the device (that is starts the measurements).

Blocks until this device is stared.

Raises **InvalidStateException** – If device is in invalid state (that is not READY

Returns None

Return type None

state = None

State of this device should be in DEVICE_STATES, full state chart is avilable in: *Device state chart*.

stop()

Stops the acquisituon on the device (that is stops the measurements).

Blocks until this device is stared.

Raises

- **InvalidStateException** – If device is in invalid state (that is not RUNNING)
- **DeviceRuntimeException** – If any exception occours

Returns None

Return type None

tearDown()

tear_down()

Called when current process is being disabled.

This method can be called multiple times.

Note: do not override this method, override `_tear_down()`.

Raises **DeviceRuntimeException** – If any exception occours

Returns None

Return type None

exception silf.backend.commons.device._device.**InvalidCallToAssertState**

Bases: Warning

CHAPTER 4

API fine print

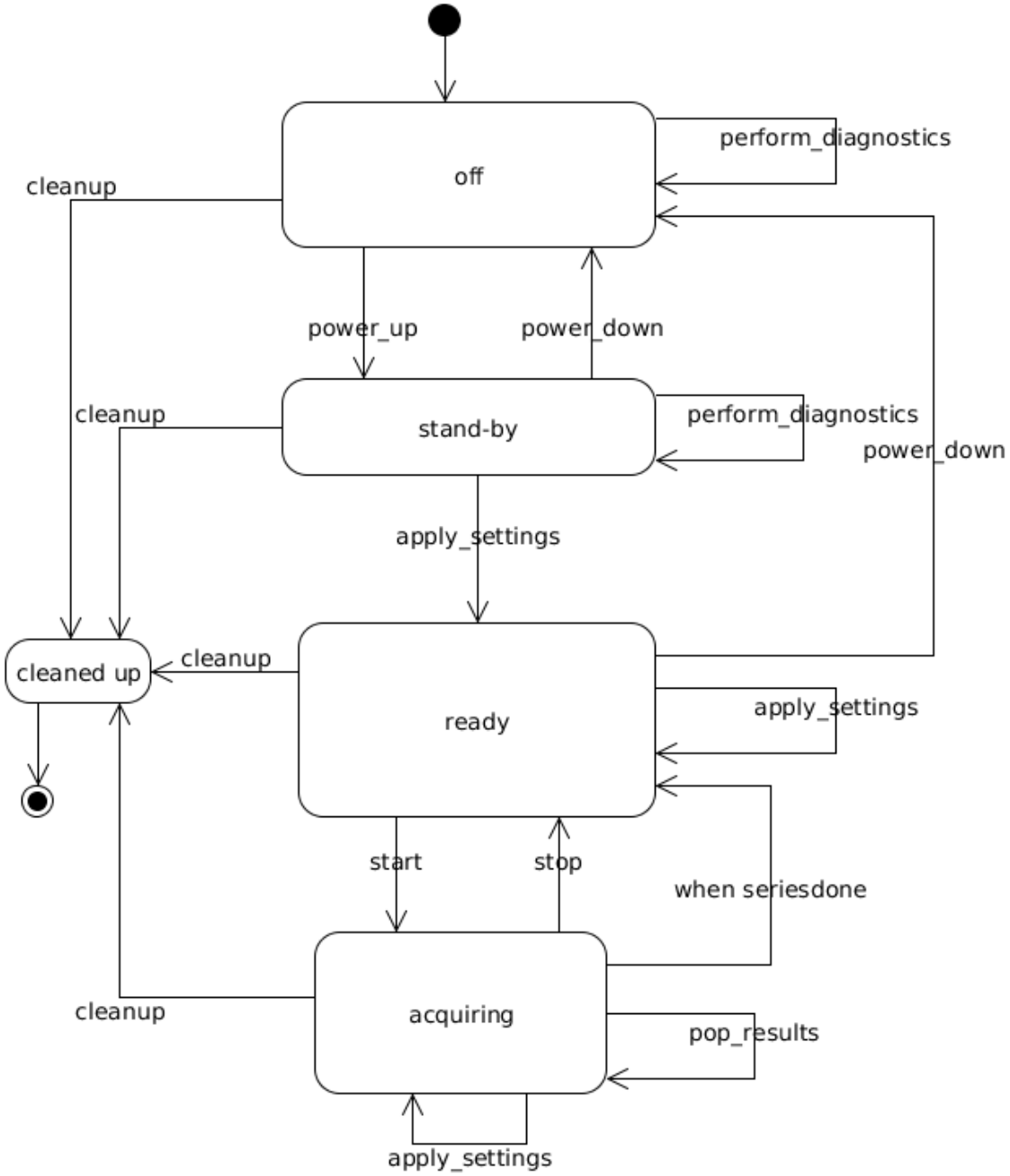

Fig. 4.1: Device state chart

CHAPTER 5

Power management

Note: If your device does not need to power itself up or down, please just ignore *power_up()* and *power_down()* methods.

Devices should be powered up when we start call *power_up()*, but needn't do so, they must be powered up when after we exit from `start()`. So there are three methods in which devices should power up:

- *power_up()*, this method is called relatively early in during the experiment, and should allow plenty of time to initialize everyhing
- `apply_settings()`, use this method if your device powers up quickly.
- `start()`, if your device is volatile and you want to minimize the time it is powered up use this.

You can power down the device when following methods are called:

- *power_down()*
- *stop()*

CHAPTER 6

Threading considerations

Devices are accessed from single thread. All methods sould exit relatively fast, **you should not use loops that are infinite** (or **can be infinite** — for example if hardware will not respond).

CHAPTER 7

Change device state

It is quite important to change state of your device after appropriate method calls.

CHAPTER 8

How to test the devices according to the API

There are two ways in which you can test it: start ipython interpreter create device and manage it by hand:

8.1 Use `DeviceWorkerWrapper` from interpreter

Import classes:

```
>>> from silf.backend.commons_test.device.test_device import *
>>> from silf.backend.commons.device_manager import start_worker_interactive
```

Start the device:

```
>>> work = start_worker_interactive('foo', MockDevice,
... configure_logging=False, auto_pull_results=False)

>>> work.state
'off'

>>> work.power_up()
UUID(...)
```

Let's setup the device:

```
>>> work.apply_settings({"foo": 3, "bar": 2})
UUID(...)

>>> work.start()
UUID(...)
```

This device will perform own acquisition in separete process, well wait for results to be acquired:

```
>>> time.sleep(1.2)
```

First `pop_results()` will return stale data, and schedule acquisition of new data:

```
>>> work.state
'running'
>>> work.pop_results() == []
True
```

Wait for results to get processed (will be faster on server!)

```
>>> time.sleep(0.5)
>>> results = work.pop_results()
>>> results == [{'foo_result': 3, 'bar_result': 2}]
True
```

Kill it without waiting;

```
>>> work.kill(wait_time=None)
```

8.2 Auto result pooling

You can configure this to auto poll for results:

```
>>> work = start_worker_interactive('foo', MockDevice,
... configure_logging=False, auto_pull_results=True)


>>> work.power_up()
UUID(...)
```

As in last test:

```
>>> work.apply_settings({"foo": 3, "bar": 2})
UUID(...)
>>> work.start()
UUID(...)
```

Wait for results to be gathered

```
>>> time.sleep(2)
```

Notice that results are avilable at once (no need to query)

```
>>> results = work.pop_results()
>>> results == [{'foo_result': 3, 'bar_result': 2}]
True

>>> work.kill(wait_time=None)
>>> results == [{'foo_result': 3, 'bar_result': 2}]
True

>>> work.kill(wait_time=None)
```

CHAPTER 9

Device Api examples

This is pseudocode

9.1 Engine driver

This imaginary device implements an engine. This is not actual experiment code, sxperiment will not be doing any waiting!

```
engine = ImaginaryDriver()

assert engine.state == 'off'

engine.power_up() # Powers up the device

assert engine.state == 'stand-by'

engine.apply_settings({"position" : 512})

assert engine.state == 'ready'

engine.start() # Start the engine

assert engine.state == 'acquiring'

# Silnik ruszył i teraz jest w stanie `acquiring`

# .. wait

while engine.state != 'ready':
    time.sleep(0.1)

# Silnik doszedł do końca i jest w stanie `ready`

# Następny pukt
```

```
engine.apply_settings({"position" : 1024})

engine.start() # Start the engine
```

9.2 Engine driver

Imaginary voltimeter

```
volt = ImaginaryVoltimeter()

assert volt.state == 'off'

volt.power_up() # Powers up the device

assert volt.state == 'stand-by'

volt.apply_settings({'range' : 15})

assert volt.state == 'ready'

volt.start() # Start the volt

assert volt.state == 'acquiring'

while volt.state != 'ready':
    time.sleep(0.1)

assert volt.pop_results() == [{'voltage' : 243.11}]
```

9.3 Engine and voltimeter connected

It works that so voltimeter measures single point after position is set by the engine.

```
engine = ImaginaryDriver()
volt = ImaginaryVoltimeter()

engine.power_up() # Powers up the device
volt.power_up() # Powers up the device

engine.apply_settings({"position" : 512})

engine.start();

while engine.state != 'ready':
    time.sleep(0.1)

volt.apply_settings({'range' : 15})

volt.start() # Start the volt

while volt.state != 'ready':
    time.sleep(0.1)
```

```
assert volt.pop_results() == [{'voltage' : 243.11}]

engine.apply_settings({"position" : 1024})

engine.start();

while engine.state != 'ready':
    time.sleep(0.1)

while volt.state != 'ready':
    time.sleep(0.1)

assert volt.pop_results() == [{'voltage' : 123.123}]
```

Python Module Index

S

Index