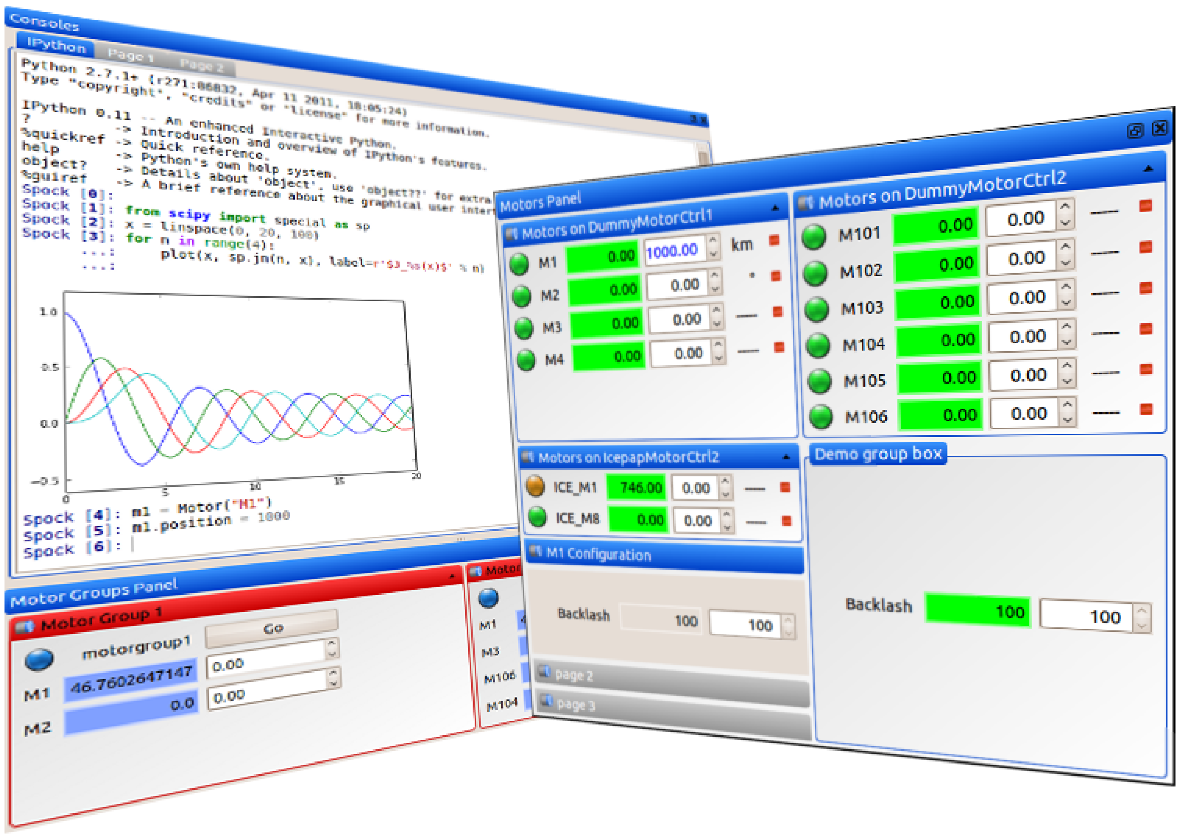


```
Python 2.7.1+ (r271:86832, Apr 11 2011, 18:05:24)
Type "copyright", "credits" or "license" for more information.

IPython 0.11 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra
%guiref   -> A brief reference about the graphical user inter
Spock [0]:
Spock [1]: from scipy import special as sp
Spock [2]: x = linspace(0, 20, 100)
Spock [3]: for n in range(4):
      ...:     plot(x, sp.jn(n, x), label=r'$J_%s(x)$' % n)
      ...:
Spock [4]: m1 = Motor("M1")
Spock [5]: m1.position = 1000
Spock [6]:
```

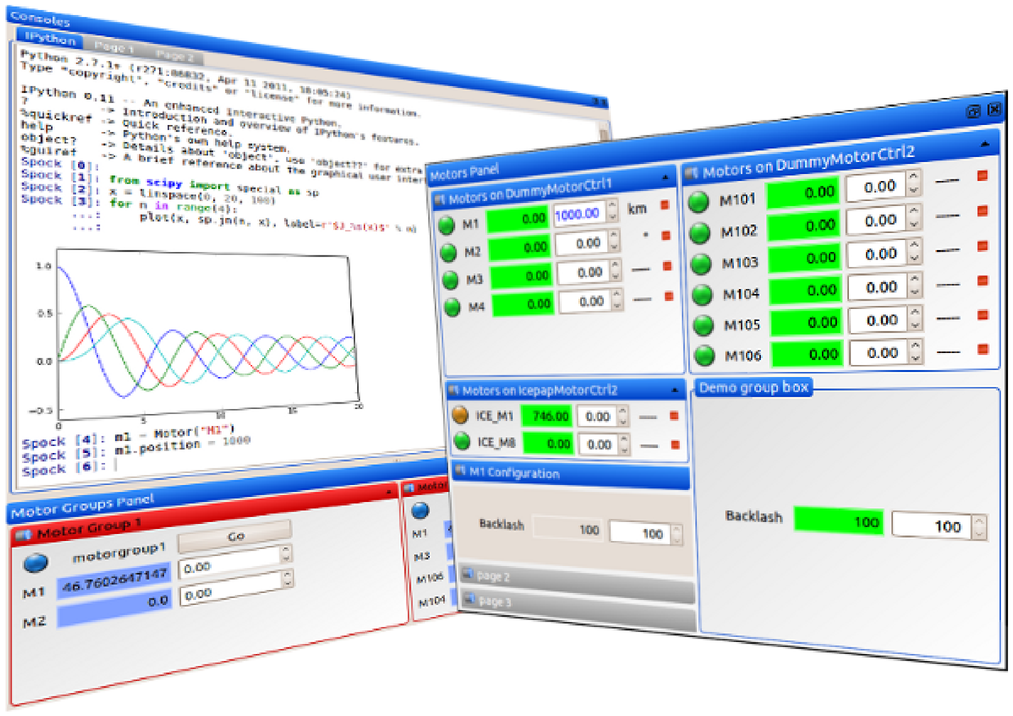
Sardana Documentation

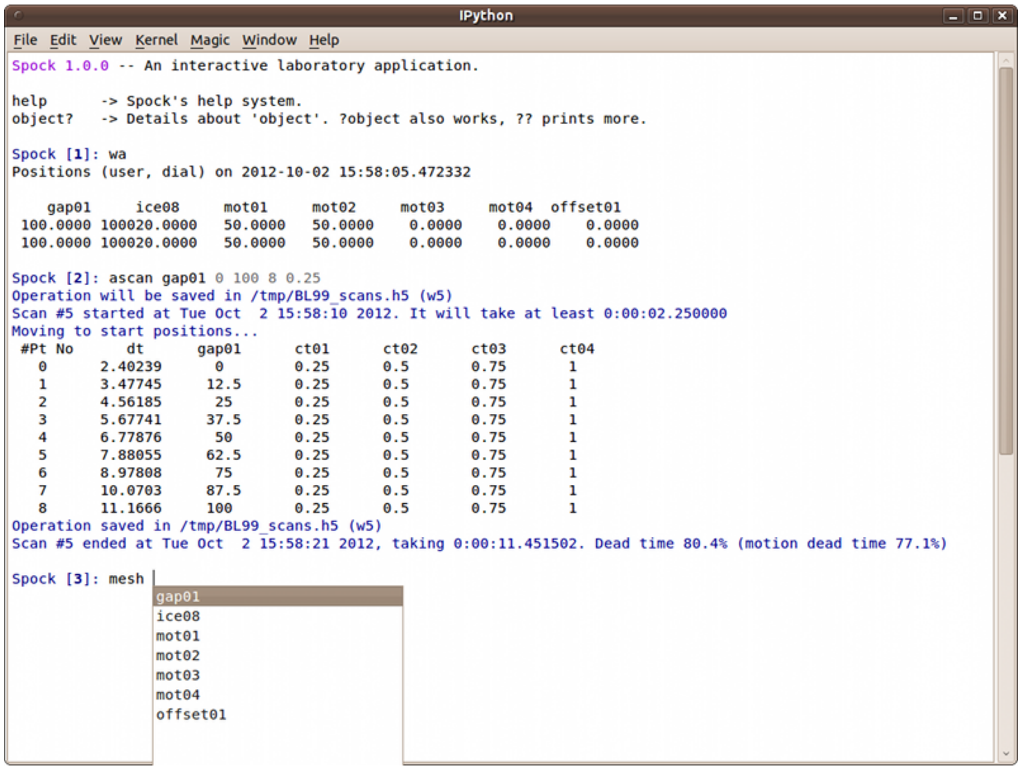
Release 2.4.1-alpha

Sardana team

Jul 11, 2018

Contents





```
Spock 1.0.0 -- An interactive laboratory application.

help      -> Spock's help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

Spock [1]: wa
Positions (user, dial) on 2012-10-02 15:58:05.472332

    gap01     ice08      mot01     mot02     mot03     mot04  offset01
 100.0000 100020.0000   50.0000   50.0000    0.0000    0.0000    0.0000
 100.0000 100020.0000   50.0000   50.0000    0.0000    0.0000    0.0000

Spock [2]: ascan gap01 0 100 8 0.25
Operation will be saved in /tmp/BL99_scans.h5 (w5)
Scan #5 started at Tue Oct  2 15:58:10 2012. It will take at least 0:00:02.250000
Moving to start positions...
 #Pt No       dt       gap01      ct01      ct02      ct03      ct04
   0       2.40239       0        0.25      0.5       0.75       1
   1       3.47745     12.5       0.25      0.5       0.75       1
   2       4.56185      25        0.25      0.5       0.75       1
   3       5.67741     37.5       0.25      0.5       0.75       1
   4       6.77876      50        0.25      0.5       0.75       1
   5       7.88055     62.5       0.25      0.5       0.75       1
   6       8.97808      75        0.25      0.5       0.75       1
   7       10.0703     87.5       0.25      0.5       0.75       1
   8       11.1666     100        0.25      0.5       0.75       1
Operation saved in /tmp/BL99_scans.h5 (w5)
Scan #5 ended at Tue Oct  2 15:58:21 2012, taking 0:00:11.451502. Dead time 80.4% (motion dead time 77.1%)

Spock [3]: mesh
                gap01
                ice08
                mot01
                mot02
                mot03
                mot04
                offset01
```

Sardana is a software suite for Supervision, Control and Data Acquisition in scientific installations. It aims to reduce cost and time of design, development and support of the control and data acquisition systems. Sardana development was started at the ALBA[4] synchrotron and today is supported by a larger community

[4] http://www.albasynchrotron.es

which includes several other laboratories and individuals (ALBA[5], DESY[6], MaxIV[7], Solaris[8], ESRF[9]).

You can download Sardana from PyPi[10], check its Documentation[11] or get support from its community and the latest code from the project page[12].

5 http://www.albasynchrotron.es
6 http://www.desy.de
7 http://www.maxiv.se/
8 http://www.synchrotron.uj.edu.pl/en_GB/
9 http://esrf.eu
10 http://pypi.python.org/pypi/sardana
11 http://sardana.readthedocs.org
12 https://github.com/sardana-org/sardana

CHAPTER 1

Projects related to Sardana

- Sardana uses Taurus[13] for control system access and user interfaces
- Sardana is based on Tango[14]
- The command line interface for Sardana (Spock) is based on IPython[15]

1.1 Sardana 2.4 Documentation

Sardana is a software suite for Supervision, Control and Data Acquisition in scientific installations.

1.1.1 User's Guide

Overview

Sardana is the control program initially developed at ALBA[16]. Our mission statement:

> *Produce a modular, high performance, robust, and generic user environment for control applications in large and small installations. Make Sardana the generic user environment distributed in the Tango project and the standard basis of collaborations in control.*

Up to now, control applications in large installations have been notoriously difficult to share. Inspired by the success of the Tango[17] collaboration, ALBA[18] decided to start the creation of a generic tool to enlarge the scope of the Tango[19] project to include a standard client program - or better a standard generic user environment. From the beginning our aim has been to involve others in this process. At this moment in time the user environment consists of a highly configurable standard graphical user interface, a standard command

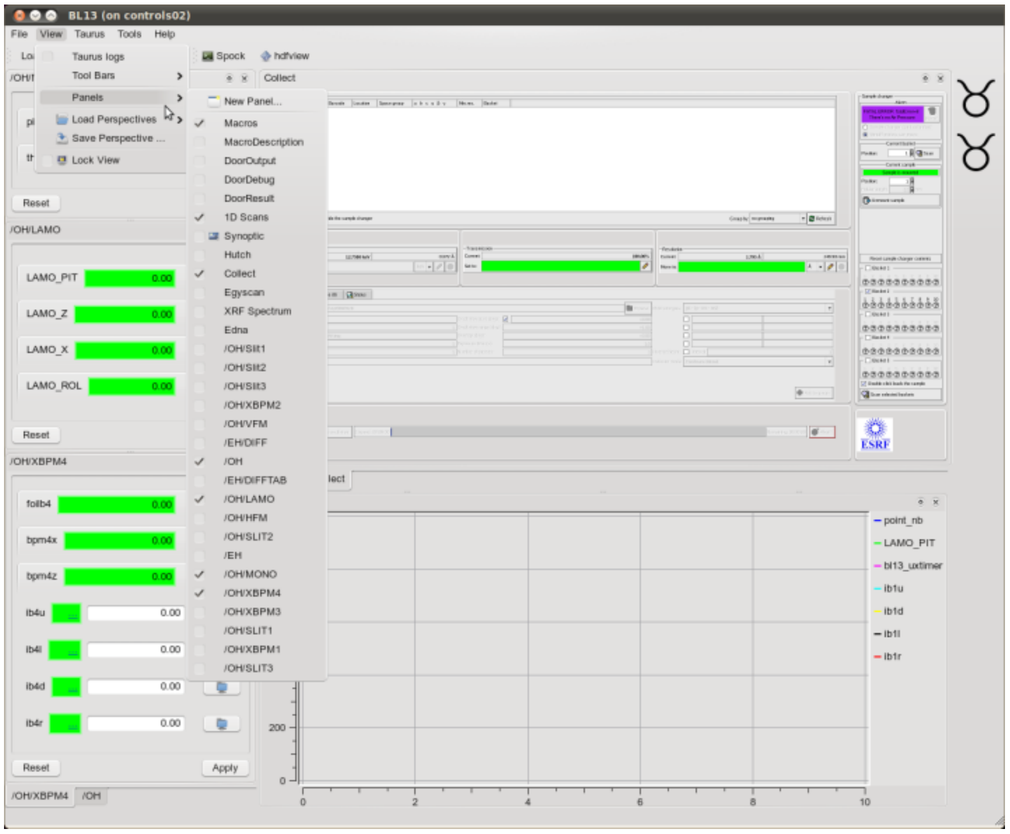
[13] http://taurus-scada.org/
[14] http://www.tango-controls.org/
[15] http://ipython.org/
[16] http://www.cells.es/
[17] http://www.tango-controls.org/
[18] http://www.cells.es/
[19] http://www.tango-controls.org/

line interface understanding SPEC[20] commands, and a standard way to compose new applications either by programming or with a graphical tool. It further consists of a standard macro executer, standard set of macros, a standard range of common hardware types (like motors, counters, cameras and so on) and a configuration editor to set all this up. The origin of the Sardana name comes from a Catalan dance to honor the region where the ALBA[21] synchrotron is build. The toolkit to build Sardana has been C++, Python[22], Qt[23] and Tango[24]. If you like the tools you will love Sardana.

What do we "sell" to our users

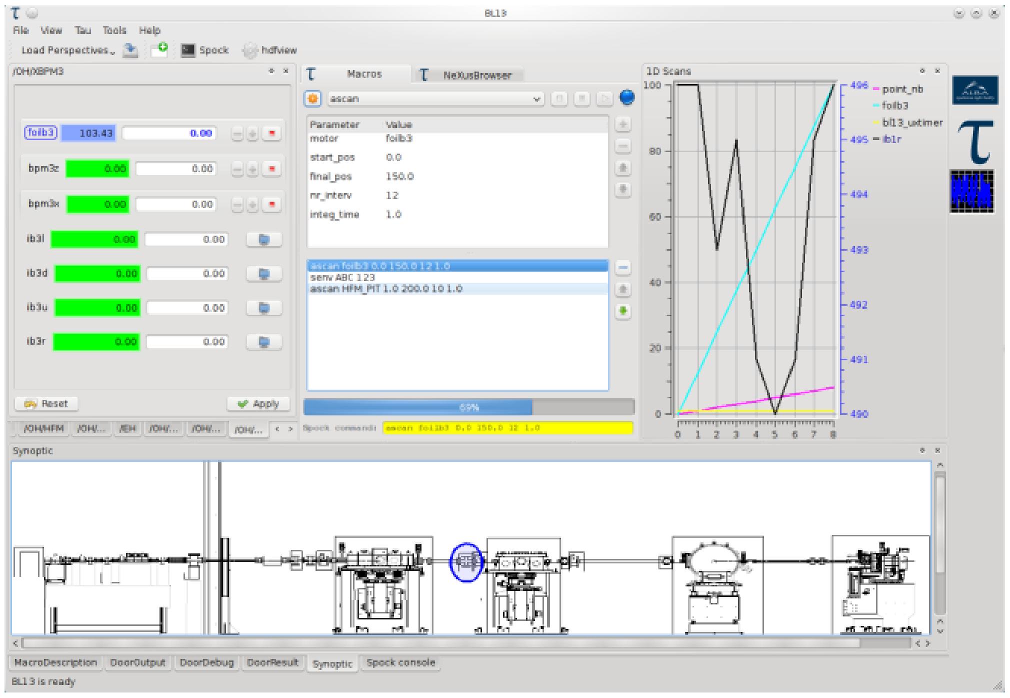
Let's start our excursion into the Sardana world by a word of caution. We will talk a lot about technical possibilities and implementation details. Our users will judge us on the ease of use of the final GUI, its robustness and the features it offers. There are millions of ways to arrive at this end result. Our claim is however that by doing it the *Sardana way* and developing the application out of *lego* components in a collaborative environment we will arrive at higher quality software with much higher efficiency.

The following screen shot of an early prototype of a specific beamline application should serve as a reminder of this final goal.



Inside this application we have many features common to other beamline control applications or w some accelerator applications. The following screen shot shows such a standard application which has been done without programming - just by configuring the application. This illustrates one of the design guidelines in Sardana: Always provide a generic interface which can be specialized for an application.

[20] http://www.certif.com/
[21] http://www.cells.es/
[22] http://www.python.org/
[23] http://qt.nokia.com/products/
[24] http://www.tango-controls.org/



foilb3	103.43	0.00
bpm3z	0.00	0.00
bpm3x	0.00	0.00
ib3l	0.00	0.00
ib3d	0.00	0.00
ib3u	0.00	0.00
ib3r	0.00	0.00

ascan

Parameter	Value
motor	foilb3
start_pos	0.0
final_pos	150.0
nr_interv	12
integ_time	1.0

```
ascan foilb3 0.0 150.0 12 1.0
senv ABC 123
ascan HFM_PIT 1.0 200.0 10 1.0
```

69%

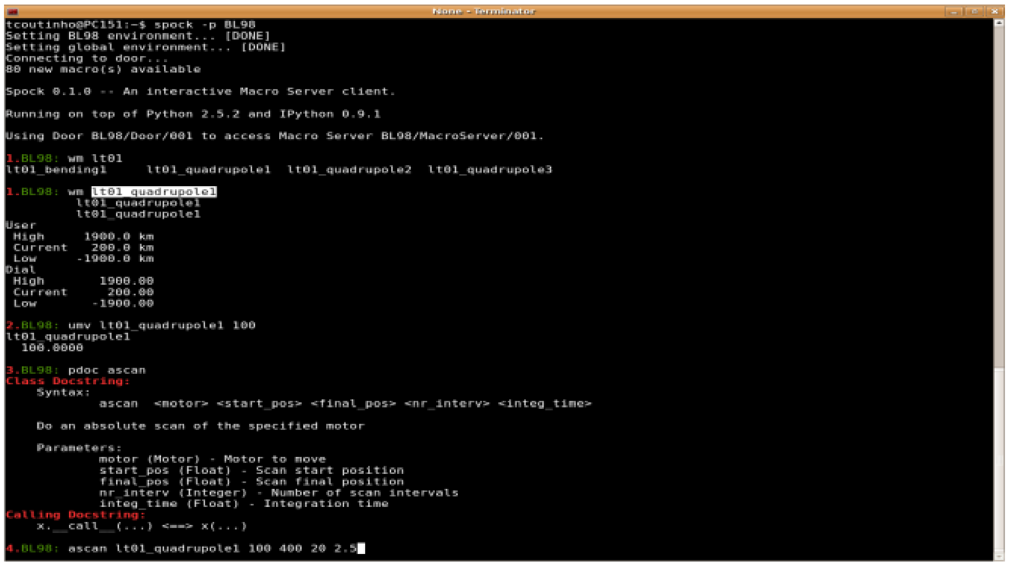
Spock command: `ascan foilb3 0.0 150.0 12 1.0`

BL13 is ready

Starting a procedure

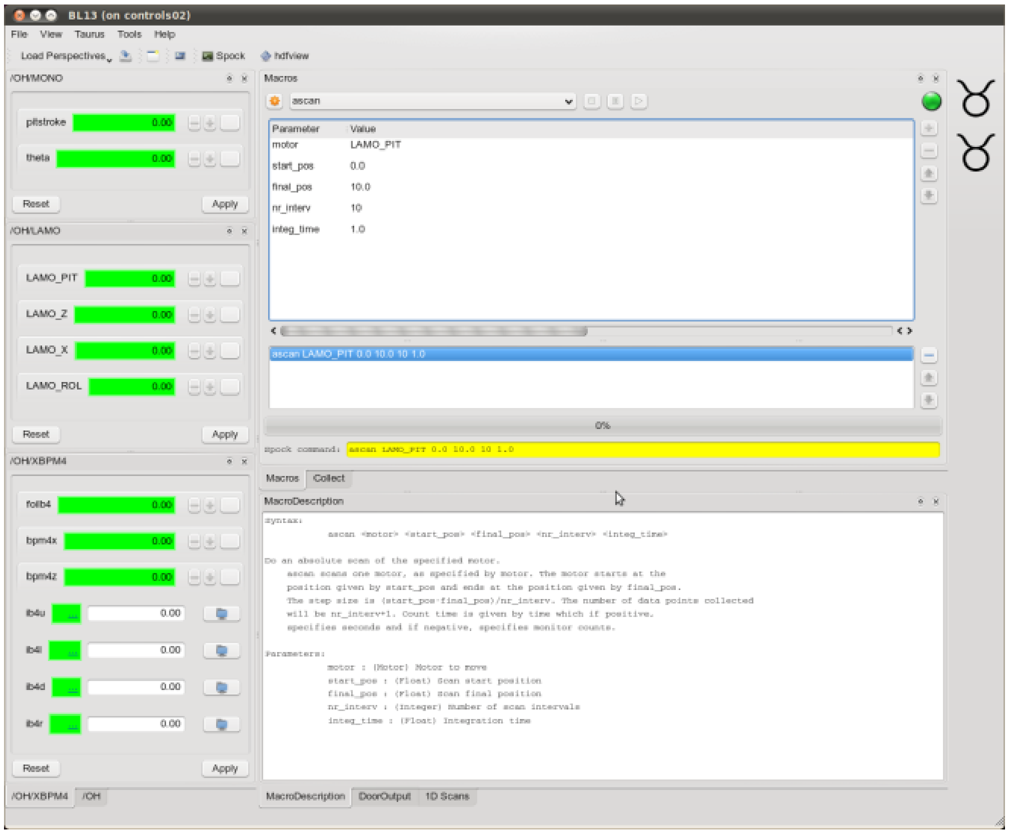
At the heart of the Sardana system are standard reusable procedures. From past experiences, the importance of standard procedures has been realized and has influenced most of the major design decisions. To illustrate this point, please let me walk you through different ways how to start such a procedure without going into too many details. You might want to think of a *scan* as an example. One way of starting a procedure is with a command line interface. Users familiar with SPEC[25] will immediately recognize this way. In effect, inside Sardana most of the standard SPEC[26] commands (including many diffractometer geometries thanks to Frédéric Picca from the SOLEIL[27] synchrotron) are provided as standard procedures and can be invoked in the same way.

[25] http://www.certif.com/
[26] http://www.certif.com/
[27] http://www.synchrotron-soleil.fr/



```
tcoutinho@PC151:~$ spock -p BL98
Setting BL98 environment... [DONE]
Setting global environment... [DONE]
Connecting to door...
80 new macro(s) available

Spock 0.1.0 -- An interactive Macro Server client.

Running on top of Python 2.5.2 and IPython 0.9.1

Using Door BL98/Door/001 to access Macro Server BL98/MacroServer/001.

1.BL98: wm lt01
lt01_bending1     lt01_quadrupole1  lt01_quadrupole2  lt01_quadrupole3

1.BL98: wm lt01_quadrupole1
         lt01_quadrupole1
         lt01_quadrupole1
User
 High     1900.0 km
 Current   200.0 km
 Low     -1900.0 km
Dial
 High       1900.00
 Current     200.00
 Low       -1900.00

2.BL98: umv lt01_quadrupole1 100
lt01_quadrupole1
  100.0000

3.BL98: pdoc ascan
Class Docstring:
    Syntax:
            ascan  <motor> <start_pos> <final_pos> <nr_interv> <integ_time>

    Do an absolute scan of the specified motor

    Parameters:
            motor (Motor) - Motor to move
            start_pos (Float) - Scan start position
            final_pos (Float) - Scan final position
            nr_interv (Integer) - Number of scan intervals
            integ_time (Float) - Integration time
Calling Docstring:
    x.__call__(...) <==> x(...)

4.BL98: ascan lt01_quadrupole1 100 400 20 2.5
```

Every procedure can also be started from a GUI. This does not need any programming or configuration from the user of the system. When a new procedure is created, it is automatically visible inside the GUI and command line tools.



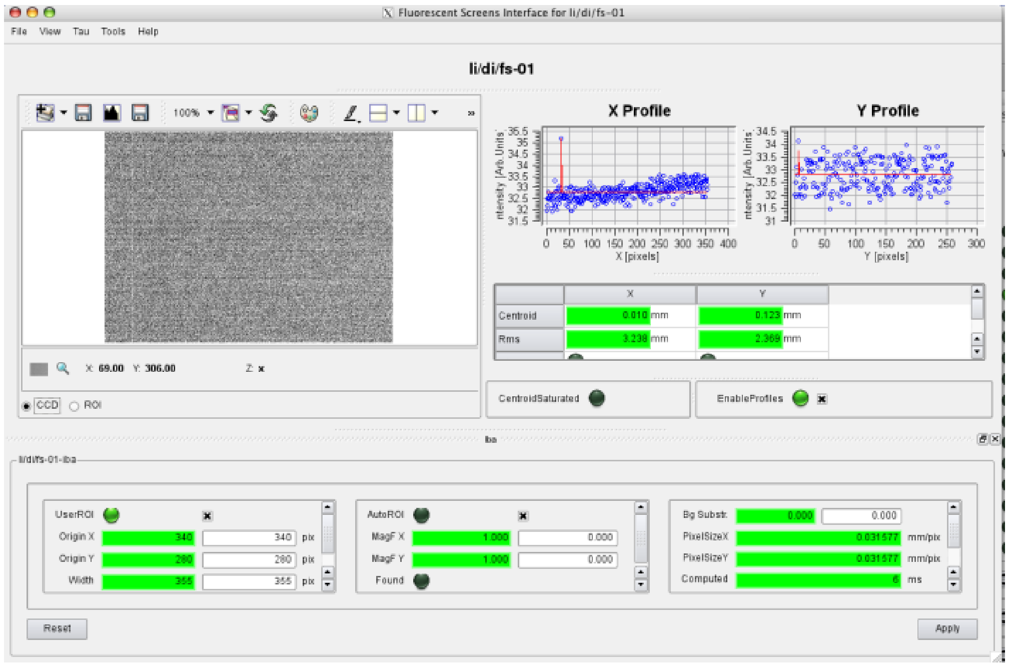
Parameter	Value
motor	LAMO_PIT
start_pos	0.0
final_pos	10.0
nr_interv	10
integ_time	1.0

```
ascan LAMO_PIT 0.0 10.0 10 1.0
```

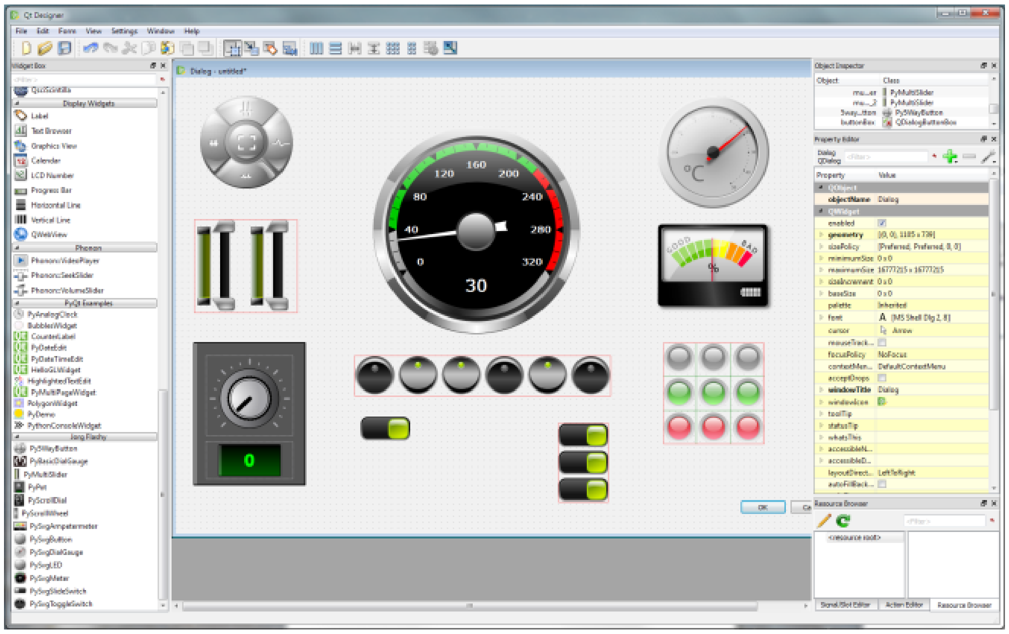
```
spock command: ascan LAMO_PIT 0.0 10.0 10 1.0
```

```
Syntax:
          ascan <motor> <start_pos> <final_pos> <nr_interv> <integ_time>

Do an absolute scan of the specified motor.
    ascan scans one motor, as specified by motor. The motor starts at the
    position given by start_pos and ends at the position given by final_pos.
    The step size is (start_pos-final_pos)/nr_interv. The number of data points collected
    will be nr_interv+1. Count time is given by time which if positive,
    specifies seconds and if negative, specifies monitor counts.

Parameters:
          motor : (Motor) Motor to move
          start_pos : (Float) Scan start position
          final_pos : (Float) Scan final position
          nr_interv : (Integer) Number of scan intervals
          integ_time : (Float) Integration time
```

This GUI interface will mainly be used for procedures which are rarely used and where a specialized interface has not yet been developed. An example of how to use the same procedure in order to carry out energy spread and emittance measurements is presented in the following picture.



The standard Qt[28] designer can be used to create new graphical elements (widgets) and connect them to the system for even greater flexibility. The following screen shot shows the standard qt designer with some fancy widgets developed in house.



Taurus as a toolkit for applications

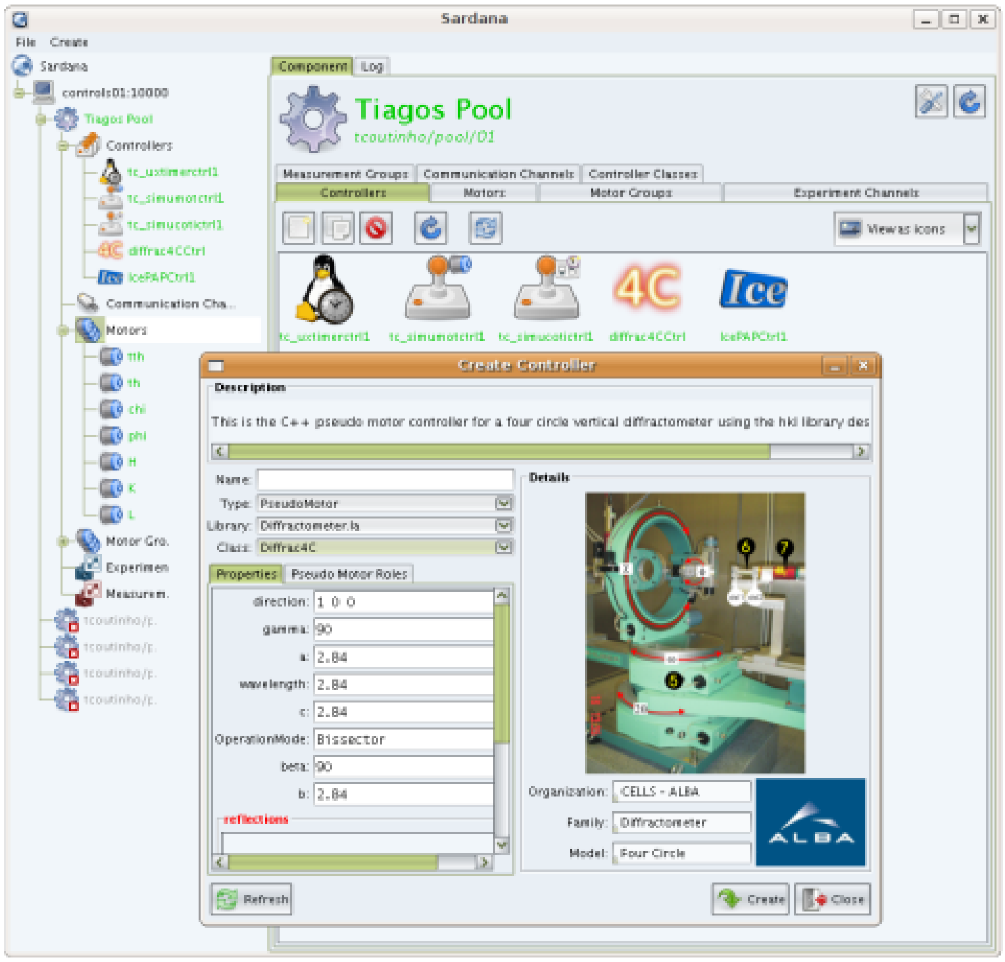
The GUI toolkit for Sardana is called Taurus[29]. The graphical user interfaces in this paper have been created with this toolkit. It can be used in conjunction or independent from the rest of the system. It can be used to create custom panels inside the generic GUI or to create stand alone applications. Again, this approach of

[28] http://qt.nokia.com/products/
[29] http://packages.python.org/taurus/

take what you need has been implemented to foster the widest range of collaborations. Almost all applications in the ALBA[30] machine control system have been created with this toolkit. Creating the applications out of standard components has been proven to be extremely powerful. In the *Graphical user interface screen shots* chapter you can see some of the graphical user interfaces used.

Configure – don't program

The Sardana system comes with a configuration editor to allow non-experts to add and configure components. The editor adapts dynamically to the hardware controllers present. New hardware controller can be easily written and integrated into the system without restarting it.



This is the C++ pseudo motor controller for a four circle vertical diffractometer using the hkl library des

Name:
Type: PseudoMotor
Library: Diffractometer.la
Class: Diffrac4C

direction: 1 0 0
gamma: 90
a: 2.84
wavelength: 2.84
c: 2.84
OperationMode: Bissector
beta: 90
b: 2.84

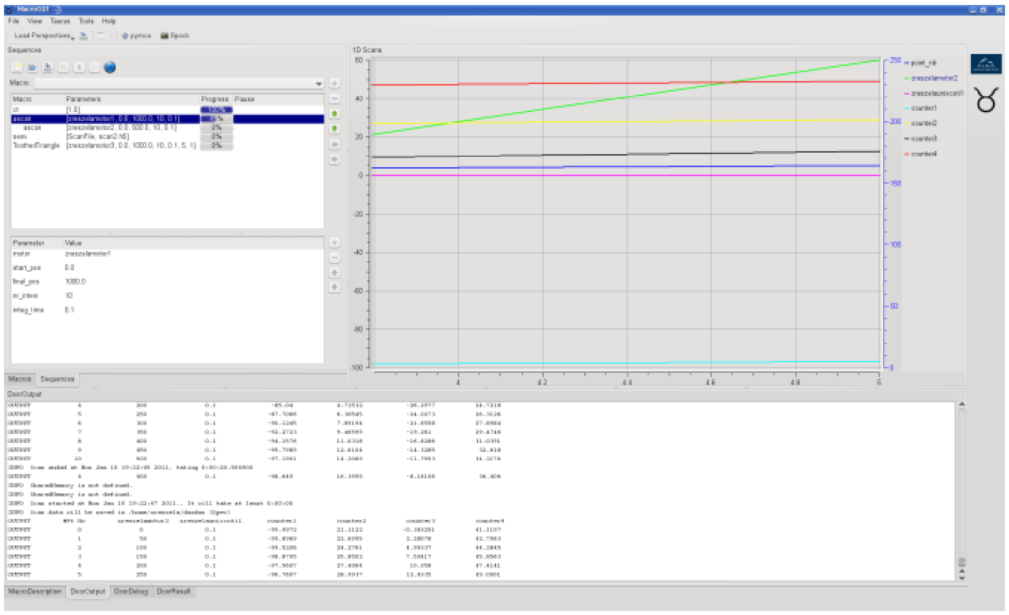
Organization: CELLS - ALBA
Family: Diffractometer
Model: Four Circle

This configuration editor is currently being rewritten to be more wizard based and provide better guidance for the end user.

How to write your own procedure

Another example I would like to look into is how to write your own procedure. The simplest possible way is to use an editor to assemble commands and execute them. This batch files type of procedures are useful to automatically run procedures over night and for similar simple applications. The following screen shots show the procedure executer with this feature enabled.

[30] http://www.cells.es/



To go further I would like to explain some internal details. All procedures are executed in a central place (called the macro server). There can be more than one macro server per system but for the following I assume the common case of a unique macro server. This macro server holds all the general procedures centrally. It provides a controlled environment for these procedures. They can be edited, run, debugged under its supervision. This allows for example to automatically roll back changes made in case of problems, log access and grant permissions. The procedures executed in the macro server provided by the current Sardana system are Python[31] functions or classes. Writing a procedure as a function is more straightforward and recommended for the beginners. Writing it is a class is a way to group the different methods which concerns this procedure. As an example, in some procedures it could be possible to do very specific things in case the user orders an emergency abort of the procedure. The following example shows the procedure to move a motor.

```python
from sardana.macroserver.macro import macro, Type

@macro([ ["moveable", Type.Moveable, None, "moveable to move"],
         ["position", Type.Float, None, "absolute position"] ])
def move(self, moveable, position):
    """This macro moves a moveable to the specified position"""
    moveable.move(position)
    self.output("%s is now at %s", moveable.getName(), moveable.getPosition())
```

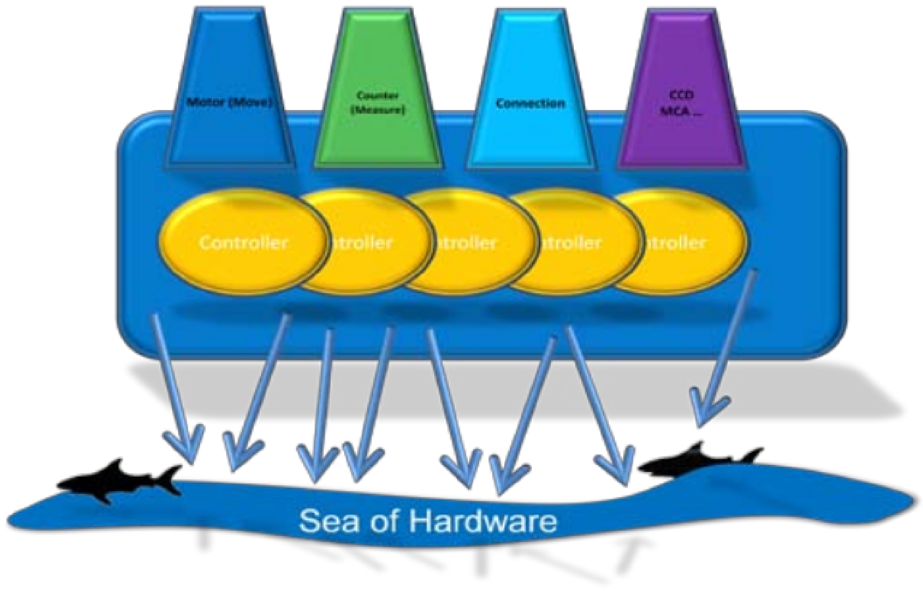
As you can see in the example, the procedure must be documented and the input parameters described. From this information, the graphical user interface is constructed. It is also possible now to start the procedure from the command line interface and use the tab key to automatically complete the input. The actual action is actually carried out in the run method. The motor movement is started and the procedure waits until it arrives at its destiny. The Python[32] classes should stay small and very simple. All complicated code can be put into modules and tested separately from the system.

How to adapt it to your own hardware

As the system has been thought from the beginning to be used at different institutes, no assumptions of the hardware used could be made. There exists therefore a mechanism to adapt the Sardana system to

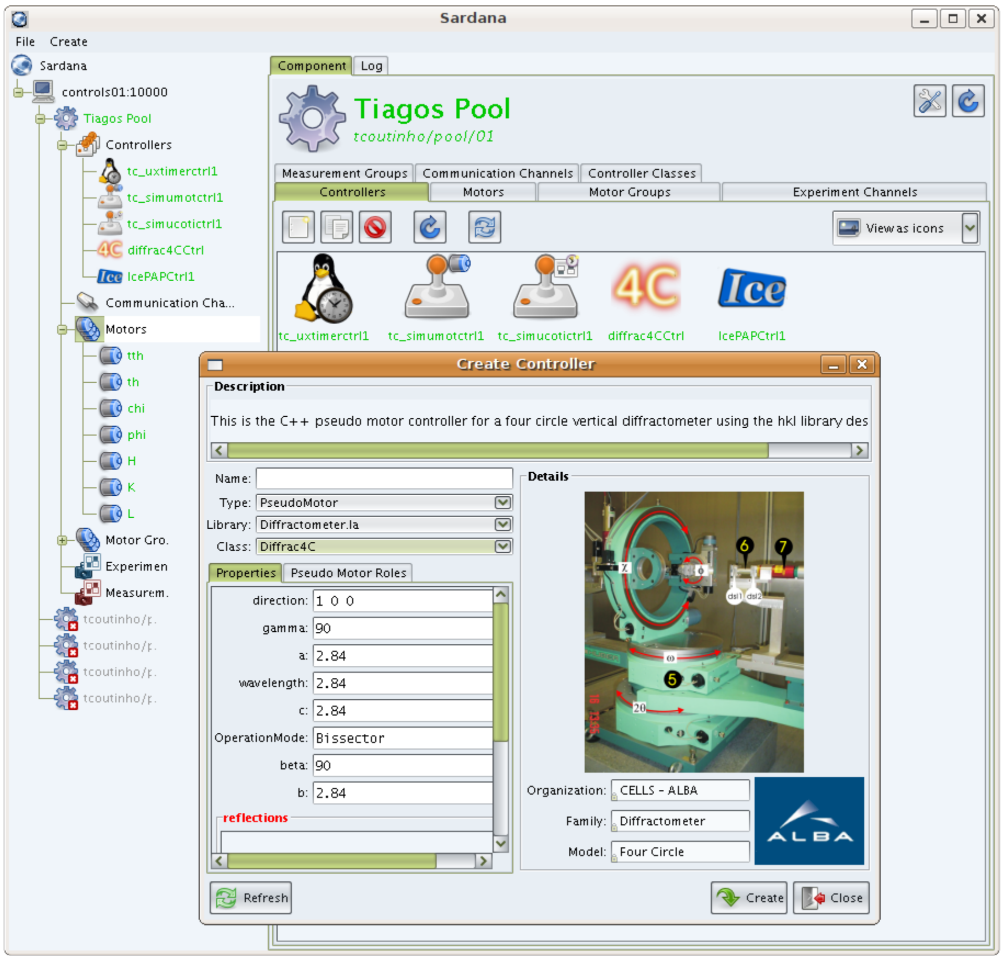
[31] http://www.python.org/
[32] http://www.python.org/

your own hardware. This adaptor also has another very important role to play. This is best explained with the motor as example. We consider more or less everything which can be changed in the system a motor. The term which should have better been used to describe this thing should have been therefore *movable*. A motor can be a temperature of a temperature controller which can be changed, a motor from an insertion device which needs a highly complicated protocol to be moved, or just about anything. Sometimes we also consider calculated value like H,K,L, the height of a table, and the gap of a slit to be a motor. All these different *motors* can be scanned with the same generic procedures without having to worry about on which elements it is working on. You can add one of these pseudo motors with the configuration editor. It is easily possible to add new types of pseudo motors. This has only to be done once and the Sardana system already provides a large variety of these types.



Please find in the following an example for adding a completely new type in the case of a *slit*.

The actual information how to create a motor of type *slit* is kept in the two methods calc_physical and calc_pseudo which can be used to do the transformation between the different coordinate systems. Or to say it in the language of Sardana between the pseudo motors gap and offset and the real motors left blade and right blade.



Description

This is the C++ pseudo motor controller for a four circle vertical diffractometer using the hkl library des

Name:
Type: PseudoMotor
Library: Diffractometer.la
Class: Diffrac4C

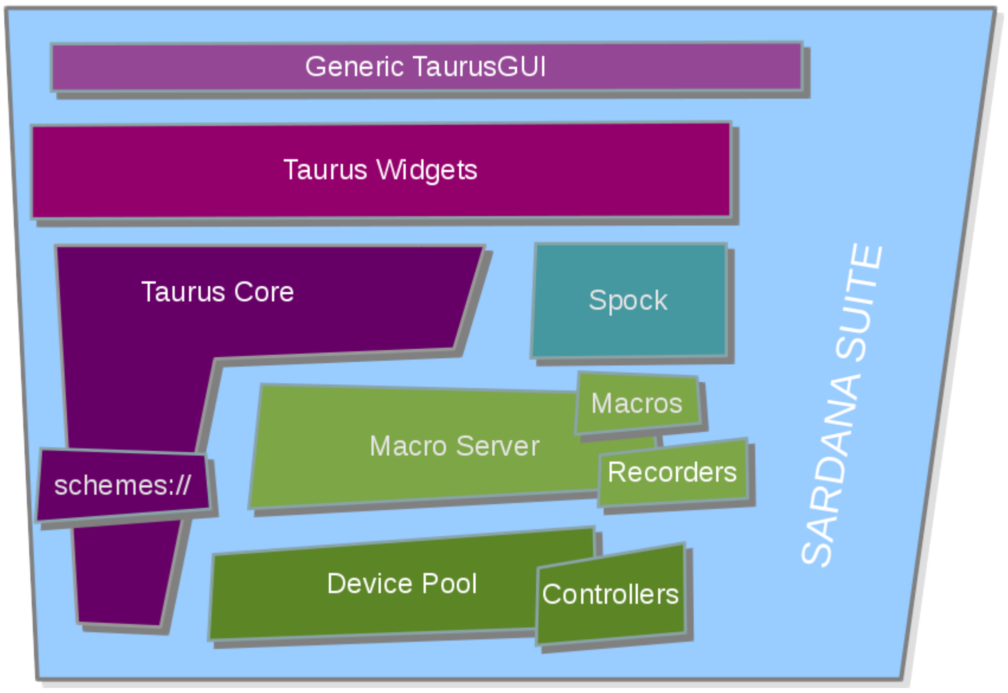
direction: 1 0 0
gamma: 90
a: 2.84
wavelength: 2.84
c: 2.84
OperationMode: Bissector
beta: 90
b: 2.84

Organization: CELLS - ALBA
Family: Diffractometer
Model: Four Circle

Once again the information in the beginning allows the graphical user interface to be created automatically once it is loaded into the system.

Symbolic Sketch

I would like to end this summary with a symbolic sketch of the different subsystems in Sardana.



The user will normally not be concerned with these implementation details. It is presented here to allow appreciating the modularity of the system.

Getting started

The next chapters describe the necessary steps to get started with sardana, from installation to having a running system on your machine.

Installing

Installing with pip[1] (platform-independent)

Sardana can be installed using pip. The following command will automatically download and install the latest release of Sardana (see pip –help for options):

```
pip install sardana
```

You can test the installation by running:

```
python -c "import sardana; print sardana.Release.version"
```

Installing from PyPI manually[2] (platform-independent)

[1] This command requires super user previledges on linux systems. If your user has them you can usually prefix the command with *sudo*: `sudo pip -U sardana`. Alternatively, if you don't have administrator previledges, you can install locally in your user directory with: `pip --user sardana` In this case the executables are located at <HOME_DIR>/.local/bin. Make sure the PATH is pointing there or you execute from there.

[2] *setup.py install* requires user previledges on linux systems. If your user has them you can usually prefix the command with *sudo*: `sudo python setup.py install`. Alternatively, if you don't have administrator previledges, you can install locally in your user directory with: `python setup.py install --user` In this case the executables are located at <HOME_DIR>/.local/bin. Make

You may alternatively install from a downloaded release package:

1. Download the latest release of Sardana from http://pypi.python.org/pypi/sardana
2. Extract the downloaded source into a temporary directory and change to it
3. run:

   ```
   python setup.py install
   ```

You can test the installation by running:

```
python -c "import sardana; print sardana.Release.version"
```

Linux (Debian-based)

Since v1.4, Sardana is part of the official repositories of Debian (and Ubuntu and other Debian-based distros). You can install it and all its dependencies by doing (as root):

```
aptitude install python-sardana
```

You can test the installation by running:

```
python -c "import sardana; print sardana.Release.version"
```

(see more detailed instructions in this step-by-step howto[33])

Windows

1. Download the latest windows binary from http://pypi.python.org/pypi/sardana
2. Run the installation executable
3. test the installation:

   ```
   C:\Python27\python -c "import sardana; print sardana.Release.version"
   ```

Windows installation shortcut

This chapter provides a quick shortcut to all windows packages which are necessary to run Sardana on your windows machine

1. Install all dependencies:
   (a) Download and install latest PyTango[34] from PyTango downdoad page[35]
   (b) Download and install latest Taurus[36] from Taurus downdoad page[37]
   (c) Download and install latest lxml[38] from lxml downdoad page[39]

sure the PATH is pointing there or you execute from there.

[33] https://sourceforge.net/p/sardana/wiki/Howto-Sardana-on-Debian8/
[34] http://pytango.readthedocs.io/
[35] http://pypi.python.org/pypi/PyTango
[36] http://www.taurus-scada.org/
[37] http://pypi.python.org/pypi/taurus
[38] http://lxml.de
[39] http://pypi.python.org/pypi/lxml

   (d) Download and install latest itango from itango download page[40]

2. Finally download and install latest Sardana from Sardana dowdnoad page[41]

Working directly from Git

If you intend to do changes to Sardana itself, or want to try the latest developments, it is convenient to work directly from the git source in "develop" (aka "editable") mode, so that you do not need to re-install on each change.

You can clone sardana from the main git repository:

```
git clone https://github.com/sardana-org/sardana.git sardana
```

Then, to work in editable mode, just do:

```
pip install -e ./sardana
```

Dependencies

Sardana has dependencies on some python libraries:

- Sardana uses Tango as the middleware so you need PyTango[42] 7 or later installed. You can check it by doing:

  ```
  python -c 'import PyTango; print PyTango.Release.version'
  ```

- Sardana clients are developed with Taurus so you need Taurus[43] 3.6.0 or later installed. You can check it by doing:

  ```
  python -c 'import taurus; print taurus.Release.version'
  ```

- Sardana operate some data in the XML format and requires lxml[44] library 2.1 or later. You can check it by doing:

  ```
  python -c 'import lxml.etree; print lxml.etree.LXML_VERSION'
  ```

- spock (Sardana CLI) requires itango 0.0.1 or later[3].

Running Sardana as a tango server

Note: if you have Tango <= 7.2.6 without all patches applied, Sardana server will not work due to a known bug. Please follow the instructions from *Running Pool and MacroServer tango servers separately* instead.

Sardana is based on a client-server architecture. On the server part, sardana can be setup with many different configurations. Advanced details on sardana server configuration can be found here **<LINK>**.

[40] http://pypi.python.org/pypi/itango
[41] http://pypi.python.org/pypi/sardana
[42] http://pytango.readthedocs.io/
[43] http://www.taurus-scada.org/
[44] http://lxml.de
[3] PyTango < 9 is compatible with itango >= 0.0.1 and < 0.1.0, while higher versions with itango >= 0.1.6.

This chapter describes how to run sardana server with it's simplest configuration. The only decision you have to make is which name you will give to your system. From here on *lab-01* will be used as the system name. Please replace this name with your own system name whenever apropriate.

The sardana server is called (guess what) *Sardana*. To start the server just type in the command line:

```
homer@pc001:~$ Sardana lab-01
```

The first time the server is executed, it will inform you that server *lab-01* is not registered and it will offer to register it. Just answer 'y'. This will register a new instance of Sardana called *lab-01* and the server will be started. You should get an output like this:

```
homer@pc001:~$ Sardana lab-01
lab-01 does not exist. Do you wish create a new one (Y/n) ? y
DServer/Sardana/Lab-01 has no event channel defined in the database - creating it
```

That't it! You now have a running sardana server. Not very impressive, is it? The *Running the client* chapter describes how to start up a *CLI* application called *spock* which connects to the sardana server you have just started through an object of type *Door* called *Door_lab-01_1*.

You can therefore skip the next chapter and go directly to *Running the client*.

Running Pool and MacroServer tango servers separately

Note: You should only read this chapter if you have Tango <= 7.2.6 without all patches applied. If you do, please follow in instructions from *Running Sardana as a tango server* instead.

It is possible to separate sardana server into two different servers (in the first sardana versions, this was actually the only way start the sardana system). These servers are called *Pool* and *MacroServer*. The *Pool* server takes care of hardware communication and *MacroServer* executes procedures (macros) using a connection to Pool(s) server(s).

To start the Pool server just type in the command line:

```
homer@pc001:~$ Pool lab-01
```

The first time the server is executed, it will inform you that server *lab-01* is not registered and it will offer to register it. Just answer 'y'. This will register a new instance of Pool called *lab-01* and the server will be started. You should get an output like this:

```
homer@pc001:~$ Pool lab-01
lab-01 does not exist. Do you wish create a new one (Y/n) ? y
DServer/Pool/Lab-01 has no event channel defined in the database - creating it
```

Next, start the MacroServer server in the command line:

```
homer@pc001:~$ MacroServer lab-01
```

The first time the server is executed, it will inform you that server *lab-01* is not registered and it will offer to register it. Just answer 'y'. Next, it will ask you to which Pool(s) you want your MacroServer to communicate with. Select the previously created Pool from the list, press `Return` once and `Return` again to finish with Pool selection. This will register a new instance of MacroServer called *lab-01* and the server will be started. You should get an output like this:

```
homer@pc001:~$ MacroServer lab-01
lab-01 does not exist. Do you wish create a new one (Y/n) ?
Pool_lab-01_1 (a.k.a. Pool/lab-01/1) (running)
Please select pool to connect to (return to finish): Pool_lab-01_1
Please select pool to connect to (return to finish):
DServer/MacroServer/lab-01 has no event channel defined in the database - creating it
```

Running the client

After the server has been started, you can start one or more client applications (*CLI*s and/or *GUI*s) that connect to the server. Each client connects to a specific *door* on the server. A single sardana can be configured with many *doors* allowing multiple clients to be connected at the same time.

When the sardana server was first executed, part of the registration process created one *door* for you so now you just have to start the client application from the command line:

```
homer@pc001:~$ spock
```

Spock is an IPython[45] based *CLI*. When you start spock without arguments it will assume a default profile called *spockdoor*. The first time spock is executed, it will inform you that profile *spockdoor* doesn't exist and it will offer to create one. Just answer 'y'. After, it will ask you to which *door* should the default *spockdoor* profile connect to. Select the door name corresponding to your sardana server (*Door_lab-01_1*) and press return. By now you should get an output like this:

```
homer@pc001:~$ spock
Profile 'spockdoor' does not exist. Do you want to create one now ([y]/n)? y
Available Door devices from sardanamachine:10000 :
Door_lab-01_1 (a.k.a. Door/lab-01/1)
Door name from the list? Door_lab-01_1

Storing ipython_config.py in /home/homer/.config/ipython/profile_spockdoor... [DONE]
Spock 1.0.0 -- An interactive laboratory application.

help      -> Spock's help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

IPython profile: spockdoor

Connected to Door_lab-01_1

Door_lab-01_1 [1]:
```

That's it! You now have a running sardana client. Still not impressed, I see! The next chapter describes how to start adding new elements to your sardana environment.

Populating your sardana with items

One of sardana's goals is to allow you to execute *procedures* (what we call in sardana *macros*, hence from here on we will use the term *macro*). A *macro* is basically a piece of code. You can write macros using the Python[46] language to do all sorts of things. The sky is the limit here!

[45] http://ipython.org/
[46] http://www.python.org/

Sardana comes with a *catalog of macros* that help users in a laboratory to run their experiments. Most of these *macros* involve interaction with sardana elements like motors and experimental channels. Therefore, the first step in a new sardana demo is to populate your system with some elements. Fortunately, sardana comes with a *macro* called *sar_demo* that does just that. To execute this *macro* just type on the command line `sar_demo`. You should get an output like this:

```
Door_lab-01_1 [1]: sar_demo

Creating controllers motctrl01, ctctrl01... [DONE]
Creating motors mot01, mot02, mot03, mot04... [DONE]
Creating measurement group mntgrp01... [DONE]
```

You should now have in your sardana system a set of simulated motors and counters with which you can play.

Hint: for clearing sardana from the elements created by the demo, execute `clear_sar_demo`

The next chapter (*spock*) will give you a complete overview of spock's interface.

Spock

Spock is the prefered *CLI* for sardana. It is based on IPython[47]. Spock automatically loads other IPython[48] extensions like the ones for PyTango[49] and *pylab*. It as been extended in sardana to provide a customized interface for executing macros and automatic access to sardana elements.

Spock tries to mimic SPEC[50]'s command line interface. Most SPEC[51] commands are available from spock console.

Starting spock from the command line

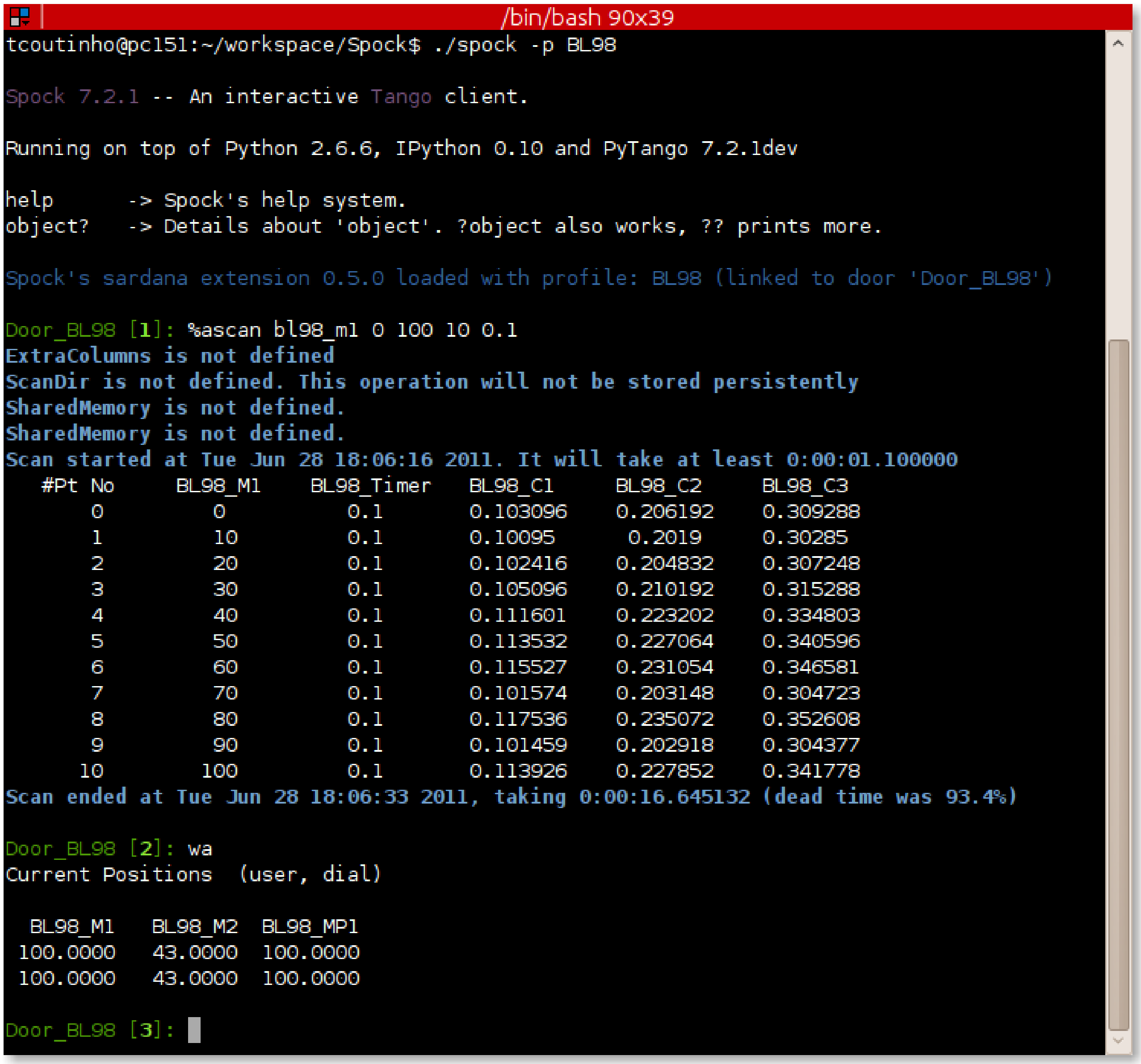
To start spock just type in the command line:

```
marge@machine02:~$ spock
```

This will start spock with a "default profile" for the user your are logged with. There may be many sardana servers running on your system so the first time you start spock, it will ask you to which sardana system you want to connect to by asking to which of the existing doors you want to use:

```
marge@machine02:~$ spock
Profile 'spockdoor' does not exist. Do you want to create one now ([y]/n)?
Available Door devices from homer:10000 :
On Sardana LAB-01:
    LAB-01-D01 (running)
    LAB-01-D02 (running)
On Sardana LAB-02:
    LAB-02-D01
Please select a Door from the list? LAB-01-D01
Storing ipy_profile_spockdoor.py in /home/marge/.ipython... [DONE]
```

[47] http://ipython.org/
[48] http://ipython.org/
[49] http://packages.python.org/PyTango/
[50] http://www.certif.com/
[51] http://www.certif.com/



```
tcoutinho@pc151:~/workspace/Spock$ ./spock -p BL98

Spock 7.2.1 -- An interactive Tango client.

Running on top of Python 2.6.6, IPython 0.10 and PyTango 7.2.1dev

help      -> Spock's help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

Spock's sardana extension 0.5.0 loaded with profile: BL98 (linked to door 'Door_BL98')

Door_BL98 [1]: %ascan bl98_m1 0 100 10 0.1
ExtraColumns is not defined
ScanDir is not defined. This operation will not be stored persistently
SharedMemory is not defined.
SharedMemory is not defined.
Scan started at Tue Jun 28 18:06:16 2011. It will take at least 0:00:01.100000
   #Pt No     BL98_M1    BL98_Timer   BL98_C1     BL98_C2     BL98_C3
       0           0          0.1      0.103096    0.206192    0.309288
       1          10          0.1      0.10095      0.2019     0.30285
       2          20          0.1      0.102416    0.204832    0.307248
       3          30          0.1      0.105096    0.210192    0.315288
       4          40          0.1      0.111601    0.223202    0.334803
       5          50          0.1      0.113532    0.227064    0.340596
       6          60          0.1      0.115527    0.231054    0.346581
       7          70          0.1      0.101574    0.203148    0.304723
       8          80          0.1      0.117536    0.235072    0.352608
       9          90          0.1      0.101459    0.202918    0.304377
      10         100          0.1      0.113926    0.227852    0.341778
Scan ended at Tue Jun 28 18:06:33 2011, taking 0:00:16.645132 (dead time was 93.4%)

Door_BL98 [2]: wa
Current Positions  (user, dial)

  BL98_M1    BL98_M2   BL98_MP1
 100.0000    43.0000   100.0000
 100.0000    43.0000   100.0000

Door_BL98 [3]: 
```

Fig. 1: Spock *CLI* in action

Note: If only one Door exists in the entire system, spock will automatically connect to that door thus avoiding the previous questions.

Afterward, spock *CLI* will start normally:

```
Spock 7.2.1 -- An interactive sardana client.

help      -> Spock's help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

Spock's sardana extension 1.0 loaded with profile: spockdoor (linked to door 'LAB-01-
↪D01')

LAB-01-D01 [1]:
```

Starting spock with a custom profile

spock allows each user to start a spock session with different configurations (known in spock as *profiles*). All you have to do is start spock with the profile name as an option.

If you use ipython version > 0.10 you can do it using **–profile** option:

```
spock --profile=<profile name>
```

Example:

```
marge@machine02:~$ spock --profile=D1
```

Otherwise (ipython version 0.10) you can do it using **-p** option:

```
spock -p <profile name>
```

Example:

```
marge@machine02:~$ spock -p D1
```

The first time a certain profile is used you will be asked to which door you want to connect to (see previous chapter).

Spock IPython[52] Primer

As mentioned before, spock console is based on IPython[53]. Everything you can do in IPython is available in spock. The IPython[54] documentation provides excelent tutorials, tips & tricks, cookbooks, videos, presentations and reference guide. For comodity we summarize some of the most interesting IPython[55] chapters here:

- IPython web page[56]

[52] http://ipython.org/
[53] http://ipython.org/
[54] http://ipython.org/
[55] http://ipython.org/
[56] http://ipython.org/

- Introducing IPython[57]
- IPython Tips & Tricks[58]
- Command-line usage[59]

Executing macros

Executing sardana macros in spock is the most useful feature of spock. It is very simple to execute a macro: just type the macro name followed by a space separated list of parameters (if the macro has any parameters). For example, one of the most used macros is the *wa* (stands for "where all") that shows all current motor positions. To execute it just type:

```
LAB-01-D01 [1]: wa

Current Positions  (user, dial)

   Energy        Gap     Offset
 100.0000    43.0000  100.0000
 100.0000    43.0000  100.0000
```

(*user* for *user position* (number above); *dial* for *dial position* (number below).)

A similar macro exists that only shows the desired motor positions (*wm*):

```
LAB-01-D01 [1]: wm gap offset
                Gap      Offset
User
 High          500.0      100.0
 Current       100.0       43.0
 Low             5.0     -100.0
Dial
 High          500.0      100.0
 Current       100.0       43.0
 Low             5.0     -100.0
```

To get the list of all existing macros use `lsmac`:

```
LAB-01-D01 [1]: lsdef
               Name         Module                                          Brief␣
↪Description
------------------- ------------- ----------------------------------------------------
↪--------
             a2scan         scans two-motor scan.     a2scan scans two motors, as␣
↪specifi[...]
             a2scan         scans three-motor scan .     a3scan scans three motors,␣
↪as sp[...]
              ascan         scans Do an absolute scan of the specified motor.     ␣
↪ascan s[...]
            defmeas        expert                                 Create a new␣
↪measurement group
              fscan         scans N-dimensional scan along user defined paths.     ␣
↪The mo[...]
                lsa         lists                                     Lists all␣
↪existing objects
```

(continues on next page)

[57] http://ipython.org/ipython-doc/stable/interactive/tutorial.html#tutorial
[58] http://ipython.org/ipython-doc/stable/interactive/tips.html#tips
[59] http://ipython.org/ipython-doc/stable/interactive/reference.html#command-line-options

(continued from previous page)

```
                  lsm         lists                                              Lists␣
↪all motors
                lsmac        expert                                              Lists␣
↪all macros.
                   mv      standard                  Move motor(s) to the specified␣
↪position(s)
                  mvr      standard           Move motor(s) relative to the current␣
↪position(s)
                   wa      standard                                    Show all motor␣
↪position.
                   wm      standard                   Show the position of the␣
↪specified motors.
<...>
```

Miscellaneous

- *lsm* shows the list of motors.
- *lsct* shows the list of counters.
- *lsmeas* shows the list of measurement groups
- *lsctrl* shows the list of controllers
- *sar_info* *object* displays detailed information about an element

Stopping macros

Some macros may take a long time to execute. To stop a macro in the middle of its execution type `Control+c`.

Macros that move motors or acquire data from sensors will automatically stop all motion and/or all acquisition.

Exiting spock

To exit spock type `Control+d` or `exit()` inside a spock console.

Getting help

spock not only knows all the macros the sardana server can run but it also information about each macro parameters, result and documentation. Therefore it can give you precise help on each macro. To get help about a certain macro just type the macro name directly followed by a question mark('?'):

```
LAB-01-D01 [1]: ascan?

Syntax:
        ascan <motor> <start_pos> <final_pos> <nr_interv> <integ_time>

Do an absolute scan of the specified motor.
    ascan scans one motor, as specified by motor. The motor starts at the
    position given by start_pos and ends at the position given by final_pos.
```

(continues on next page)

(continued from previous page)

```
    The step size is (start_pos-final_pos)/nr_interv. The number of data points␣
↪collected
    will be nr_interv+1. Count time is given by time which if positive,
    specifies seconds and if negative, specifies monitor counts.

Parameters:
        motor : (Motor) Motor to move
        start_pos : (Float) Scan start position
        final_pos : (Float) Scan final position
        nr_interv : (Integer) Number of scan intervals
        integ_time : (Float) Integration time
```

Moving motors

A single motor may be moved using the `mv` *motor position* macro. Example:

```
LAB-01-D01 [1]: mv gap 50
```

will move the *gap* motor to 50. The prompt only comes back after the motion as finished.

Alternatively, you can have the motor position displayed on the screen as it is moving by using the `umv` macro instead. To stop the motor(s) before they have finished moving, type `Control+c`.

You can use the `mvr` *motor relative_position* macro to move a motor relative to its current position:

```
LAB-01-D01 [1]: mvr gap 2
```

will move *gap* by two user units.

Counting

You can count using the `ct` *value* macro. Without arguments, this macro counts for one second using the active measurement group set by the environment variable *ActiveMntGrp*.

```
Door_lab-01_1 [1]: ct 1.6

Wed Jul 11 11:47:55 2012

  ct01  =         1.6
  ct02  =         3.2
  ct03  =         4.8
  ct04  =         6.4
```

To see the list of available measurement groups type `lsmeas`. The active measuremnt group is marked with an asterisk (*):

```
Door_lab-01_1 [1]: lsmeas

  Active        Name    Timer Experim. channels
 -------- ---------- ------- ---------------------------------------------------------
↪--
    *       mntgrp01    ct01 ct01, ct02, ct03, ct04
            mntgrp21    ct04 ct04, pcII0, pcII02
            mntgrp24    ct04 ct04, pcII0
```

to switch active measurement groups type `senv` **ActiveMntGrp** *mg_name*.

You can also create, modify and select measurement groups using the *expconf* command

Scanning

Sardana provides a catalog of different standard scan macros. Absolute-position motor scans such as `ascan`, `a2scan` and `a3scan` move one, two or three motors at a time. Relative-position motor scans are `dscan`, `d2scan` and `d3scan`. The relative-position scans all return the motors to their starting positions after the last point. Two motors can be scanned over a grid of points using the `mesh` scan.

Continuous versions exist of many of the standard scan macros (e.g. `ascanc`, `d3scanc`, `meshc`,…). The continuous scans differ from their standard counterparts (also known as *step* scans) in that the data acquisition is done without stopping the motors. Continuous scans are generally faster but less precise than step scans, and some details must be considered (see *Scans*).

As it happens with `ct`, the scan macros will also use the active measurement group to decide which experiment channels will be involved in the operation.

Here is the output of performing an `ascan` of the gap in a slit:

```
LAB-01-D01 [1]: ascan gap 0.9 1.1 20 1
ScanDir is not defined. This operation will not be stored persistently. Use "senv␣
↪ScanDir <abs directory>" to enable it
Scan #4 started at Wed Jul 11 12:56:47 2012. It will take at least 0:00:21
 #Pt No    gap       ct01      ct02      ct03
  0        0.9          1      4604      8939
  1       0.91          1      5822      8820
  2       0.92          1      7254      9544
  3       0.93          1      9254      8789
  4       0.94          1     11265      8804
  5       0.95          1     13583      8909
  6       0.96          1     15938      8821
  7       0.97          1     18076      9110
  8       0.98          1     19638      8839
  9       0.99          1     20825      8950
 10          1          1     21135      8917
 11       1.01          1     20765      9013
 12       1.02          1     19687      9135
 13       1.03          1     18034      8836
 14       1.04          1     15876      8901
 15       1.05          1     13576      8933
 16       1.06          1     11328      9022
 17       1.07          1      9244      9205
 18       1.08          1      7348      8957
 19       1.09          1      5738      8801
 20        1.1          1      4575      8975
Scan #4 ended at Wed Jul 11 12:57:18 2012, taking 0:00:31.656980 (dead time was 33.7%)
```

Scan storage

As you can see, by default, the scan is not recorded into any file. To store your scans in a file, you must set the environment variables **ScanDir** and **ScanFile**:

```
LAB-01-D01 [1]: senv ScanDir /tmp
ScanDir = /tmp

LAB-01-D01 [2]: senv ScanFile scans.h5
ScanFile = scans.h5
```

Sardana will activate a proper recorder to store the scans persistently (currently, *.h5* will store in NeXus[60] format. All other extensions are interpreted as SPEC[61] format).

You can also store in multiples files by assigning the **ScanFile** with a list of files:

```
LAB-01-D01 [2]: senv ScanFile "['scans.h5', 'scans.dat']"
ScanFile = ['scans.h5', 'scans.dat']
```

Viewing scan data

Sardana provides a scan data viewer for scans which were stored in a NeXus[62] file. Without arguments, `showscan` will show you the result of the last scan in a *GUI*:

`showscan` *scan_number* will display data for the given scan number.

The history of scans is available through the *scanhist* macro:

```
LAB-01-D01 [1]: scanhist
   #                            Title               Start time                End time     ␣
↪  Stored
 --- ------------------------------- --------------------- --------------------- -----
↪--------
   1     dscan mot01 20.0 30.0 10 0.1   2012-07-03 10:35:30   2012-07-03 10:35:30  ␣
↪Not stored!
   3     dscan mot01 20.0 30.0 10 0.1   2012-07-03 10:36:38   2012-07-03 10:36:43  ␣
↪Not stored!
   4    ascan gap01 10.0 100.0 20 1.0            12:56:47              12:57:18  ␣
↪Not stored!
   5      ascan gap01 1.0 10.0 20 0.1            13:19:05              13:19:13     ␣
↪scans.h5
```

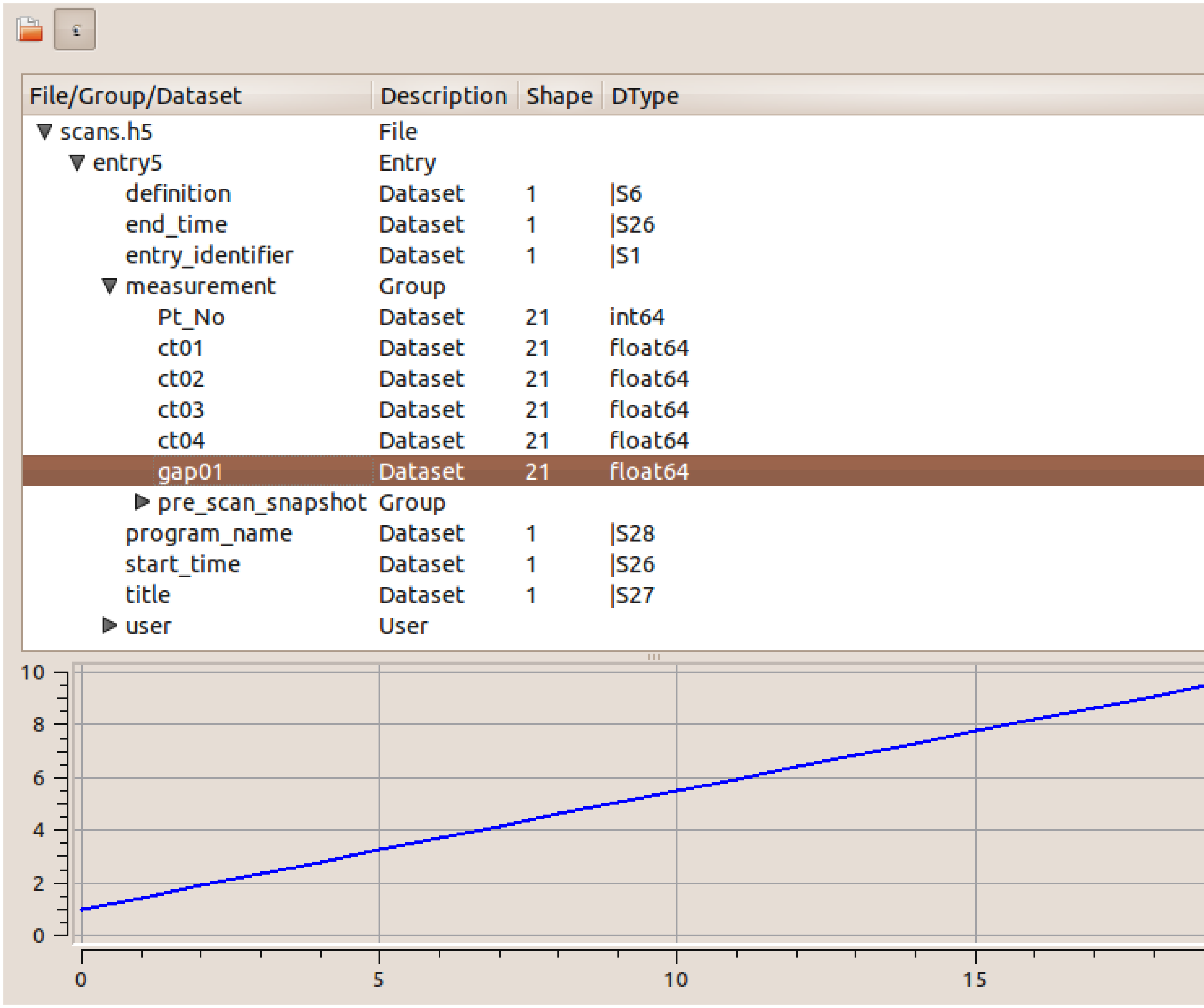
Accessing macro data

The command `macrodata` allows to retrieve the data of the last macro run in spock. If this macro does not provide any data an error message is thrown. Example accesing scan data:

```
Door_1 [9]: ascan mot17 1 10 2 1
ScanDir is not defined. This operation will not be stored persistently. Use "expconf"␣
↪(or "senv ScanDir <abs directory>") to enable it
Scan #2 started at Tue Feb 13 11:16:18 2018. It will take at least 0:00:05.048528
0          1          1          3          4       0.865325
1         5.5         1          3          4       2.51148
2          10         1          3          4       4.16662
Scan #2 ended at Tue Feb 13 11:16:24 2018, taking 0:00:05.201949. Dead time 42.3%␣
↪(motion dead time 40.5%)
```

(continues on next page)

[60] http://www.nexusformat.org/
[61] http://www.certif.com/
[62] http://www.nexusformat.org/



File/Group/Dataset	Description	Shape	DType
▼ scans.h5	File		
▼ entry5	Entry		
definition	Dataset	1	|S6
end_time	Dataset	1	|S26
entry_identifier	Dataset	1	|S1
▼ measurement	Group		
Pt_No	Dataset	21	int64
ct01	Dataset	21	float64
ct02	Dataset	21	float64
ct03	Dataset	21	float64
ct04	Dataset	21	float64
gap01	Dataset	21	float64
▶ pre_scan_snapshot	Group		
program_name	Dataset	1	|S28
start_time	Dataset	1	|S26
title	Dataset	1	|S27
▶ user	User		

Fig. 2: Scan data viewer in action

(continued from previous page)

```
#Pt No     mot17       ct17       ct19       ct20        dt
Door_1 [10]: r = %macrodata
Door_1 [11]: r[0].data.keys()
Result [11]:
['point_nb',
'timestamp',
'mot17',
'haso111n:10000/expchan/ctctrl05/4',
'haso111n:10000/expchan/ctctrl05/1',
'haso111n:10000/expchan/ctctrl05/3']
Door_1 [12]: r[0].data['point_nb']
Result [12]: 0
Door_1 [13]: r[0].data['mot17']
Result [13]: 1.0
Door_1 [16]: r[0].data['haso111n:10000/expchan/ctctrl05/1']
Result [16]: 1.0
```

Editing macros

The command `edmac` allows to edit the macros directly from spock. See *Writing macros* section.

Debugging problems

Spock provides some commands that help to debug or recognize the errors in case a macro fails when being executed.

- `www` prints the error message from the last macro execution
- `debug` used with `on` as parameter activates the print out of the debug messages during macro execution. Set it to `off` to deactivate it.
- `post_mortem` prints the current logger messages. If no argument is specified it reads the `debug` stream. Valid values are `output`, `critical`, `error`, `warning`, `info`, `debug` and `result`.

Using spock as a Python[63] console

You can write any Python[64] code inside a spock console since spock uses IPython[65] as a command line interpreter. For example, the following will work inside a spock console:

```
LAB-01-D01 [1]: def f():
           ...:     print("Hello, World!")
           ...:
           ...:

LAB-01-D01 [2]: f()
Hello, World!
```

[63] http://www.python.org/
[64] http://www.python.org/
[65] http://ipython.org/

Using spock as a Tango[66] console

As metioned in the beggining of this chapter, the sardana spock automatically activates the PyTango[67] 's ipython console extension. Therefore all Tango[68] features are automatically available on the sardana spock console. For example, creating a `DeviceProxy` will work inside the sardana spock console:

```
LAB-01-D01 [1]: tgtest = PyTango.DeviceProxy("sys/tg_test/1")

LAB-01-D01 [2]: print( tgtest.state() )
RUNNING
```

Sardana Taurus Extension widgets

Sardana provides several `taurus`[72]-based widgets for being used in GUIs

MacroExecutor User's Interface

Contents

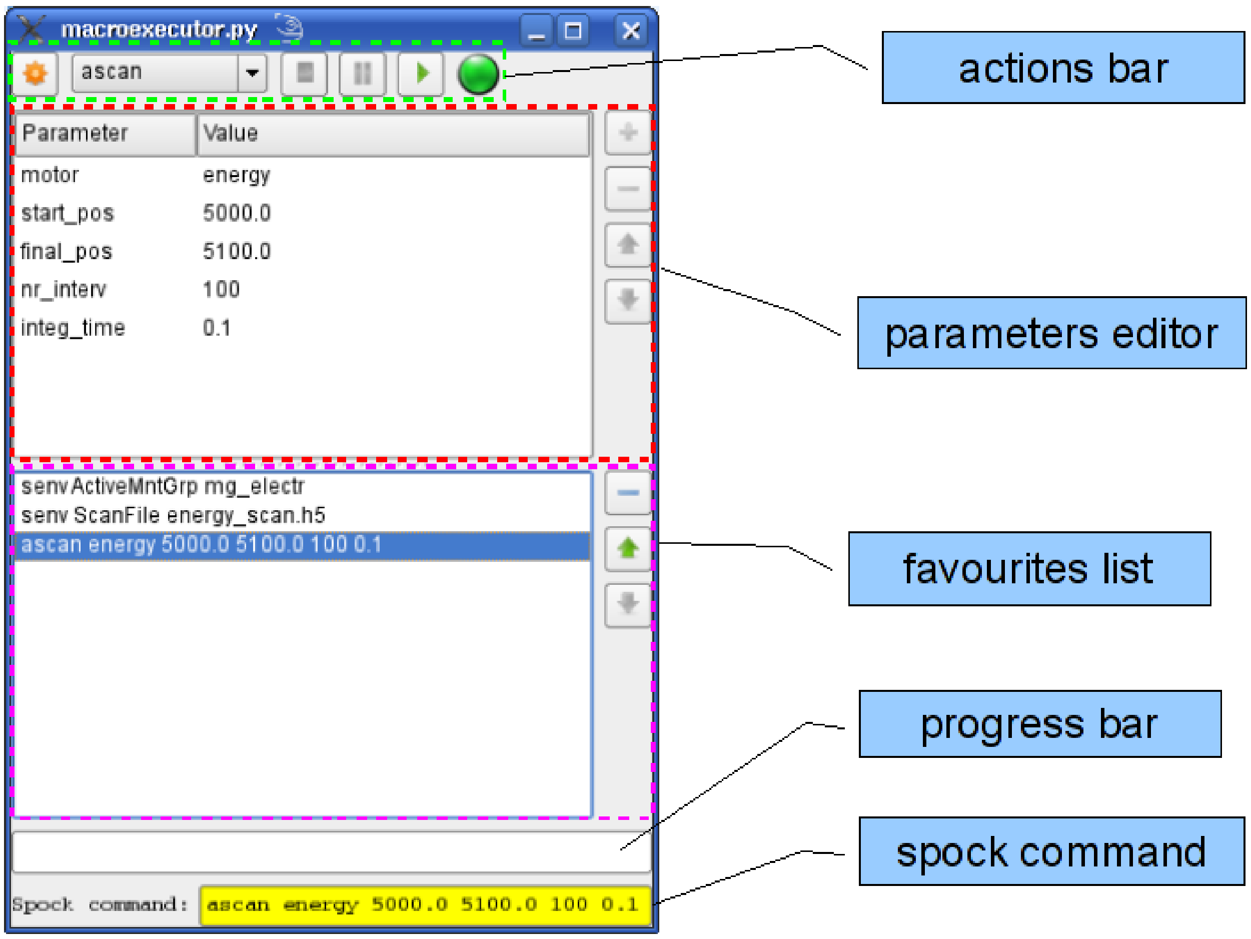
MacroExecutor provides an user-friendly graphical interface to macro execution. It is divided into 3 main areas: *actions bar*, *parameters editor* and *favourites list*. Their functionalities are supported by *Spock command line* and *macro progress bar*. User has full control over macros thanks to action buttons: Start(Resume), Stop, Pause located in *actions bar* Graphical *parameters editor* provides a clear way to set and modify macro execution settings (parameters). Macros which are more frequently used can be permanently stored in *favourites list*. Once macro was started Door's state led and *macro progress bar* informs user about its status. Current macro settings (parameters) are translated to spock syntax, and represented in non editable *spock command line*.

MacroExecutor as a stand-alone application

You may also use *MacroExecutor* as a stand-alone application. In this case it appears embedded in window and some extra functionalities are provided. You can launch the stand-alone *MacroExecutor* with the following command:

```
macroexecutor [options] [<macro_executor_dev_name> <door_dev_name>]
```

[66] http://www.tango-controls.org/
[67] http://packages.python.org/PyTango/
[68] http://www.tango-controls.org/
[72] http://taurus-scada.org/devel/api/taurus.html#module-taurus



Parameter	Value
motor	energy
start_pos	5000.0
final_pos	5100.0
nr_interv	100
integ_time	0.1

```
senv ActiveMntGrp mg_electr
senv ScanFile energy_scan.h5
ascan energy 5000.0 5100.0 100 0.1
```

Spock command: `ascan energy 5000.0 5100.0 100 0.1`

Options:

```
--taurus-log-level=LEVEL
                      taurus log level. Allowed values are (case
                      insensitive): critical, error, warning/warn, info,
                      debug, trace

--taurus-polling-period=MILLISEC
                      taurus global polling period in milliseconds

--taurus-serialization-mode=SERIAL
                      taurus serialization mode. Allowed values are (case
                      insensitive): serial, concurrent (default)

--tango-host=TANGO_HOST
                      Tango host name
```

The model list is optional and is a space-separated list of two device names: macro server and door. If not provided at the application startup, models can be later on changed in configuration dialog.

Extra functionalities:

- Changing macro configuration

Todo: This chapter is not ready... Sorry for inconvenience.

- Configuring custom editors

Todo: This chapter is not ready... Sorry for inconvenience.

Editing macro parameters

Using standard editor

If no custom parameter editor is assigned to macro, default editor is used to configure execution settings (parameters). Parameters are represented in form of tree (with hidden root node) - every parameter is a separate branch with two columns: parameter name and parameter value. Editor is populated with default values of parameters, if this in not a case 'None' values are used. (If macro execution settings were restored e.g. from favourites list, editor is populated with stored values). Values become editable either by double-clicking on them, or by pressing F2 button when value is selected. This action opens default parameter editor (combobox with predefined values, spin box etc.).

In case of macros with single parameters only, tree has only a one level branch, and then tree representation looks more like a list (because of hidden root node)

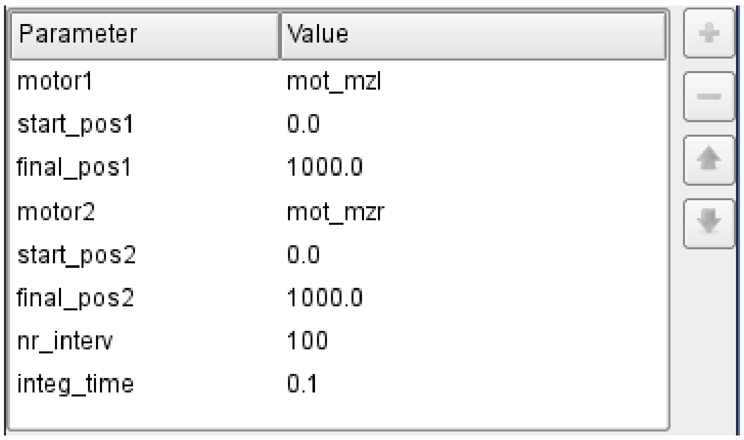
In case of macros which contain repeat parameters, concept of tree is more visible.

- adding new parameter repetition

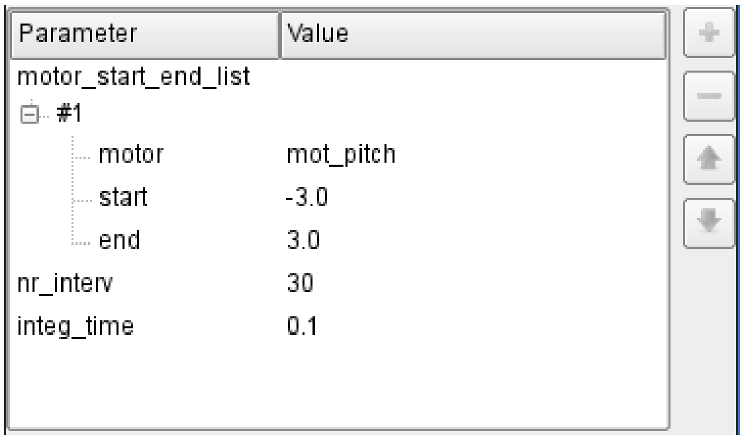
First select parameter node and if its maximum number of repetition is not exceeded, button with '+' sign appears enabled. After pressing this button child branch with new repetition appears in tree editor.

- modifying repetition order

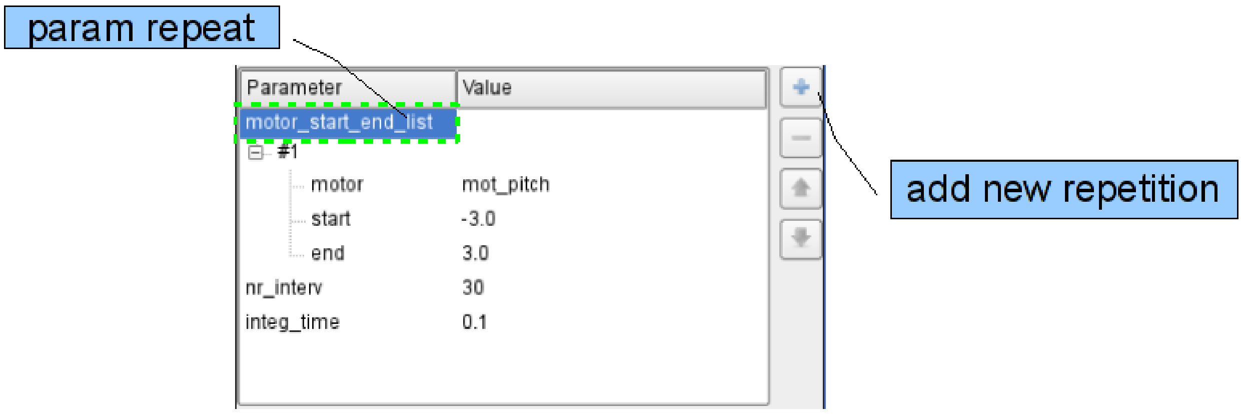
First select repetition node (with #<number> text), and buttons with arrows becomes enable (if it is feasible to change order)



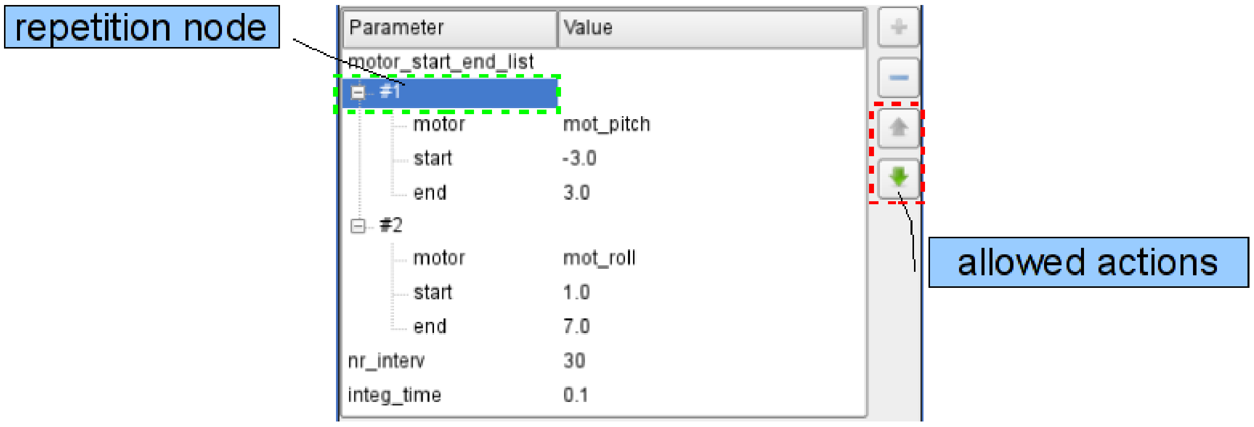
Parameter	Value
motor1	mot_mzl
start_pos1	0.0
final_pos1	1000.0
motor2	mot_mzr
start_pos2	0.0
final_pos2	1000.0
nr_interv	100
integ_time	0.1



Parameter	Value
motor_start_end_list	
#1	
motor	mot_pitch
start	-3.0
end	3.0
nr_interv	30
integ_time	0.1



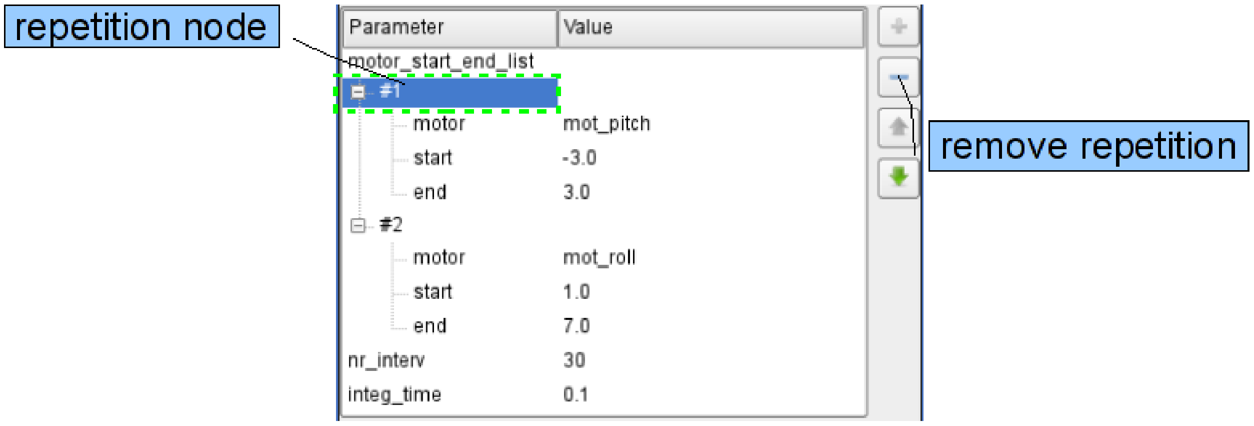
Parameter	Value
motor_start_end_list	
#1	
motor	mot_pitch
start	-3.0
end	3.0
nr_interv	30
integ_time	0.1



Parameter	Value
motor_start_end_list	
#1	
motor	mot_pitch
start	-3.0
end	3.0
#2	
motor	mot_roll
start	1.0
end	7.0
nr_interv	30
integ_time	0.1

- removing parameter repetition

First select repetition node (with #<number> text), and if it's minimum number of repetition is not reached, button with '-' sign appears enabled. After pressing this button child branch disappears from tree editor. (see previous picture)



Parameter	Value
motor_start_end_list	
#1	
motor	mot_pitch
start	-3.0
end	3.0
#2	
motor	mot_roll
start	1.0
end	7.0
nr_interv	30
integ_time	0.1

Using custom editors

Todo: This chapter is not ready... Sorry for inconvenince.

Editing favourites list

Once macro parameters are configured they can be easily stored in favourites list for later reuse.

- adding a favourite

Clicking in Add to favourites button (the one with yellow star), adds a new entry in favourite list, with current macro and its current settings.

- restoring a favourite

To restore macro from favourites list just select it in the list and macro parameters editor will immediately populate with stored settings.

- modifying favouites list

First select favourite macro and buttons with arrows becomes enable (if it is feasible to change order)

- removing a favourite

First select favourite macro, button with '-' sign appears enabled. After pressing this button, previously selected macro disappears from the list.

Sequencer User's Interface

Contents

Sequencer provides an user-friendly interface to compose and execute sequences of macros. Sequence of macros allows execution of ordered set of macros with just one trigger. It also allows using a concept of hooks (macros attached and executed in defined places of other macros). It is divided into 3 main areas: *actions bar*, *sequence editor* and *parameters editor*. *Sequence editor* allows you modifying sequences in many ways: appending new macros, changing macros locations and removing macros. Graphical *parameters editor* (standard/custom) provides a clear way to set/modify macro execution settings(parameters). Once sequence of macros is in execution phase, *Sequencer* informs user about its state with Door's state led and macros progress bars. User has full control over sequence, with action buttons: Start, Stop, Pause, Resume. If desirable, sequences can be permanently stored into a file and later on restored from there. This functionality is provided thanks to action buttons: Save and Open a sequence.

Sequencer as a stand-alone application

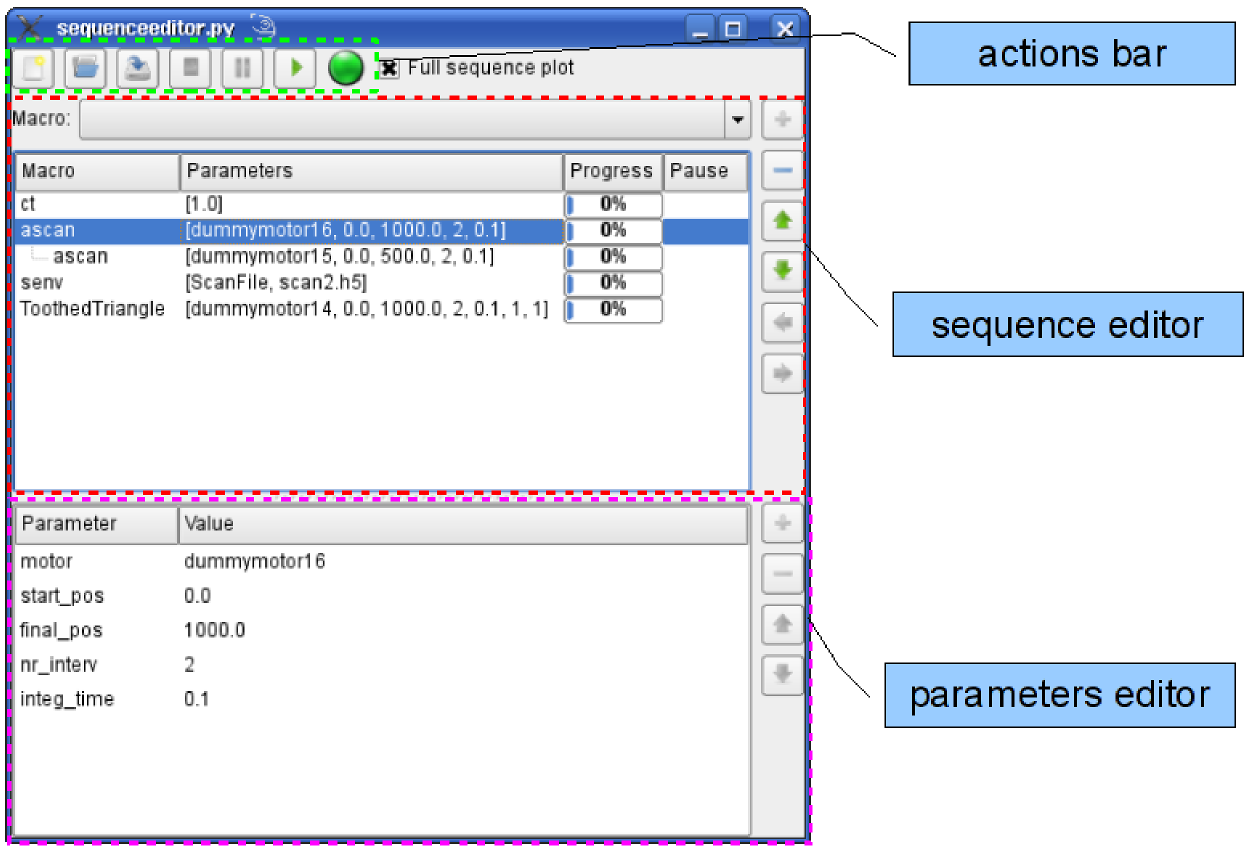
You may also use *Sequencer* as a stand-alone application. In this case it appears embedded in window and some extra functionalities are provided. You can launch the stand-alone *Sequencer* with the following command:

```
sequencer [options] [<macro_executor_dev_name> <door_dev_name>]
```

Options:

```
--taurus-log-level=LEVEL
                      taurus log level. Allowed values are (case
                      insensitive): critical, error, warning/warn, info,
                      debug, trace

--taurus-polling-period=MILLISEC
                      taurus global polling period in milliseconds

--taurus-serialization-mode=SERIAL
                      taurus serialization mode. Allowed values are (case
                      insensitive): serial, concurrent (default)
```

(continues on next page)



Macro	Parameters	Progress	Pause
ct	[1.0]	0%	
ascan	[dummymotor16, 0.0, 1000.0, 2, 0.1]	0%	
ascan	[dummymotor15, 0.0, 500.0, 2, 0.1]	0%	
senv	[ScanFile, scan2.h5]	0%	
ToothedTriangle	[dummymotor14, 0.0, 1000.0, 2, 0.1, 1, 1]	0%	

Parameter	Value
motor	dummymotor16
start_pos	0.0
final_pos	1000.0
nr_interv	2
integ_time	0.1

(continued from previous page)

```
--tango-host=TANGO_HOST
                        Tango host name
```

The model list is optional and is a space-separated list of two device names: macro server and door. If not provided at the application startup, device names can be later on selected from Macro Configuration Dialog.

Extra functionalities:

- MacroConfigurationDialog

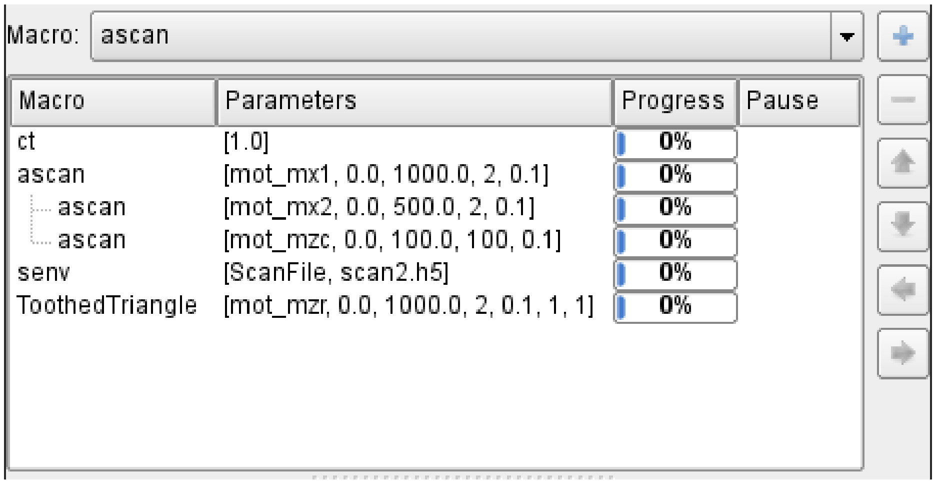
Todo: This chapter in not ready… Sorry for inconvenience.

- CustomEditorsPathDialog

Todo: This chapter in not ready… Sorry for inconvenience.

Editing sequence

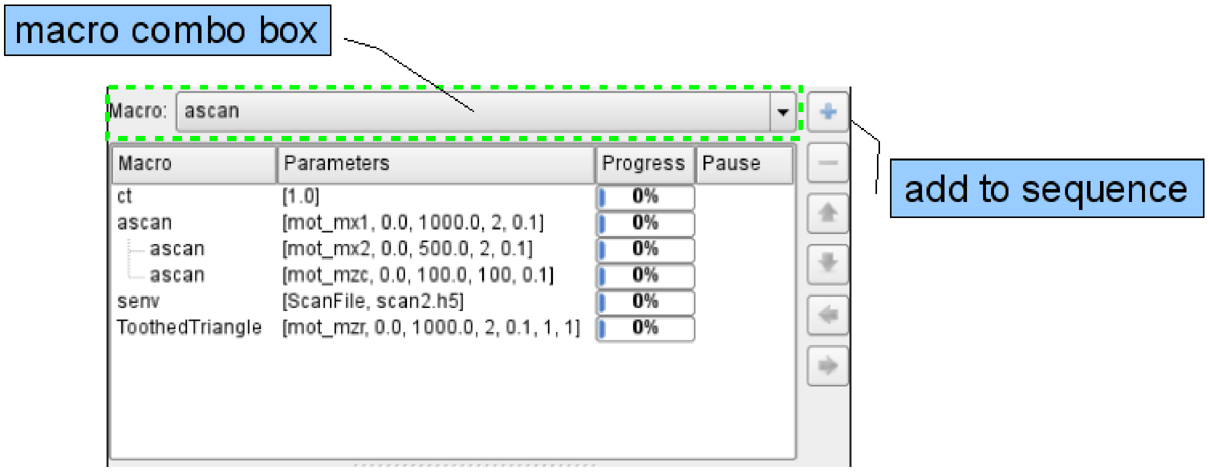
Sequence is represented as a flat list of ordered macros, in this view each macro is represented as a new line with 4 columns: Macro (macro name), Parameters (comma separated parameter values), Progress (macro progress bar) and Pause (pause point before macro execution - not implemented yet). Macros which contain hooks, expand with branched macros. Macro parameters values can be edited from *parameters editor*, to do so select one macro in sequence editor by clicking on it. Selected macro becomes highlighted, and *parameters editor* populate with its current parameters values.



Macro	Parameters	Progress	Pause
ct	[1.0]	0%	
ascan	[mot_mx1, 0.0, 1000.0, 2, 0.1]	0%	
ascan	[mot_mx2, 0.0, 500.0, 2, 0.1]	0%	
ascan	[mot_mzc, 0.0, 100.0, 100, 0.1]	0%	
senv	[ScanFile, scan2.h5]	0%	
ToothedTriangle	[mot_mzr, 0.0, 1000.0, 2, 0.1, 1, 1]	0%	

- adding a new macro

First select macro from macro combo box, and when you are sure to add it to the sequence, press '+' button. To add macro as a hook of other macro, before adding it, please select its parent macro in the sequence, and then press '+' button. If no macro was selected as a parent, macro will be automatically appended at the end of the list.



Macro	Parameters	Progress	Pause
ct	[1.0]	0%	
ascan	[mot_mx1, 0.0, 1000.0, 2, 0.1]	0%	
ascan	[mot_mx2, 0.0, 500.0, 2, 0.1]	0%	
ascan	[mot_mzc, 0.0, 100.0, 100, 0.1]	0%	
senv	[ScanFile, scan2.h5]	0%	
ToothedTriangle	[mot_mzr, 0.0, 1000.0, 2, 0.1, 1, 1]	0%	

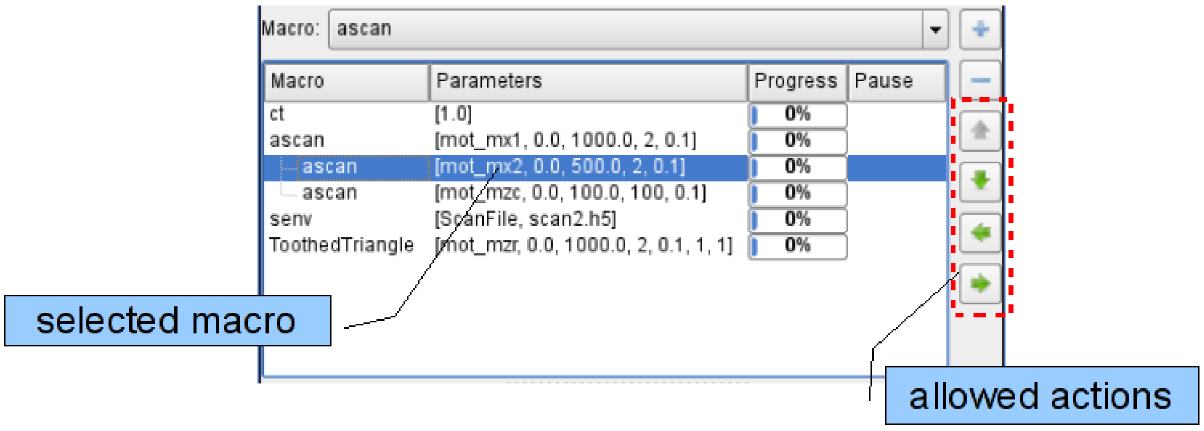
- reorganizing sequence

Macros which are already part of a sequence, can be freely moved around, either in execution order or in hook place (if new macro accepts hooks). To move macro first select it in the sequence by single clicking on it (it will become highlighted). Then a set of buttons with arrows become enabled. Clicking on them will cause selected macro changin its position in the sequence (either vertically - execution order or horizontal parent macro - hook macro relationship)

- remove macro

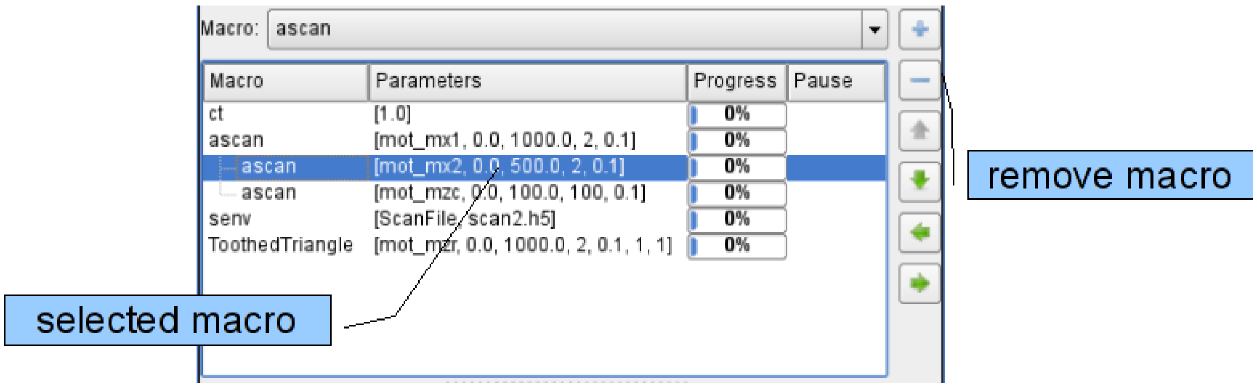
Macros which are already part of a sequence, can be freely removed from it. To do so first select macro in a sequence by single clicking on it (it will become highlighted). Then button with '-' becomes enabled. Clicking on it removes selected macro.

- configuring hook execution place



Macro: ascan

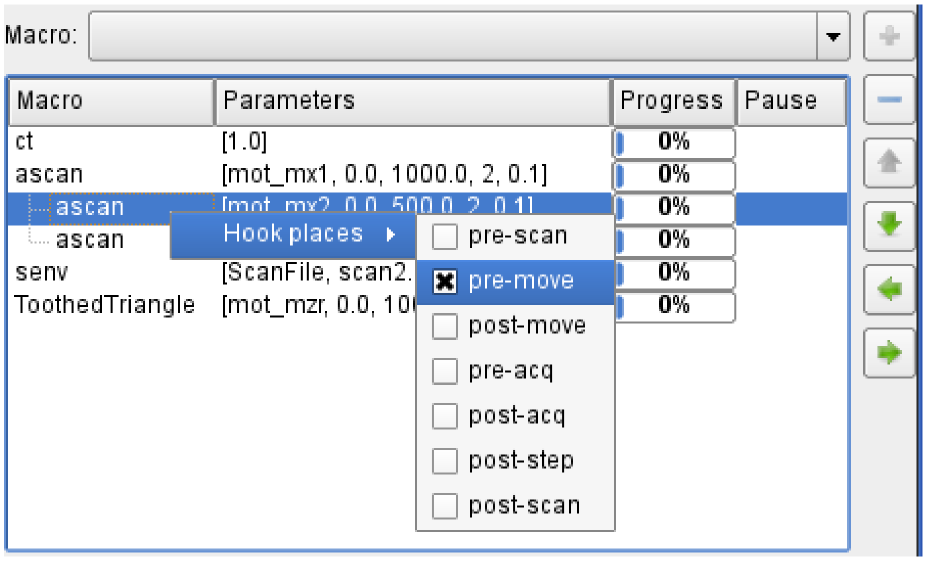
Macro	Parameters	Progress	Pause
ct	[1.0]	0%	
ascan	[mot_mx1, 0.0, 1000.0, 2, 0.1]	0%	
ascan	[mot_mx2, 0.0, 500.0, 2, 0.1]	0%	
ascan	[mot_mzc, 0.0, 100.0, 100, 0.1]	0%	
senv	[ScanFile, scan2.h5]	0%	
ToothedTriangle	[mot_mzr, 0.0, 1000.0, 2, 0.1, 1, 1]	0%	



Macro: ascan

Macro	Parameters	Progress	Pause
ct	[1.0]	0%	
ascan	[mot_mx1, 0.0, 1000.0, 2, 0.1]	0%	
ascan	[mot_mx2, 0.0, 500.0, 2, 0.1]	0%	
ascan	[mot_mzc, 0.0, 100.0, 100, 0.1]	0%	
senv	[ScanFile, scan2.h5]	0%	
ToothedTriangle	[mot_mzr, 0.0, 1000.0, 2, 0.1, 1, 1]	0%	

If macro is embedded as a hook in parent macro, please follow these instructions to configure its hook execution place. First select macro in a sequence by single clicking on it (it will become highlighted). Then using right mouse button open context menu, go to 'Hook places' sub-menu and select hook places which interest you (you can select more than one).



Macro	Parameters	Progress	Pause
ct	[1.0]	0%	
ascan	[mot_mx1, 0.0, 1000.0, 2, 0.1]	0%	
ascan	[mot_mx2, 0.0, 500.0, 2, 0.1]	0%	
ascan		0%	
senv	[ScanFile, scan2.	0%	
ToothedTriangle	[mot_mzr, 0.0, 10	0%	

Editing macro parameters

To obtain information about editing macro parameters, please refer to the following link *Editing macro parameters*

Experiment Configuration user interface

Contents

Experiment Configuration widget a.k.a. expconf is a complete interface to define all the experiment configuration. It consists of three main groups of parameters organized in tabs:

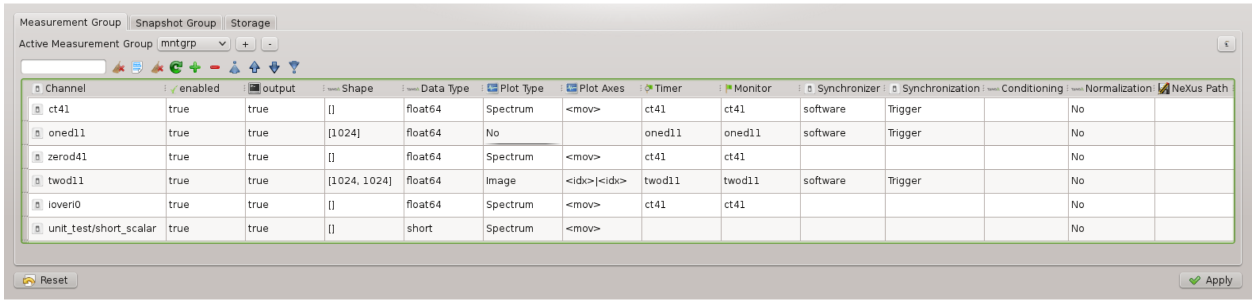
- Measurement group
- Snapshot group
- Storage

The parameters may be modified in an arbitrary order, at any of the tabs, and will be maintained as pending to apply until either applied or reset by the user.

Measurement group configuration

In the measurement group tab the user can:

- create or remove a measurement group
- select the active measurement group
- add or remove channels of the measurement group
- reorganize the order of the channels in the measurement group
- change configuration of a particular channel (or its controller) in the selected measurement group



Channel	enabled	output	Shape	Data Type	Plot Type	Plot Axes	Timer	Monitor	Synchronizer	Synchronization	Conditioning	Normalization	NeXus Path
ct41	true	true	[]	float64	Spectrum	<mov>	ct41	ct41	software	Trigger		No	
oned11	true	true	[1024]	float64	No		oned11	oned11	software	Trigger		No	
zerod41	true	true	[]	float64	Spectrum	<mov>	ct41	ct41				No	
twod11	true	true	[1024, 1024]	float64	Image	<idx>|<idx>	twod11	twod11	software	Trigger		No	
ioveri0	true	true	[]	float64	Spectrum	<mov>	ct41	ct41				No	
unit_test/short_scalar	true	true	[]	short	Spectrum	<mov>						No	

Fig. 3: Measurement group tab of the expconf widget with the *mntgrp* configuration.

Experimental channel configuration

In the measurement group table the user can modify the following parameters of a given channel or its controller:

- enabled - include or exclude (True or False) the channel in the acquisition process.
- output - whether the channel acquisition results should be printed, for example, by the output recorder during the scan. Can be either True or False.
- shape - shape of the data
- data type - type of the data
- plot type - select the online scan plot type for the channel. Can have one of the following values: - No - no plot - Spectrum - suitable for scalar values - Image - suitable for spectrum values
- plot axes - select the abscissa (x axis) of the plot. Can be either - <idx> - scan index (point number) - <mov> - master moveable (in case of a2scan - the first motor) used in the scan - any of the scalar experimental channels used in the measurement group
- timer - channel to be used as timer. Timer controls the acqusition in terms of the integration time. Applies on the controller level.
- monitor - channel to be used as monitor. Monitor controls the acquisition in terms of the monitor counts. Applies on the controller level.
- synchronizer - the element that will synchronize the channel's acquisition. Can be either a *Trigger/Gate* element or the *software* synchronizer. Configurable only for the timerable controllers. Applies on the controller level.
- synchronization - the synchronization type. Can be either *Trigger* or *Gate*. Configurable only for the timerable controllers. Applies on the controller level.

- conditioning - expression to evaluate on the data before displaying it
- normalization - normalization mode for the data
- nexus path - location of the data of this channel withing the NeXus tree

Sardana Editor's interface

Contents

Todo: Sardana Editor documentation to be written

Scans

Perhaps the most used type of macro is the scan macros. In general terms, we call *scan* to a macro that moves one or more *motors* and acquires data along the path of the motor(s).

Note: Sardana provides a *Scan Framework* for developing scan macros so that the scan macros behave in a consistent way. Unless otherwise specified, the following discussion applies to scan macros based on such framework.

The various scan macros mostly differ in how many motors are moved and the definition of their paths.

Typically, the selection of which data is going to be acquired depends on the active *measurement group* and is *not* fixed by the macro itself (although there is no limitation in this sense).

Depending on whether the motors are stopped before acquiring the data or not, we can classify the scan macros in *step* scans or *continuous* scans, respectively.

Step scans

In a step scan, the motors are moved to given points, and once they reach each point they stop. Then, one or more channels are acquired for a certain amount of time, and only when the data acquisition is finished, the motors proceed to the next point.

In this way, the position associated to a data readout is well known and does not change during the acquisition time.

Some examples of step scan macros are: `ascan`, `a2scan`, ... `dscan`, `d2scan`, ... `mesh`.

Continuous scans

In a continuous scan, the motors are not stopped for acquisition, which therefore takes place while the motors are moving. The most common reason for using this type of scan is optimizing the acquisition time by not having to wait for motors to accelerate and decelerate between acquisitions.

The continuous scans introduce some constraints and issues that should be considered.

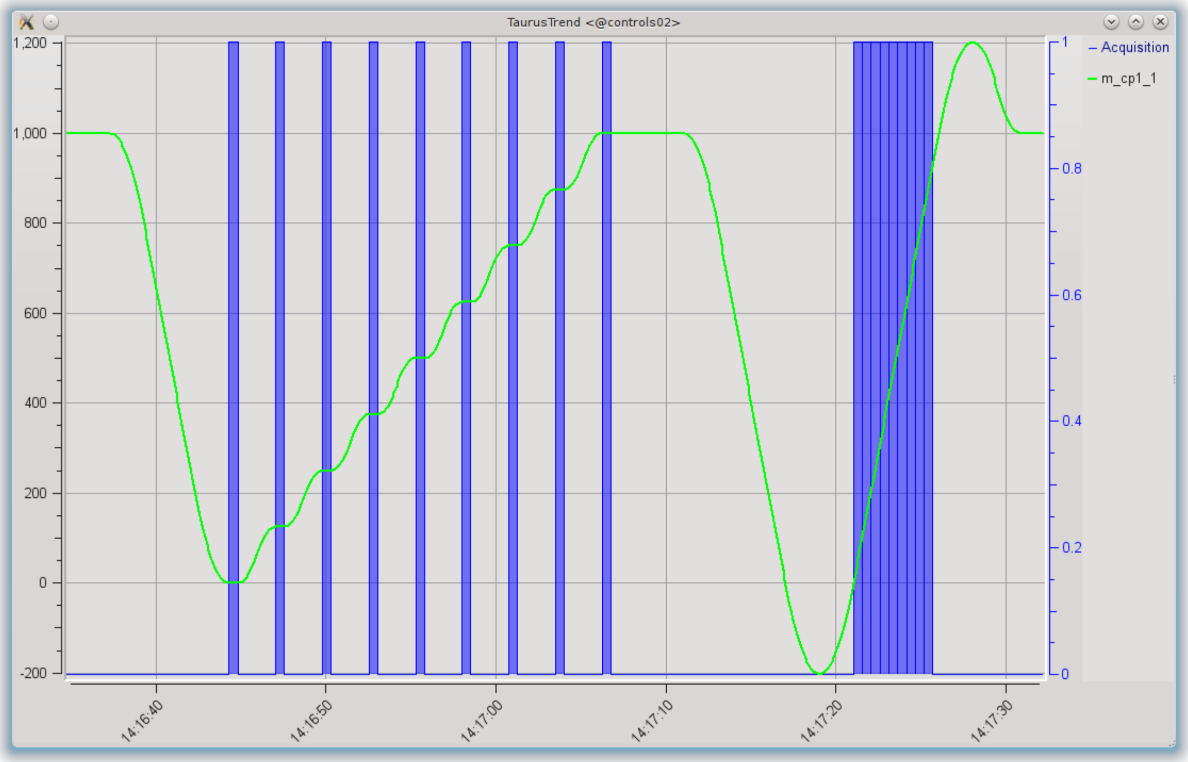


Fig. 4: Trend plot showing a step scan (`ascan` *m_cp1_1 0 1000 8 .5*) followed by a continuous scan (`ascanc` *m_cp1_1 0 1000 .5*). The line corresponds to the motor position and the blue shaded areas correspond to the intervals in which the data acquisition took place.

1. If a continuous scan involves moving more than one motor simultaneously (as it is done, e.g. in `a2scan`), then the movements of the motors should be synchronized so that they all start their path at the same time and finish it at the same time.
2. If motors do not maintain a constant velocity along the path of their movement, the trajectories followed when using more than one motor may not be linear.
3. While in step scans it is possible to scan two pseudo-motors that access the same physical motors (e.g. the *gap* and *offset* of a slit, being both pseudo-motors accessing the same physical motors attached to each blade of the slit), in a continuous scan the motions cannot be decoupled in a synchronized way.
4. Backslash correction is incompatible with continuous scans, so you should keep in mind that continuous scans should only be done in the backslash-free direction of the motor (typically, by convention the positive one for a physical motor).

In order to address the first two issues, the *scan framework* attempts the following:

- If the motors support changing their velocity, Sardana will adjust the velocities of the motors so that they all start and finish the required path simultaneously. For motors that specify a range of allowed velocities, this range will be used (for motors that do not specify a maximum allowed velocity, the current "top velocity" will be assumed to be the maximum)
- For motors that can maintain a constant velocity after an acceleration phase (this is the case for most physical motors), Sardana will transparently extend the user-given path both at the beginning and the end in order to allow for the motors to move at constant velocity along all the user defined path (i.e., the motors are allowed time and room to accelerate before reaching the start of the path and to decelerate after the end of the nominal path selected by the user)

These two actions can be seen in the following plot of the positions of the two motors involved in a `a2scanc`.

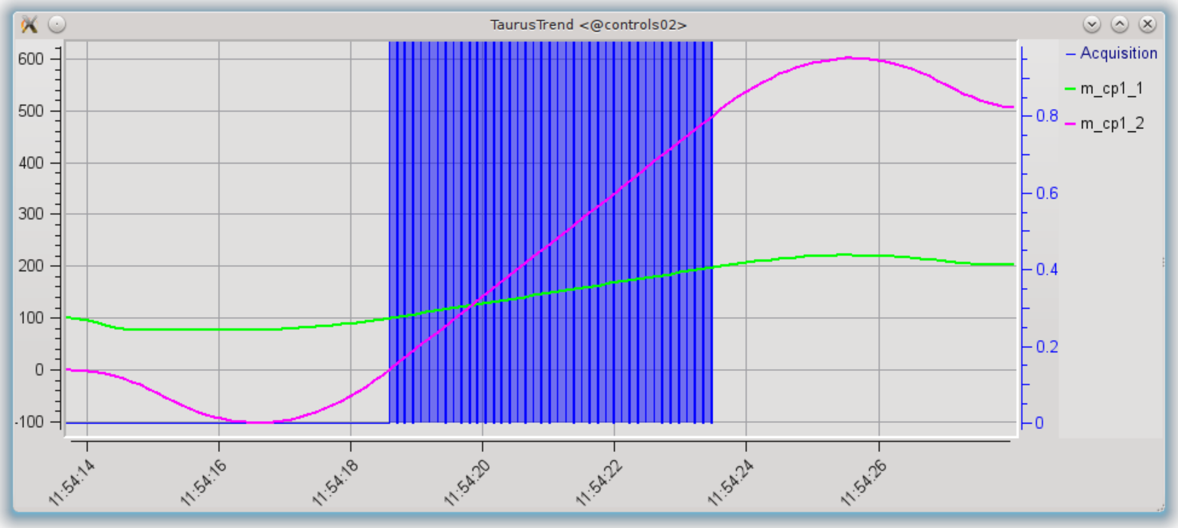


Fig. 5: Trend plot showing a two-motor continuous scan (`a2scanc` *m_cp1_1 100 200 m_cp1_2 0 500 .1*). The lines correspond to the motor positions and the blue shaded areas correspond to the intervals in which the data acquisition took place.

Both motors are capable of same velocity and acceleration, but since the required scan path for m_cp1_1 is shorter than that for m_cp1_2, its top velocity has been adjusted (gentler slope for m_cp1_1) so that both motors go through the user-requested start and stop positions simultaneously.

The same figure also shows how the paths for both motors have been automatically (and transparently, for the user) extended to guarantee that the user defined path is followed at constant velocity and that the data acquisition takes place also while the motors are running at constant velocity.

The synchronization of movement and acquisition can be done via hardware or via software. Currently Sardana provides two different interfaces for continuous scans. They can be easily differentiated by the scan name suffix:

- *c* - allows only software synchronization
- *ct* - allows both software and hardware synchronization (introduced with SEP6[73])

In the *c* type of scans, in order to optimize the acquisition time, Sardana attempts to perform as many acquisitions as allowed during the scan time. Due to the uncertainty in the delay times involved, it is not possible to know beforehand how many acquisitions will be completed. In other words, the number of acquired points along a continuous scan is not fixed (but it is guaranteed to be as large as possible). Some examples of continuous scan macros are: `ascanc`, `a2scanc`, ... `dscanc`, `d2scanc`, ... `meshc`.

In the *ct* type of scans, Sardana perform the exact number of acquisitions selected by the user by the means of hardware or software synchronization configurable on the *measurement group* level. The software synchronized channels may not follow the synchronization pace and some acquisitions may need to be skipped. In order to mitigate this risk an extra latency time can be spend in between the scan points. Another possibility is to enable data interpolation in order to fill the gaps in the scan records. Some examples of continuous scan macros are: `ascanct`, `a2scanct`, ... `dscanct`, `d2scanct`, ... At the time of writing the *ct* types of continuous scans still do not support acquiring neither of: *1D*, *2D*, *Pseudo Counter* nor external attributes e.g. Tango[74] however their support is planned in the near future.

Note: The creation of two different types of continuous scans is just the result of the iterative development of the *Scan Framework*. Ideally they will merge into one based on the *ct* approach. This process may require backwards incompatible changes (up to and including removal of the affected scan macros) if deemed necessary by the core developers.

Configuration

Scans are highly configurable using the environment variables (on how to use environment variables see environment related macros in *Standard macro catalog*).

Following variables are supported:

ApplyExtrapolation Enable/disable the extrapolation method to fill the missing parts of the very first scan records in case the software synchronized acquisition could not follow the pace. Can be used only with the continuous acquisition macros e.g. *ct* type of continuous scans or timescan. Its value is of boolean type.

> **Note:** The ApplyExtrapolation environment variable has been included in Sardana on a provisional basis. Backwards incompatible changes (up to and including removal of this variable) may occur if deemed necessary by the core developers.

ApplyInterpolation Enable/disable the zero order hold[75] a.k.a. "constant interpolation" method to fill the missing parts of the scan records in case the software synchronized acquisition could not follow

[73] http://www.sardana-controls.org/sep/?SEP6.md
[74] http://www.tango-controls.org
[75] https://en.wikipedia.org/wiki/Zero-order_hold

the pace. Can be used only with the continuous acquisition macros *ct* type of continuous scans or timescan. Its value is of boolean type.

Note: The ApplyInterpolation environment variable has been included in Sardana on a provisional basis with SEP6[76]. Backwards incompatible changes (up to and including removal of this variable) may occur if deemed necessary by the core developers.

DirectoryMap In case that the server and the client do not run on the same host, the scan data may be easily shared between them using the NFS. Since some of the tools e.g. showscan rely on the scan data file the DirectoryMap may help in overcoming the shared directory naming issues between the hosts.

Its value is a dictionary with keys pointing to the server side directory and values to the client side directory/ies (string or list of strings).

ScanDir Its value is of string type and indicates an absolute path to the directory where scan data will be stored.

ScanFile Its value may be either of type string or of list of strings. In the second case data will be duplicated in multiple files (different file formats may be used). Recorder class is implicitly selected based on the file extension. For example "myexperiment.spec" will by default store data in SPEC compatible format (see more about the extension to recorder map in *Writing recorders*).

ScanRecorder Its value may be either of type string or of list of strings. If ScanRecorder variable is defined, it explicitly indicates which recorder class should be used and for which file defined by ScanFile (based on the order).

Example 1:

```
ScanFile = myexperiment.spec
ScanRecorder = FIO_FileRecorder
```

FIO_FileRecorder will write myexperiment.spec file.

Example 2:

```
ScanFile = myexperiment.spec, myexperiment.h5
ScanRecorder = FIO_FileRecorder
```

FIO_FileRecorder will write myexperiment.spec file and NXscan_FileRecorder will write the myexpriment.h5. The selection of the second recorder is based on the extension.

SharedMemory Its value is of string type and it indicates which shared memory recorder should be used during the scan e.g. "sps" will use SPSRecorder (sps Python module must be installed on the PC where the MacroServer runs).

See also:

For more information about the implementation details of the scan macros in Sardana, see *scan framework*

Diffractometer

Sardana implements a full control of different types of diffractometers. The implementation is completely done inside a controller of the type PseudoMotor. It requires the hkl library developed by Frederic Picca and available as Debian package (https://packages.debian.org/source/sid/science/hkl). The use of the library

[76] http://www.sardana-controls.org/sep/?SEP6.md

is exclusively done inside of the controller code, so the hkl binding is only required if a diffractometer controller is going to be created.

Note: To use the HklPseudoMotorController you need to install the following Debian packages: `libhkl5` and `gir1.2-hkl-5.0`.

The Tango device created inside of the Pool for the diffractometer controller contains all the commands/attributes for setting the diffractometer, the movements are done using the real (in real space) and the pseudo (in reciprocal space) motor devices associated with this controller.

The diffractometer Sardana controller library is called HklPseudoMotorController. It contains several controller classes depending on the number of real and pseudo motors needed for each kind of diffractometer. All of them are based on a basic diffractometer class where the real implementation is done. Each class cover several diffractometer geometries. A class property, DiffractometerType, is created for setting the corresponding geometry. The class properties appear as Tango Device Properties of the device corresponding to this controller in the Pool. The value of the class properties has to be assigned when the instance of the diffractometer controller is generated in the Pool. It is fixed for each diffractometer controller, it can not be changed during running time.

Up to now the following diffractometer classes are implemented:

```
Diffrac6C (possible DiffractometerType values: "PETRA3 P09 EH2")
- motor roles: mu, omega, chi, phi, delta, gamma
- pseudomotor roles: h, k, l

DiffracE6C (possible DiffractometerType values: "E6C", "SOLEIL SIXS MED2+2")
- motor roles: mu, omega, chi, phi, gamma, delta
- pseudomotor roles: h, k, l, psi, q, alpha, qper, qpar

 DiffracE4C (possible DiffractometerType values: "E4CV", "E4CH", "SOLEIL MARS")
 - motor roles: omega, chi, phi, gamma, tth
 - pseudomotor roles: h, k, l, psi, q

 Diffrac4Cp23 (possible DiffractometerType values: "PETRA3 P23 4C")
 - motor roles: omega_t, mu, gamma, delta
 - pseudomotor roles: h, k, l, q, alpha, qper, qpar, tth2, alpha_tth2,
   incidence, emergence
```

In order to have the diffractometer in Sardana one has to create a PseudoMotor controller of any of the above classes, this will generate the controller device, for setting the diffractometer, and the pseudo motor devices, for the reciprocal space movements. The real motors have to be included as motors in Sardana, like the motors associated to any other PseudoMotor.

Example:

```
defctrl DiffracE6C e6cctrl mu=mot01 omega=mot02 chi=mot03 phi=mot04 \
  gamma=mot05 delta=mot06 h=e6ch k=e6ck l=e6cl psi=e6cpsi q=e6cq \
  alpha=e6calpha qper=e6cqper1 qpar=e6cqpar DiffractometerType E6C
```

The command above creates the devices corresponding to a diffractometer with E6C geometry, where `motXX` are the names of the Pool devices associated to each real motor (already existing Motors in Sardana), `e6cctrl` is an arbitrary name given to the controller, and `e6ch`, `e6ck` , `e6cl`, `e6cpsi`, `e6cq`, `e6calpha`, `e6cqper1` and `e6cqpar` are the arbitrary names given to the motors in reciprocal space (PseudoMotors in Sardana, created by this call).

The diffractometer controller device contains as attributes (since it is not possible add extra commands to the controllers) all the commands and settings for the diffractometer control. The hkl library needs the parameters to be set in a given order for being inizialized correctly, for that reason the attribute values are not stored in the tango database. The best practise is to save the parameters in a file and load it when startup. Example of a startup file:

```
Created at 2018-05-28 01:50

DiffractometerType E6C

Crystal    srru2o6

Wavelength 4.36871881607

A 5.97947283834 B 5.97062576634 C 17.0006259383
Alpha 90.0 Beta 90.0 Gamma 120.0

R0 0 0.0 0.0 3.0 0 1 0.0 23.983725 109.768125 286.38645 -1.52587888991e-08 45.344725
R1 1 1.0 0.0 4.0 0 1 0.0 43.03545 70.0 286.38645 -1.52587888991e-08 83.409275

Engine hkl

Mode constant_phi_vertical

PsiRef not available in current engine mode

AutoEnergyUpdate 0

U00 -1.092 U01 -0.250 U02 0.122
U10 0.410 U11 0.180 U12 0.348
U20 0.332 U21 1.176 U22 -0.027

Ux -94.4630713237 Uy 19.3202405308 Uz -162.577336525

SaveDirectory /home/p09user/crystals
```

The configuration files can be created by the diffractometer controller by calling the attribute SaveCrystal. They are loaded by the attribute LoadCrystal. If nothing is loaded the diffractometer will be inizialized with Sample set to 'default crystal', lattice, wavelenth and geometry set to the corresponding hkl library default values (according to the geometry) and engine set to 'hkl'.

The wavelength used for the diffractometer can be automatically updated setting the attribute AutoEnergyUpdate to 1 and the name of the Tango Device controlling the beam energy in the attribute EnergyDevice. The energy is read from the attribute Position of the EnergyDevice device, in eV, every time motor or pseudomotor positions are read.

Sardana provides *standard macros for controlling the diffractometer*. They need the name of the diffractometer controller device in the Pool to be set in the macroserver environment variable DiffracDevice. If the Psi angle (azimuth) will be used, the environment variable Psi has to be set, with the name corresponding PseudoMotor Pool device.

Running the macro `sar_demo_hkl` creates a simulated diffractometer in Sardana.

The diffractometer can be controlled from spock used the implemented dedicated macros, as described in the *catalog of macros*.

Macro Hooks

A hook is an extra code that can be run at certain points of a macro execution. These points are predefined for each hookable macro and passed via a "hints" mechanism. The hint tells the macro how and when to run the attached hook. Hooks allow the customization of already existing macros and can be added using three different ways:

- General Hooks
- *Sequencer Hooks*
- *Programmatic Hooks*

All available macros can be used as a hook.

General Hooks

The general hooks were implemented in Sardana after the programmatic hooks. The motivation for this implementation was to allow the customization of the scan macros without having to redefine them. The general hooks apply to all hookable macros and allow the definition of new hints. They can be controlled using dedicated macros: `lsgh`, `defgh` and `udefgh`. For each hook position, hint, several hooks can be run, they will be run in the order they were added. The same hook can be run several times in the same position if it's added several times.

Examples:

- Check motor limits in step scans (pre-scan)
- Prepare 2D detectors: create directories, set save directory, file name, file index (pre-scan)
- Set attenuators, set undulator, check shutter, check current (pre-scan)
- Restore changes (post-scan)

Standard macro catalog

motion related macros

- `wa`
- `wm`
- `pwa`
- `pwm`
- `set_lim`
- `set_lm`
- `set_pos`
- `mv`
- `umv`
- `mvr`
- `umvr`
- `tw`
- `lsm`
- `lspm`

counting macros

- `ct`
- `uct`
- `settimer`
- `lsexp`

- *lsmeas*
- *lsct*
- *ls0d*
- *ls1d*
- *ls2d*
- *lspc*

diffractometer related macros

- *addreflection*
- *affine*
- *br*
- *ca*
- *caa*
- *ci*
- *computeub*
- *freeze*
- *getmode*
- *hklscan*
- *hscan*
- *kscan*
- *latticecal*
- *loadcrystal*
- *lscan*
- *newcrystal*
- *or0*
- *or1*
- *orswap*
- *pa*
- *savecrystal*
- *setaz*
- *setlat*
- *setmode*
- *setor0*
- *setor1*
- *setorn*
- *th2th*
- *ubr*
- *wh*

environment related macros

- *lsenv*
- *senv*
- *usenv*
- *dumpenv*

list related macros

- *lsenv*
- *lsa*

- *lsm*
- *lspm*
- *lsexp*
- *lsior*
- *lsmeas*
- *lsct*
- *ls0d*
- *ls1d*
- *ls2d*
- *lspc*
- *lsctrl*
- *lsi*
- *lsctrllib*
- *lsa*
- *lsmac*
- *lsmaclib*
- *lsgh*

measurement configuration macros

- *defmeas*
- *udefmeas*

general hooks macros

- *lsgh*
- *defgh*
- *udefgh*

advanced element manipulation macros

- *defelem*
- *udefelem*
- `renameelem`
- *defctrl*
- *udefctrl*
- *prdef*

reload code macros

- *relmac*
- *relmaclib*
- *addmaclib*
- *rellib*
- *relctrlcls*
- *relctrllib*
- *addctrllib*

scan macros

- *ascan*
- *a2scan*
- *a3scan*
- *a4scan*
- *amultiscan*
- *dscan*
- *d2scan*
- *d3scan*
- *d4scan*
- *dmultiscan*
- *mesh*
- *fscan*
- *scanhist*
- *ascanc*
- *a2scanc*
- *a3scanc*
- *a4scanc*
- *dscanc*
- *d2scanc*
- *d3scanc*
- *d4scanc*
- *meshc*
- *ascanct*
- *a2scanct*
- *a3scanct*
- *a4scanct*
- dscanct
- d2scanct
- d3scanct
- d4scanct

Screenshots

Here you will find a host of example figures.

Sardana oriented graphical user interfaces

Graphical user interface screen shots

Todo: The FAQ is work-in-progress. Many answers need polishing and mostly links need to be added

[77] http://www.cells.es/
[78] http://www.cells.es/
[79] http://www.cells.es/
[80] http://www.cells.es/
[81] http://www.cells.es/
[82] http://www.cells.es/
[83] http://www.cells.es/
[84] http://www.cells.es/
[85] http://www.cells.es/

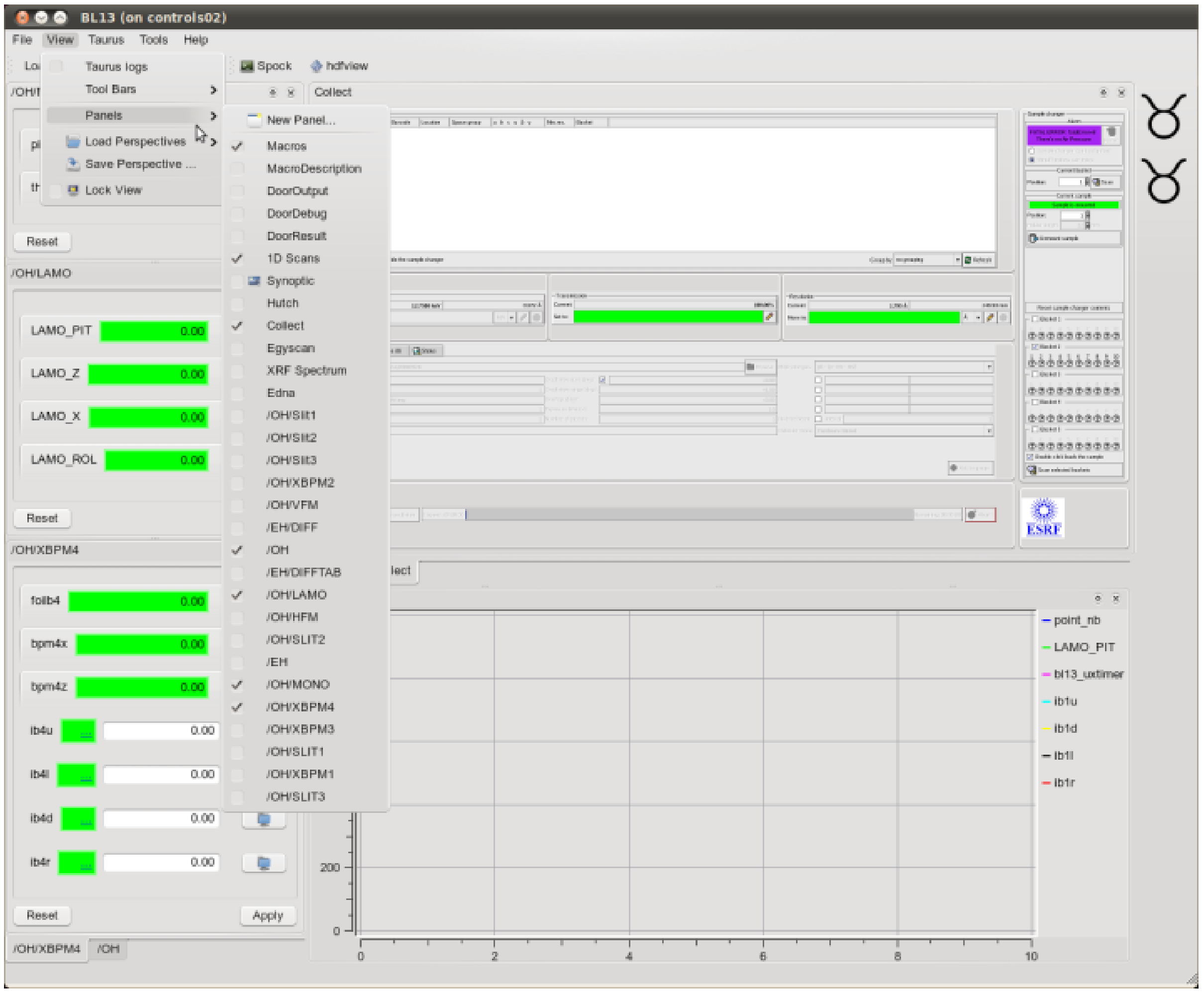
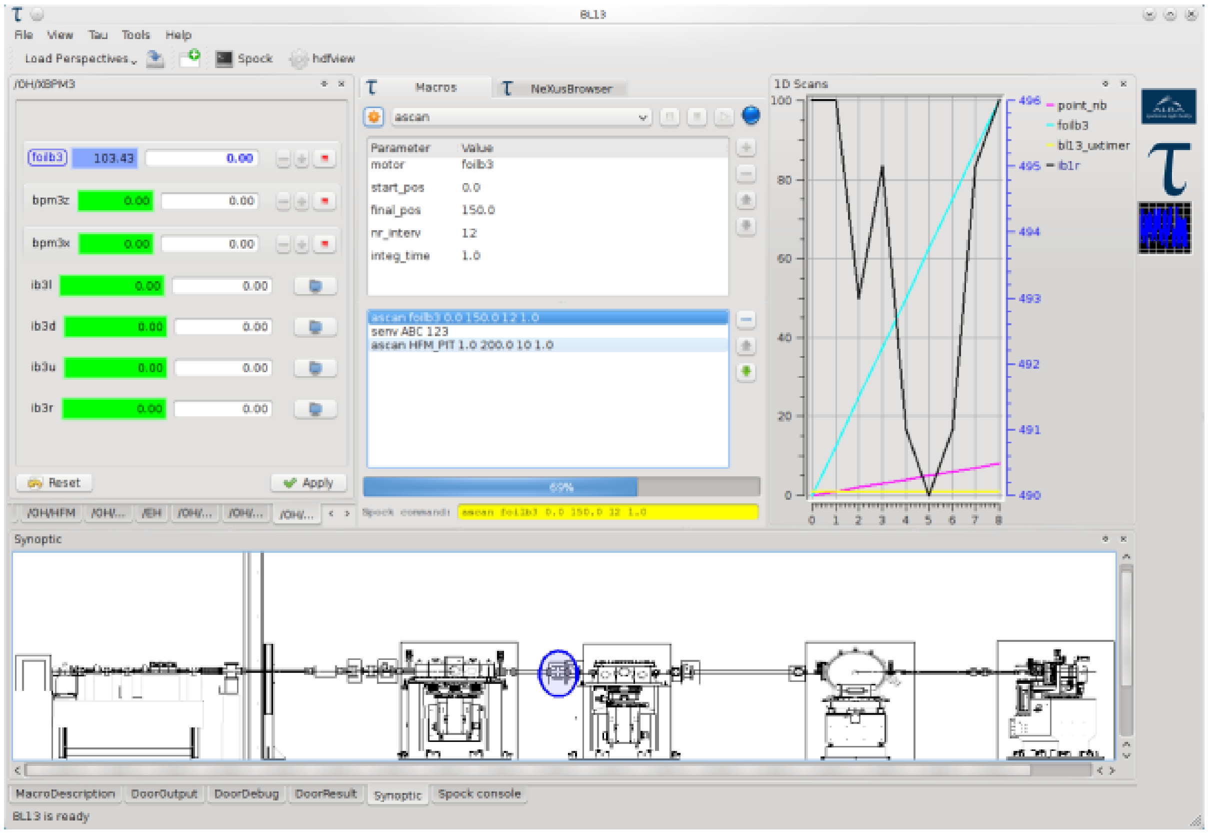


Fig. 6: TaurusGUI at work.

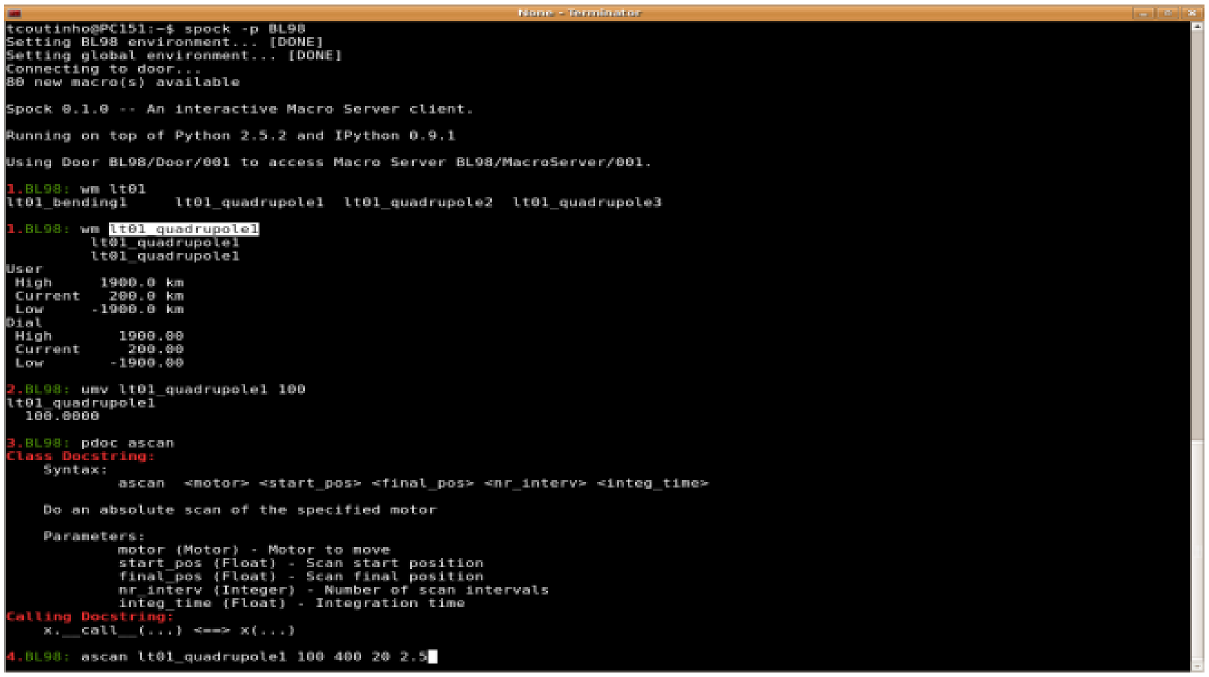


Parameter	Value
motor	foilb3
start_pos	0.0
final_pos	150.0
nr_interv	12
integ_time	1.0

```
ascan foilb3 0.0 150.0 12 1.0
senv ABC 123
ascan HFM_PIT 1.0 200.0 10 1.0
```

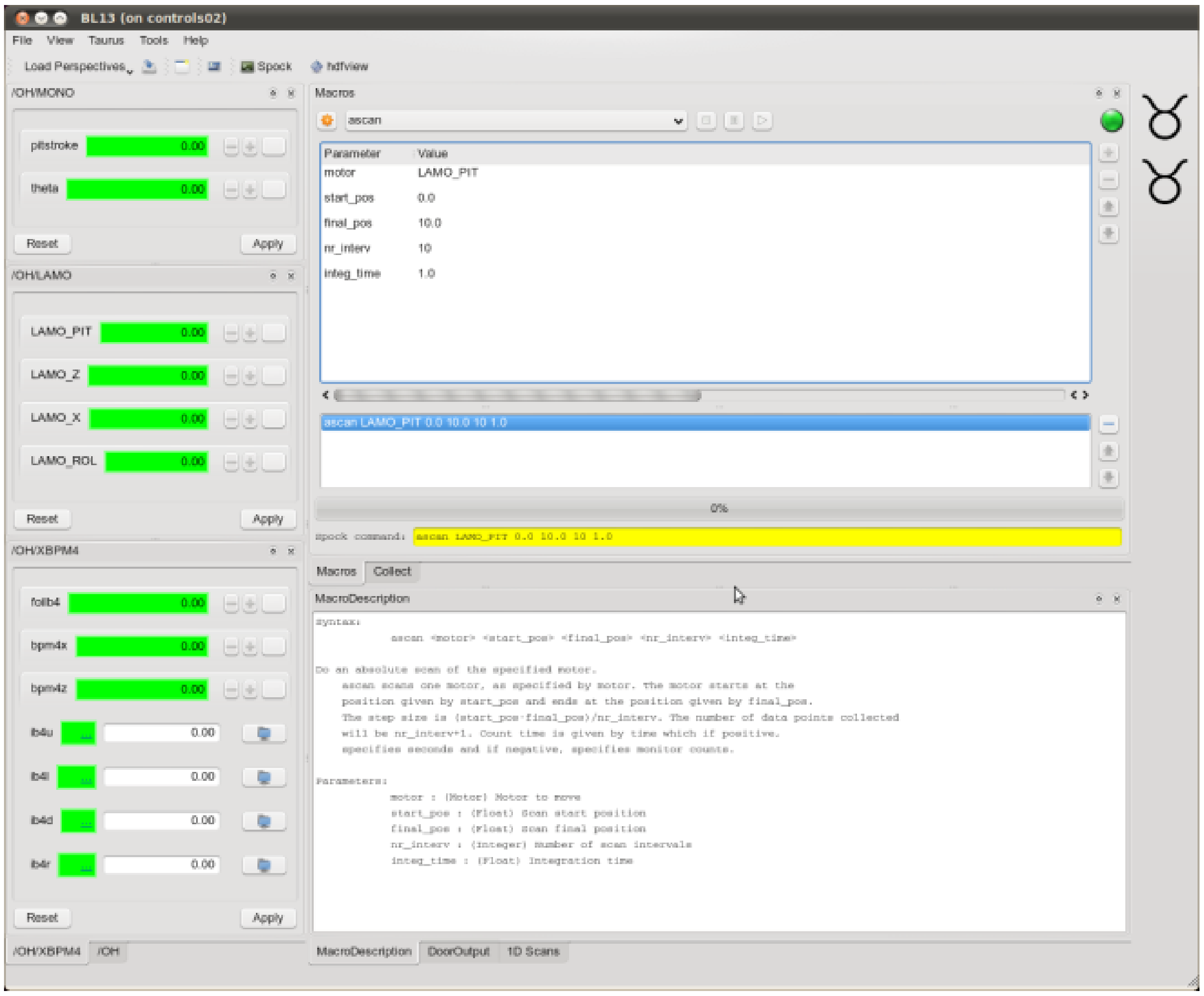
Spock command: `ascan foilb3 0.0 150.0 12 1.0`

Fig. 7: TaurusGUI with synoptic and macro widget



```
tcoutinho@PC151:~$ spock -p BL98
Setting BL98 environment... [DONE]
Setting global environment... [DONE]
Connecting to door...
80 new macro(s) available

Spock 0.1.0 -- An interactive Macro Server client.

Running on top of Python 2.5.2 and IPython 0.9.1

Using Door BL98/Door/001 to access Macro Server BL98/MacroServer/001.

1.BL98: wm lt01
lt01_bending1     lt01_quadrupole1  lt01_quadrupole2  lt01_quadrupole3

1.BL98: wm lt01_quadrupole1
         lt01_quadrupole1
         lt01_quadrupole1
User
 High      1900.0 km
 Current    200.0 km
 Low      -1900.0 km
Dial
 High        1900.00
 Current      200.00
 Low        -1900.00

2.BL98: umv lt01_quadrupole1 100
lt01_quadrupole1
  100.0000

3.BL98: pdoc ascan
Class Docstring:
    Syntax:
            ascan  <motor> <start_pos> <final_pos> <nr_interv> <integ_time>

    Do an absolute scan of the specified motor

    Parameters:
            motor (Motor) - Motor to move
            start_pos (Float) - Scan start position
            final_pos (Float) - Scan final position
            nr_interv (Integer) - Number of scan intervals
            integ_time (Float) - Integration time
Calling Docstring:
    x.__call__(...) <==> x(...)

4.BL98: ascan lt01_quadrupole1 100 400 20 2.5
```

Fig. 8: Spock console



Parameter	Value
motor	LAMO_PIT
start_pos	0.0
final_pos	10.0
nr_interv	10
integ_time	1.0

ascan LAMO_PIT 0.0 10.0 10 1.0

spock command: ascan LAMO_PIT 0.0 10.0 10 1.0

```
Syntax:
            ascan <motor> <start_pos> <final_pos> <nr_interv> <integ_time>

Do an absolute scan of the specified motor.
    ascan scans one motor, as specified by motor. The motor starts at the
    position given by start_pos and ends at the position given by final_pos.
    The step size is (start_pos-final_pos)/nr_interv. The number of data points collected
    will be nr_interv+1. Count time is given by time which if positive,
    specifies seconds and if negative, specifies monitor counts.

Parameters:
            motor : (Motor) Motor to move
            start_pos : (Float) Scan start position
            final_pos : (Float) Scan final position
            nr_interv : (Integer) Number of scan intervals
            integ_time : (Float) Integration time
```

Fig. 9: TaurusGUI with synoptic and macro panel

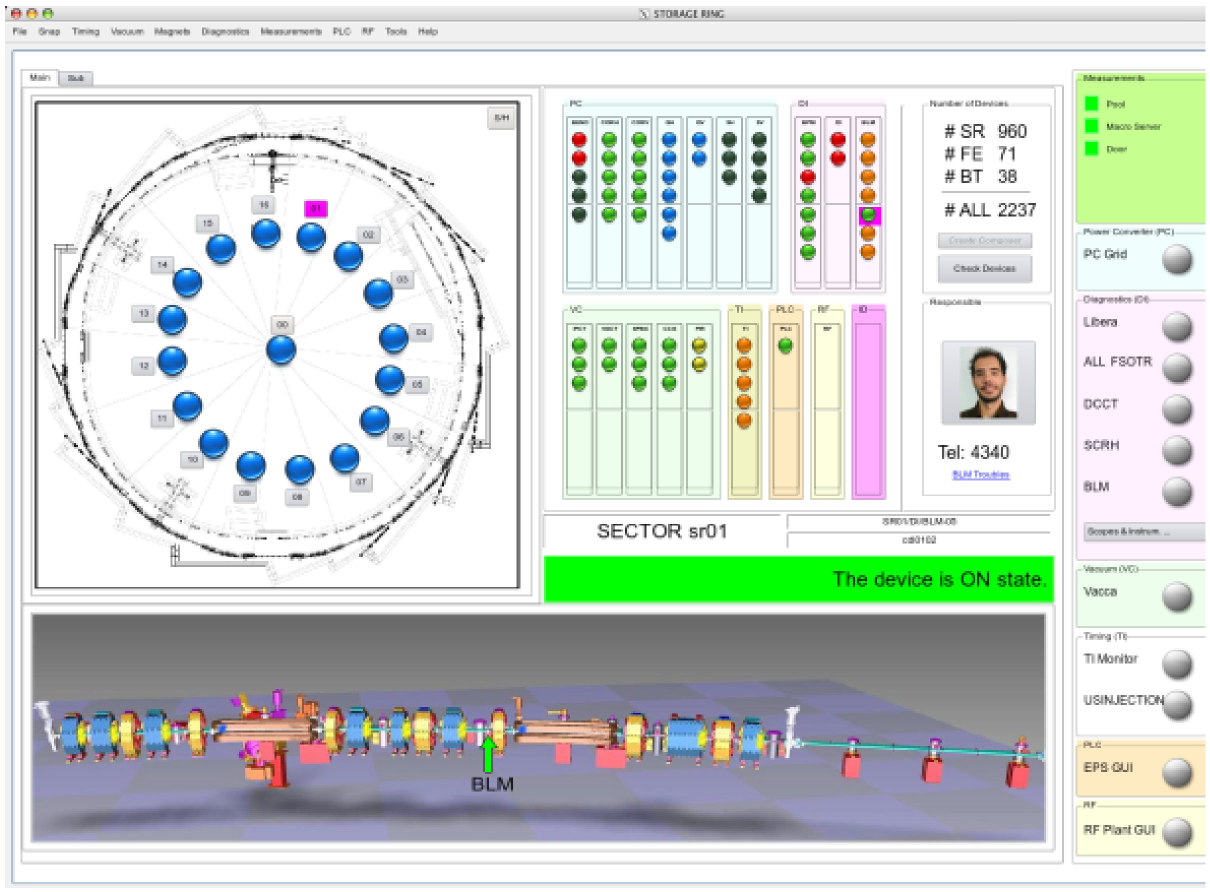


Fig. 10: ALBA[77]'s Storage ring GUI

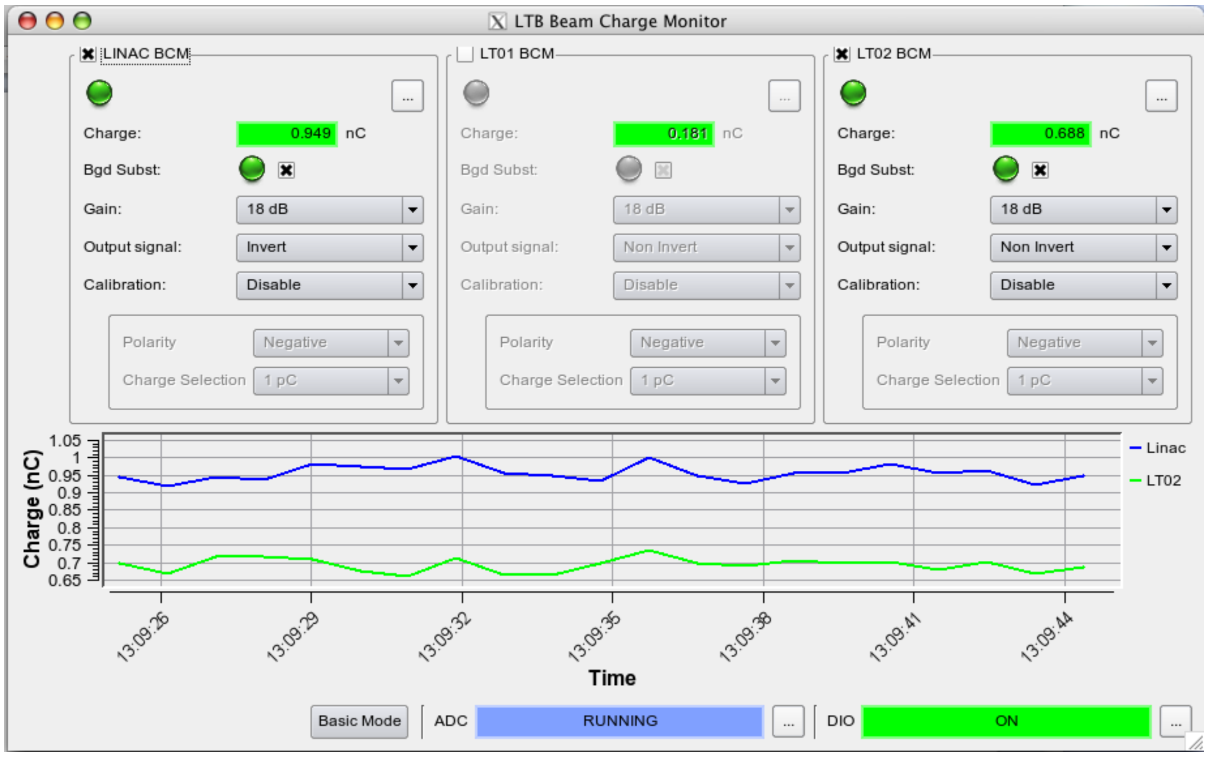


Fig. 11: ALBA[78]'s LINAC to booster beam charge monitor GUI

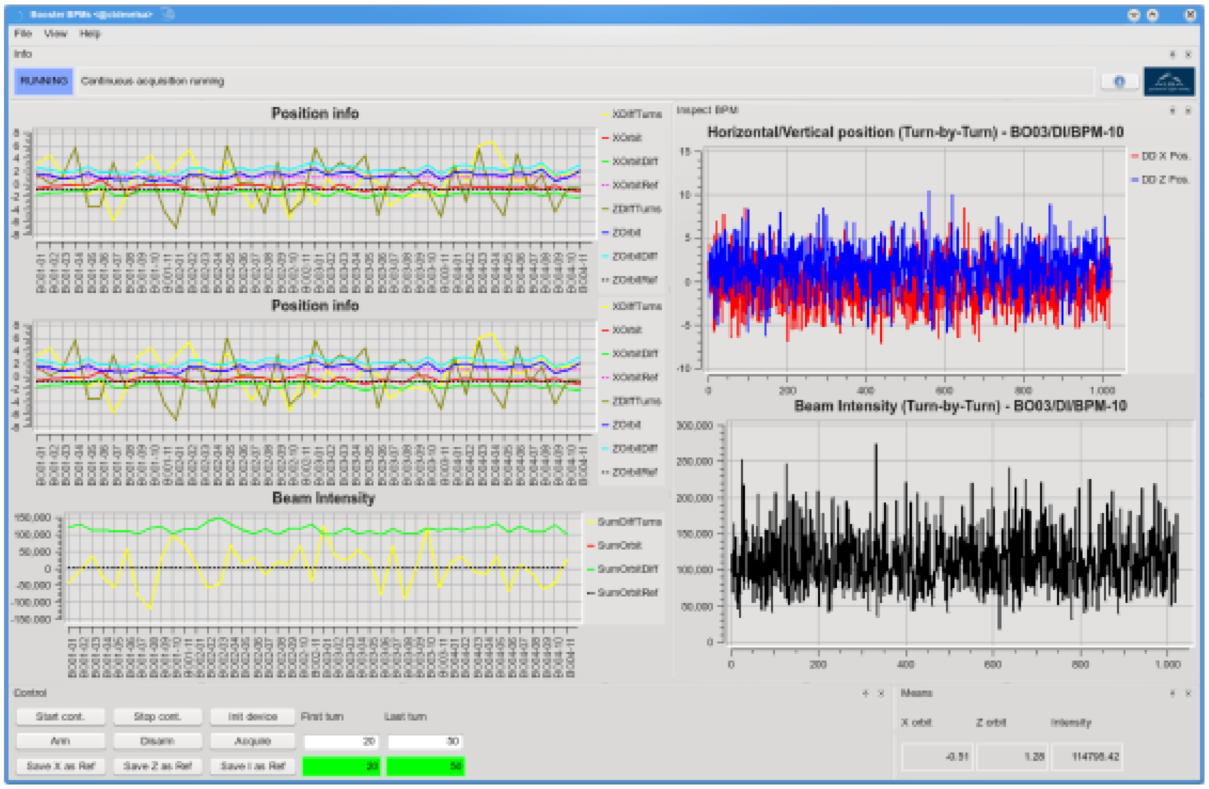


Fig. 12: ALBA[79]'s beam position monitor GUI

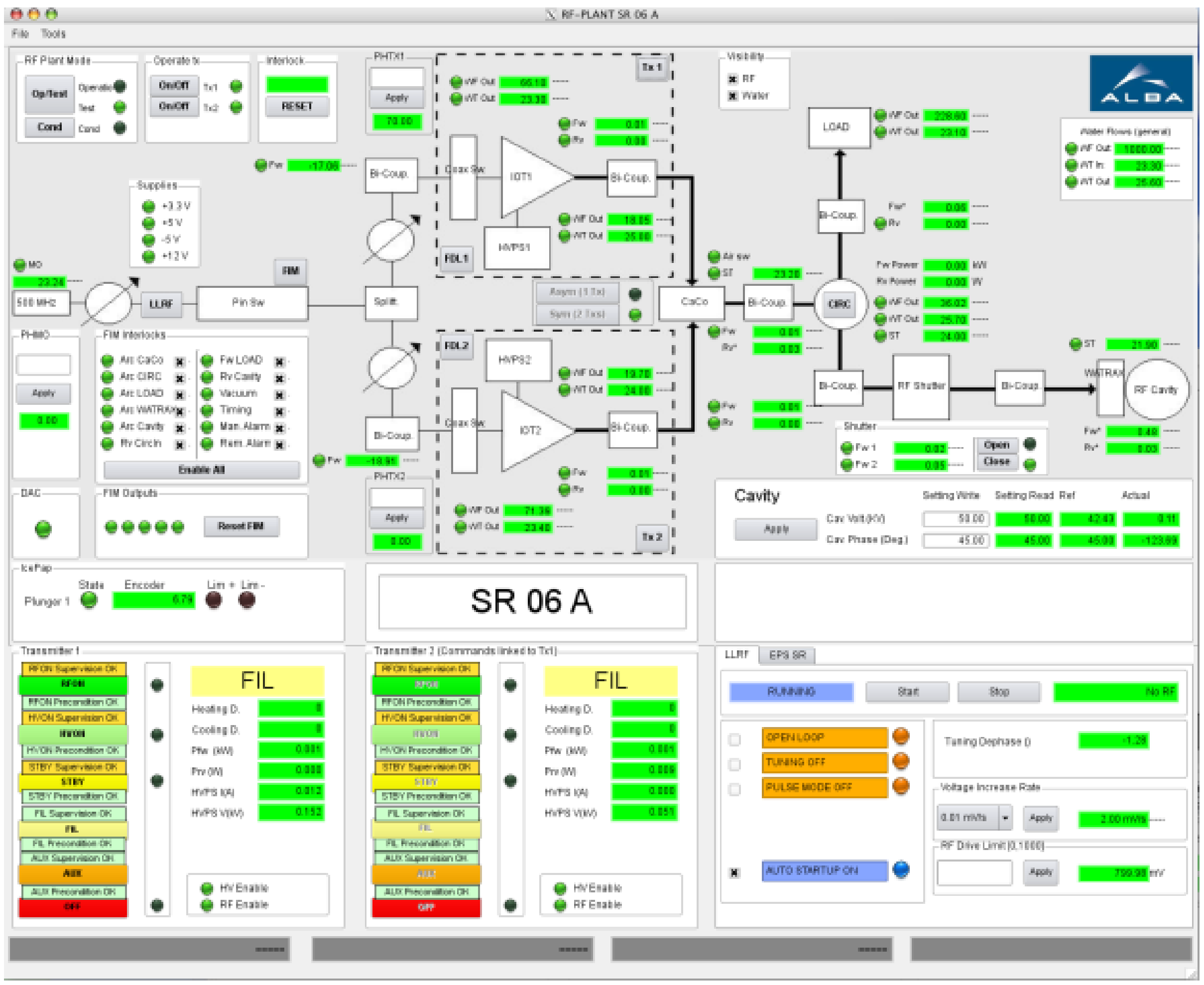
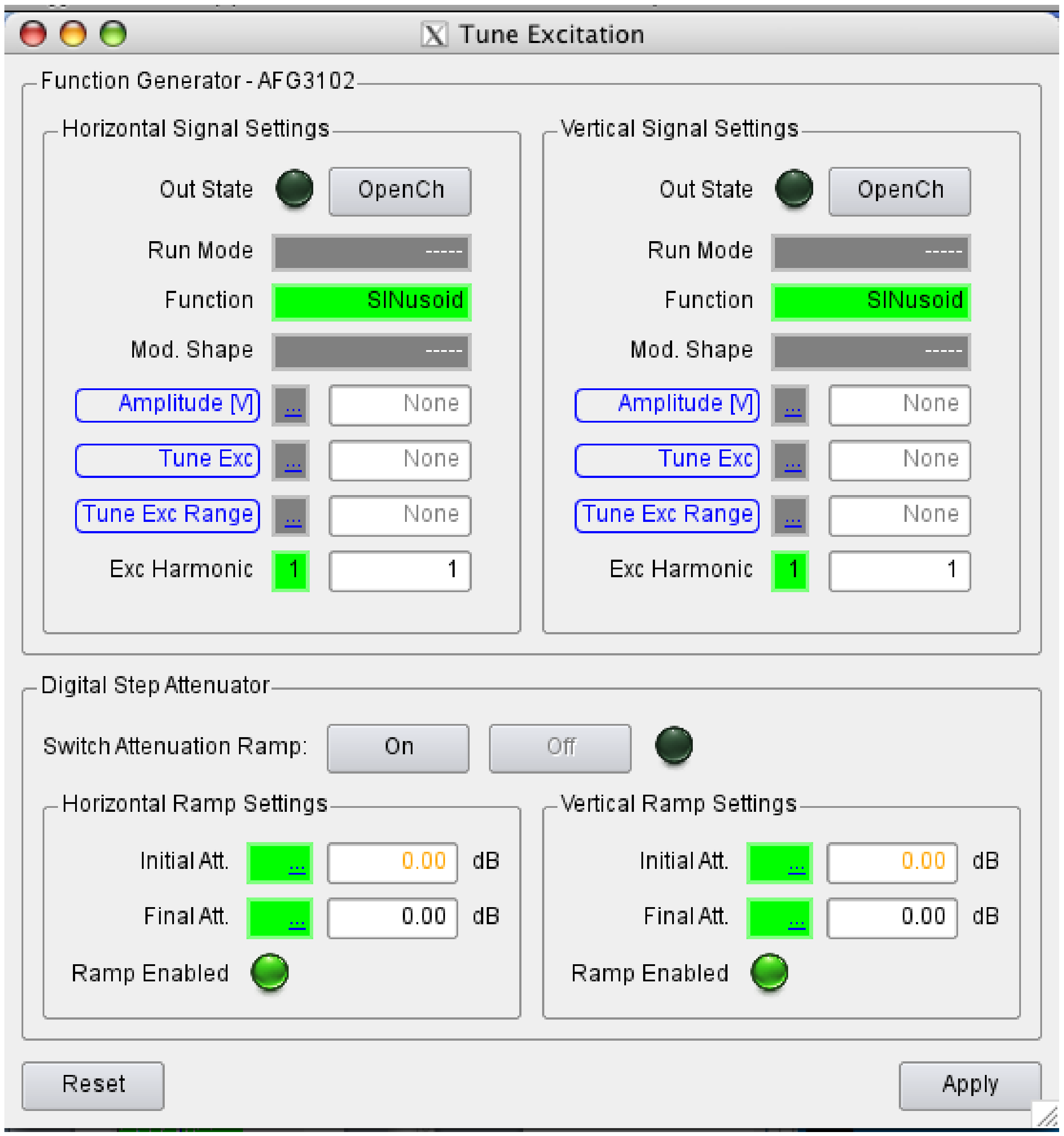


Fig. 13: ALBA[80]'s Radio frequency plant GUI


Tune Excitation

Function Generator - AFG3102

Horizontal Signal Settings

Out State OpenCh
Run Mode -----
Function SINusoid
Mod. Shape -----
Amplitude [V] ... None
Tune Exc ... None
Tune Exc Range ... None
Exc Harmonic 1 1

Vertical Signal Settings

Out State OpenCh
Run Mode -----
Function SINusoid
Mod. Shape -----
Amplitude [V] ... None
Tune Exc ... None
Tune Exc Range ... None
Exc Harmonic 1 1

Digital Step Attenuator

Switch Attenuation Ramp: On Off

Horizontal Ramp Settings

Initial Att. ... 0.00 dB
Final Att. ... 0.00 dB
Ramp Enabled

Vertical Ramp Settings

Initial Att. ... 0.00 dB
Final Att. ... 0.00 dB
Ramp Enabled

Reset Apply

Fig. 14: ALBA[81]'s tune excitation panel

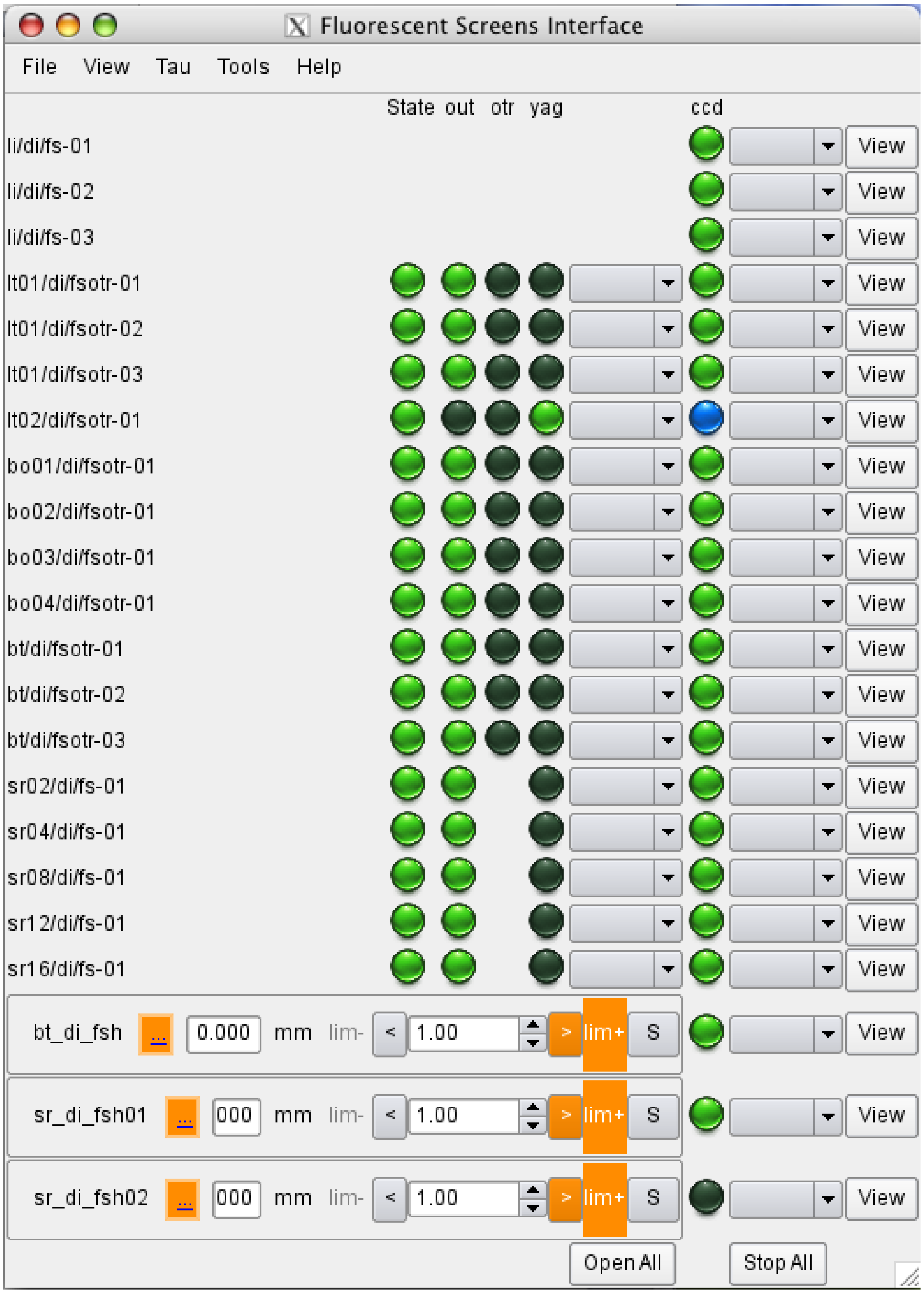


Fig. 15: ALBA[82]'s fluorescent screen main panel

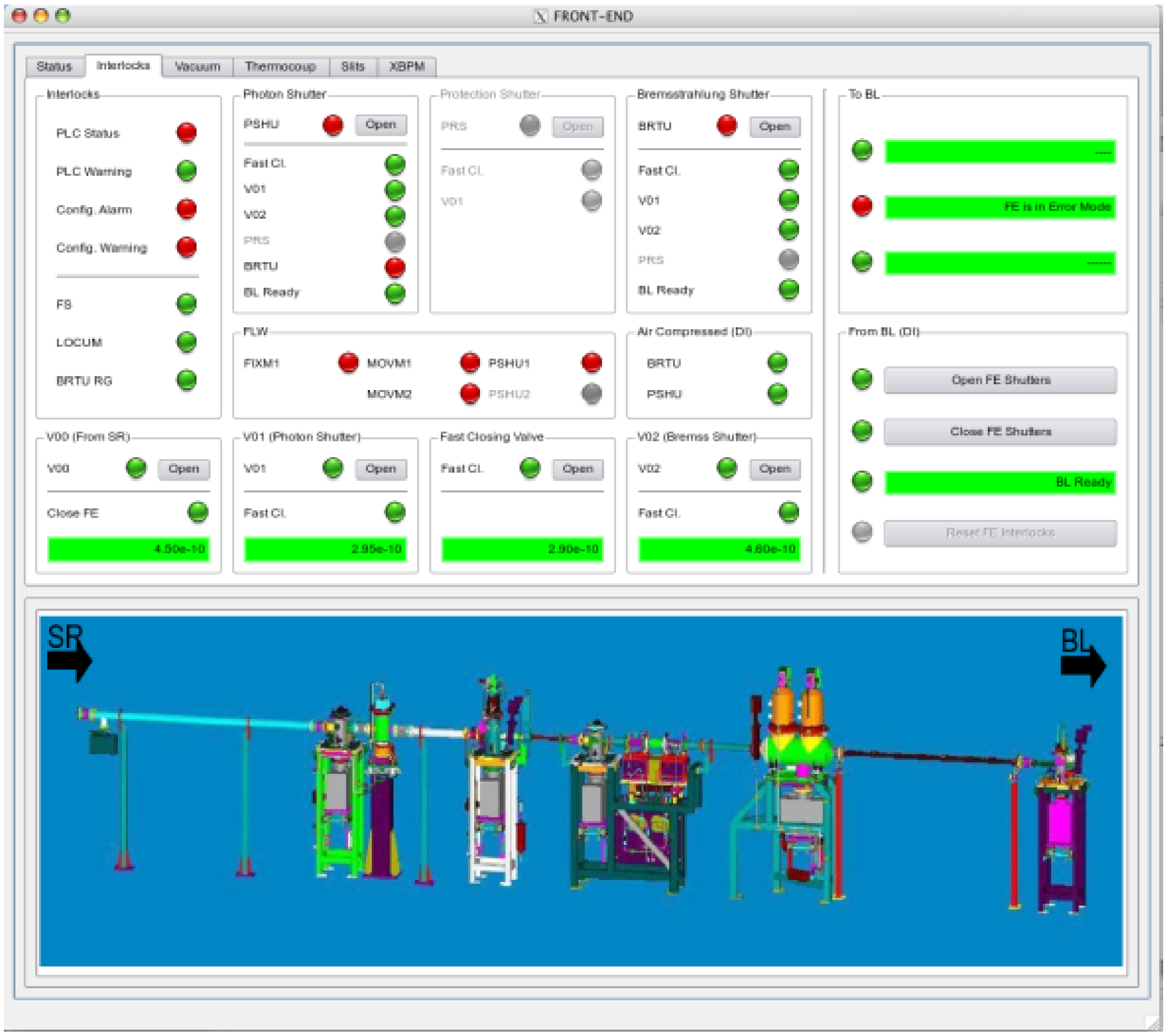
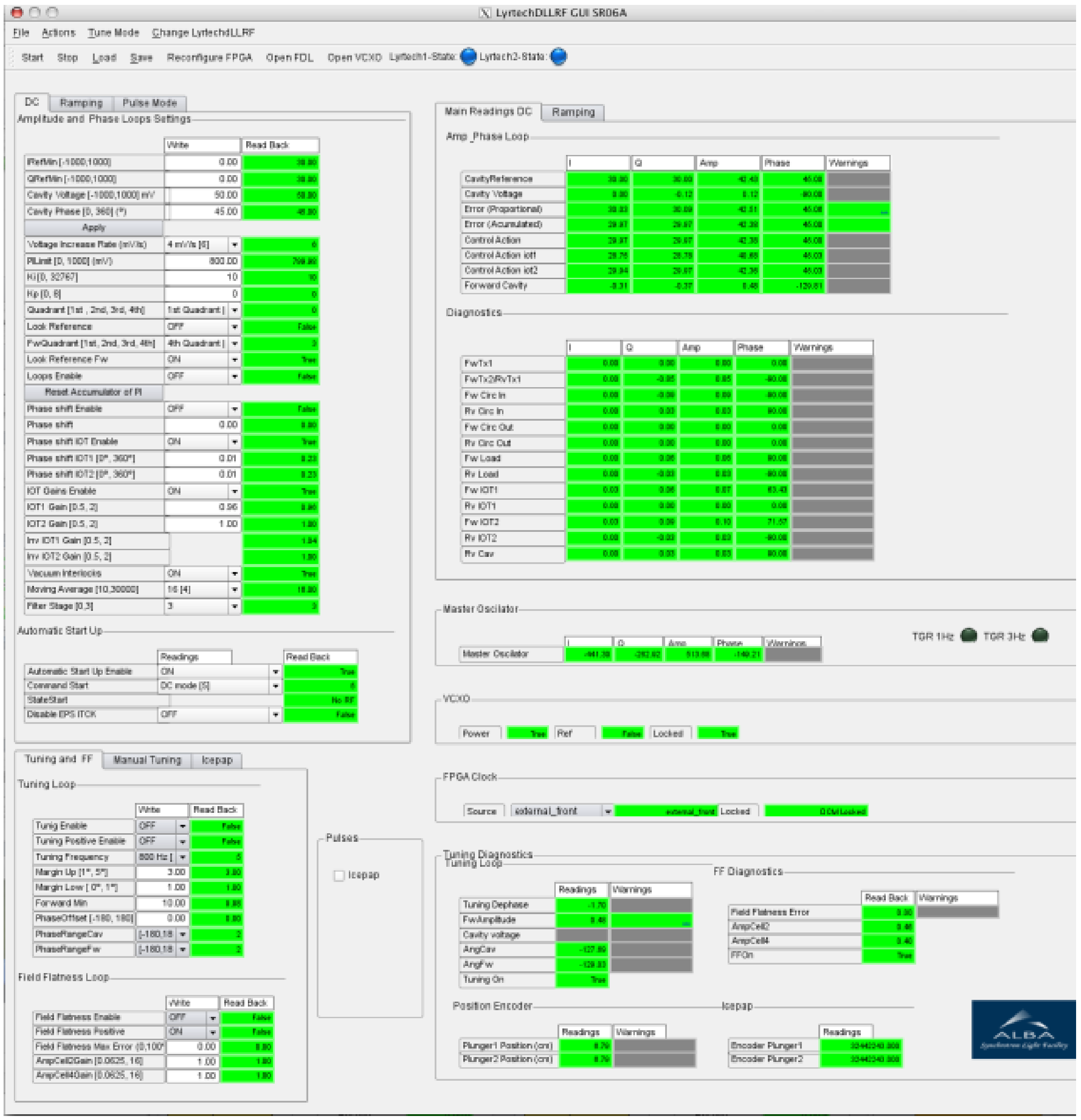


Fig. 16: ALBA[83]'s front end GUI



Amplitude and Phase Loops Settings

	Write	Read Back
IRefMin [-1000,1000]	0.00	30.00
QRefMin [-1000,1000]	0.00	30.00
Cavity Voltage [-1000,1000] mV	50.00	50.00
Cavity Phase [0, 360] (°)	45.00	45.00
Voltage Increase Rate (mV/s)	4 mV/s [6]	6
PILimit [0, 1000] (mV)	800.00	799.80
Ki [0, 32767]	10	10
Kp [0, 8]	0	0
Quadrant [1st , 2nd, 3rd, 4th]	1st Quadrant [	0
Look Reference	OFF	False
FwQuadrant [1st, 2nd, 3rd, 4th]	4th Quadrant [	3
Look Reference Fw	ON	True
Loops Enable	OFF	False
Phase shift Enable	OFF	False
Phase shift	0.00	0.00
Phase shift IOT Enable	ON	True
Phase shift IOT1 [0°, 360°]	0.01	0.23
Phase shift IOT2 [0°, 360°]	0.01	0.23
IOT Gains Enable	ON	True
IOT1 Gain [0.5, 2]	0.96	0.96
IOT2 Gain [0.5, 2]	1.00	1.00
Inv IOT1 Gain [0.5, 2]		1.04
Inv IOT2 Gain [0.5, 2]		1.00
Vacuum Interlocks	ON	True
Moving Average [10,30000]	16 [4]	16.00
Filter Stage [0,3]	3	3

Automatic Start Up

	Readings	Read Back
Automatic Start Up Enable	ON	True
Command Start	DC mode [5]	5
StateStart		No RF
Disable EPS ITCK	OFF	False

Tuning Loop

	Write	Read Back
Tunig Enable	OFF	False
Tuning Positive Enable	OFF	False
Tuning Frequency	800 Hz [	5
Margin Up [1°, 5°]	3.00	3.00
Margin Low [0°, 1°]	1.00	1.00
Forward Min	10.00	8.88
PhaseOffset [-180, 180]	0.00	0.00
PhaseRangeCav	[-180,18	2
PhaseRangeFw	[-180,18	2

Field Flatness Loop

	Write	Read Back
Field Flatness Enable	OFF	False
Field Flatness Positive	ON	False
Field Flatness Max Error (0,100°	0.00	0.00
AmpCell2Gain [0.0625, 16]	1.00	1.00
AmpCell4Gain [0.0625, 16]	1.00	1.00

Pulses

☐ Icepap

Amp_Phase Loop

	I	Q	Amp	Phase	Warnings
CavityReference	30.00	30.00	42.43	45.00	
Cavity Voltage	0.00	-0.12	0.12	-90.00	
Error (Proportional)	30.03	30.09	42.51	45.08	
Error (Acumulated)	29.97	29.97	42.38	45.08	
Control Action	29.97	29.97	42.38	45.08	
Control Action iot1	28.78	28.78	40.69	45.03	
Control Action iot2	29.94	29.97	42.38	45.03	
Forward Cavity	-0.31	-0.37	0.48	-129.81	

Diagnostics

	I	Q	Amp	Phase	Warnings
FwTx1	0.00	0.00	0.00	0.08	
FwTx2/RvTx1	0.00	-0.05	0.05	-90.08	
Fw Circ In	0.00	-0.09	0.09	-90.08	
Rv Circ In	0.00	0.03	0.03	90.08	
Fw Circ Out	0.00	0.00	0.00	0.08	
Rv Circ Out	0.00	0.00	0.00	0.08	
Fw Load	0.00	0.05	0.05	90.08	
Rv Load	0.00	-0.03	0.03	-90.08	
Fw IOT1	0.03	0.05	0.07	63.43	
Rv IOT1	0.00	0.00	0.00	0.08	
Fw IOT2	0.03	0.09	0.10	71.57	
Rv IOT2	0.00	-0.03	0.03	-90.08	
Rv Cav	0.00	0.03	0.03	90.08	

Master Oscilator

	I	Q	Amp	Phase	Warnings
Master Oscilator	-441.38	-282.82	513.08	-149.21	

TGR 1Hz ● TGR 3Hz ●

VCXO

Power	True	Ref	False	Locked	True

FPGA Clock

Source	external_front	external_front	Locked	DCM Locked

Tuning Diagnostics

Tuning Loop

	Readings	Warnings
Tuning Dephase	-1.70	
FwAmplitude	8.48	
Cavity voltage		
AngCav	-127.89	
AngFw	-129.83	
Tuning On	True	

FF Diagnostics

	Read Back	Warnings
Field Flatness Error	0.00	
AmpCell2	0.40	
AmpCell4	0.40	
FFOn	True	

Position Encoder

	Readings	Warnings
Plunger1 Position (cm)	8.79	
Plunger2 Position (cm)	8.79	

Icepap

	Readings
Encoder Plunger1	33442340.000
Encoder Plunger2	33442340.000

Fig. 17: ALBA[84]'s digital low level radio frequency GUI

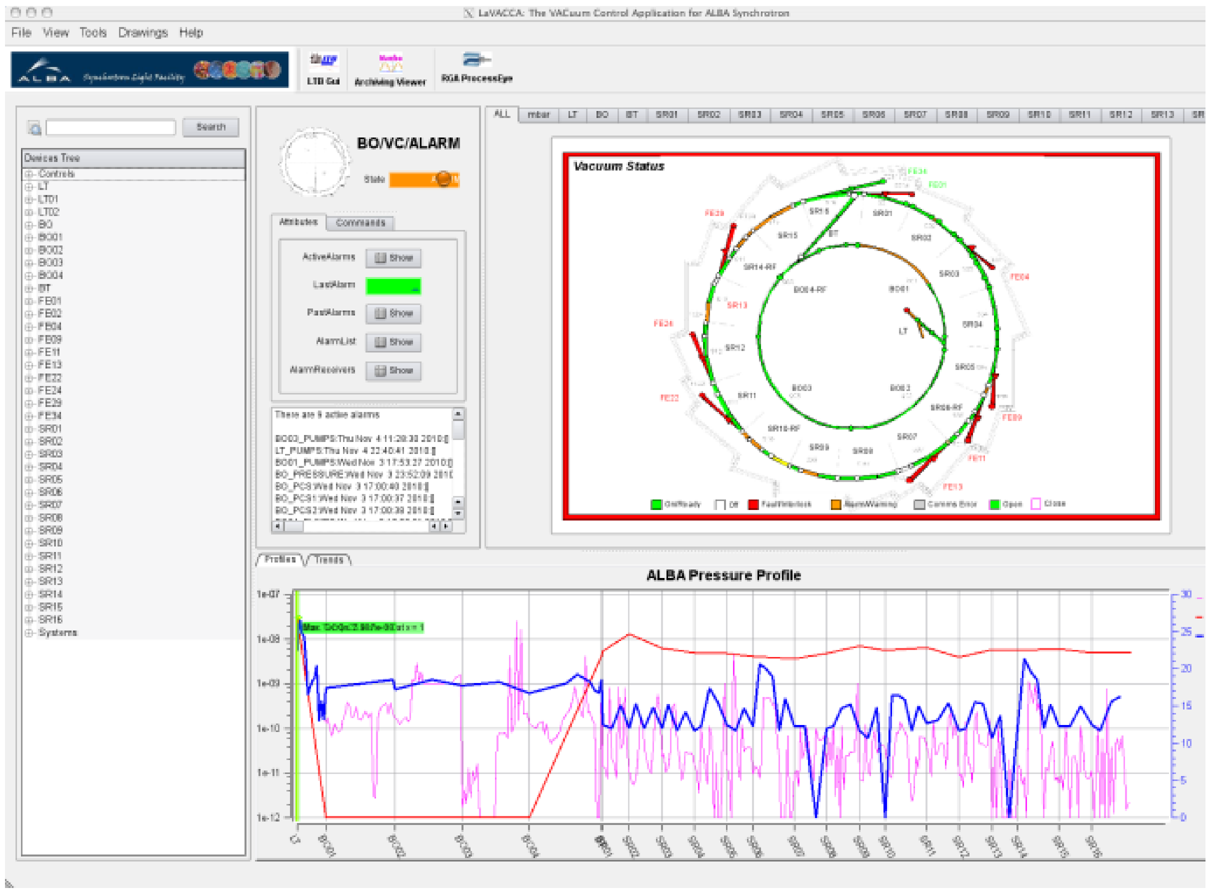
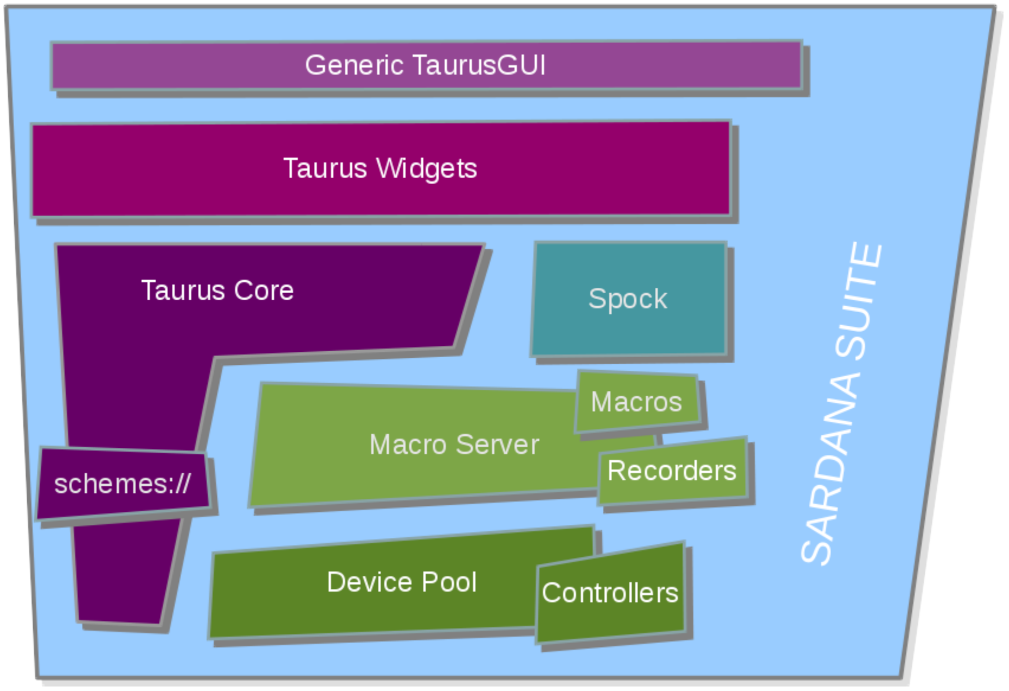


Fig. 18: ALBA[85]'s vaccum GUI

FAQ

What is the Sardana SCADA[86] and how do I get an overview over the different components?

An overview over the different Sardana components is shown in the following figure:



The basic Sardana SCADA[87] philosophy can be found *here*.

How do I install Sardana?

The Sardana SCADA[88] system consists of different components which have to be installed:

- Tango[89]: The control system middleware and tools
- PyTango[90]: The Python[91] language binding for Tango[92]
- Taurus[93]: The GUI toolkit which is part of Sardana SCADA[94]
- The Sardana device pool, macro server and tools

The complete sardana installation instructions can be found *here*.

How to work with Taurus[95] GUI?

[86] http://en.wikipedia.org/wiki/SCADA
[87] http://en.wikipedia.org/wiki/SCADA
[88] http://en.wikipedia.org/wiki/SCADA
[89] http://www.tango-controls.org/
[90] http://packages.python.org/PyTango/
[91] http://www.python.org/
[92] http://www.tango-controls.org/
[93] http://packages.python.org/taurus/
[94] http://en.wikipedia.org/wiki/SCADA
[95] http://packages.python.org/taurus/

A user documentation for the Taurus[96] *GUI* application can be found here[97].

How to produce your own Taurus[98] GUI panel?

The basic philosophy of Taurus[99] *GUI* is to provide automatic *GUI* s which are automatically replaced by more and more specific *GUI* s if these are found.

Refer to the user documentation on TaurusGUI[100] for more details on how to work with panels

How to call procedures?

The central idea of the Sardana SCADA[101] system is to execute procedures centrally. The execution can be started from either:

- *spock* offers a command line interface with commands very similar to SPEC[102]. It is documented *here*.
- Procedures can also be executed with from a *GUI*. Taurus provides generic widgets for macro execution[103].
- Procedures can also be executed in specific *GUI* s and specific Taurus[104] widgets. The *API* to execute macros from python code is documented here **<LINK>**.

How to write procedures?

User written procedures are central to the Sardana SCADA[105] system. Documentation how to write macros can be found *here*. Macro writers might also find the following documentation interesting:

- Documentation on how to debug macros can be found here **<LINK>**
- In addition of the strength of the python language macro writers can interface with common elements (motors, counters) , call other macros and use many utilities provided. The macro *API* can be found *here*.
- Documentation how to document your macros can be found *here*

How to write scan procedures?

A very common type of procedure is the *scan* where some quantity is varied while recording some other quantities. See the documentation on the *Sardana Scan API*

[96] http://packages.python.org/taurus/
[97] http://packages.python.org/taurus/
[98] http://packages.python.org/taurus/
[99] http://packages.python.org/taurus/
[100] http://www.tango-controls.org/static/taurus/latest/doc/html/users/ui/taurusgui.html
[101] http://en.wikipedia.org/wiki/SCADA
[102] http://www.certif.com/
[103] http://www.tango-controls.org/static/taurus/latest/doc/html/users/ui/macros/
[104] http://packages.python.org/taurus/
[105] http://en.wikipedia.org/wiki/SCADA

How to adapt SARDANA to your own hardware?

Sardana is meant to be interfaced to all types of different hardware with all types of control systems. For every new hardware item the specific behavior has to be programmed by writing a controller code. The documentation how to write Sardana controllers and pseudo controllers can be found *here*. This documentation also includes the *API* which can be used to interface to the specific hardware item.

How to add your own file format?

Documentation how to add your own file format can be found here **<LINK>**.

How to use the standard macros?

The list of all standard macros and their usage can be found here **<LINK>**.

How to write your own Taurus application?

You have basically two possibilities to write your own Taurus[106] application Start from get General TaurusGUI and create a configuration file. This approach is documented here **<LINK>**. Start to write your own Qt application in python starting from the Taurus[107] main window. This approach is documented here **<LINK>**.

Which are the standard Taurus graphical GUI components?

A list of all standard Taurus GUI components together with screen shots and example code can be found here **<LINK>**

How to write your own Taurus widget?

A tutorial of how to write your own Taurus widget can be found *here*.

How to work with the graphical GUI editor?

Taurus[108] uses the QtDesigner/QtCreator as a graphical editor. Documentation about QtDesigner/QtCreator[109]. The Taurus[110] specific parts here[111].

What are the minimum software requirements for sardana?

Sardana is developed under GNU/Linux, but should run also on Windows and OS-X. The dependencies for installing Sardana can be found here **<LINK>**.

[106] http://packages.python.org/taurus/
[107] http://packages.python.org/taurus/
[108] http://packages.python.org/taurus/
[109] http://qt.nokia.com/products/developer-tools/
[110] http://packages.python.org/taurus/
[111] http://taurus-scada.org/devel/designer_tutorial.html#taurusqtdesigner-tutorial

How to configure the system?

Adding and configuring hardware items on an installation is described here **<LINK>**.

How to write your own Taurus schema?

Taurus is not dependent on Tango. Other control systems or just python modules can be interfaced to it by writing a schema. This approach is documented here **<LINK>** and a tutorial can be found here **<LINK>**

What are the interfaces to the macro server and the pool?

The low level interfaces to the Sardana Device Pool and the Macro server can be found here **<LINK>**.

What are the data file formats used in the system and how can I read them?

It is easily possible to add your own file format but the standard file formats are documented here:

- The SPEC[112] file format is documented here **<LINK>** and here is a list of tools to read it **<LINK>**
- The EDF file format is documented here **<LINK>** and here is a list of tools to read it **<LINK>**
- The NEXUS file format is documented here **<LINK>** and here is a list of tools to read it **<LINK>**

What is the file format of the configuration files?

The configuration files for the Taurus[113] GUI are defined here **<LINK>**.

1.1.2 Developer's Guide

Overview

Global overview

This chapter gives an overview of the sardana architecture and describes each of the different components in some detail. If you find this document to be to technical please consider reading the *Overview* guide first.

The following chapters assume a that you have a minimum knowledge of the Tango[114] system and basic computer science.

Architecture

Sardana consists of a software library which contains sardana kernel engine, a server and a client library which allow sardana to run as a *client-server* based distributed control system. The communication protocols between servers and clients are *plug-ins* in sardana. At this time, the only implemented protocol is

[112] http://www.certif.com/
[113] http://packages.python.org/taurus/
[114] http://www.tango-controls.org/

Tango[115]. In earlier versions, sardana was tightly connected to Tango[116]. This documentation, is therefore centered in the Tango[117] server implementation. When other comunication protocols become available, the documentation will be revised.

Client applications (both *GUI* and *CLI*) can connect to the sardana server through the high level sardana client *API* or through the low level pure Tango[118] channels. Client applications can be build with the purpose of *operating* an existing sardana server or of *configuring* it.

Sardana server (SDS)

The sardana server consists of a sardana tango device server (*SDS*) running a sardana kernel engine. This server runs as an *OS daemon*. Once configured, this server acts as a container of device objects which can be accessed by the outside world as *tango device objects*. Typically, a sardana server will consist of:

- a low level **Pool** object which manages all the server objects related to motion control and data acquisition (controllers, motors, counters, experiment channels, etc).
- a **Macro Server** object which manages the execution of macros (procedures) and client connection points (called doors).
- a set of low level objects (controllers, motors, counters, experiment channels, etc) controlled by the Pool object
- a set of **Door** objects managed by the macro server. A Door is the preferred access point from a client application to the to the sardana server

A sardana server may contain only a Pool object or a Macro Server object or both. It may **NOT** contain more than one Pool object or more than one Macro Server object.

If necessary, your sardana system may be splitted into two (or more) sardana servers. A common configuration is to have a sardana server with a Pool (in this case we call the server a *Device Pool* server) and a second server with a Macro Server (this server is called *MacroServer* server).

The following figures show some of the possible alternative configurations

The following chapters describe each of the Sardana objects in more detail.

Macro Server overview

The Macro Server object is the sardana server object which manages all high level sardana objects related to macro execution, namely doors, macro libraries and macros themselves.

The main purpose of the Macro Server is to run *macros*. Macros are just pieces of Python[119] code (functions or classes) which reside in a macro library (Python[120] file). Macros can be written by anyone with knowledge of Python[121].

The Macro Server is exposed on the sardana server as a Tango[122] device. Through configuration, the Macro Server can be told to connect to a Pool device. This is the most common configuration. You can, however, tell the Macro Server to connect to more than one Pool device or to no Pool devices at all.

[115] http://www.tango-controls.org/
[116] http://www.tango-controls.org/
[117] http://www.tango-controls.org/
[118] http://www.tango-controls.org/
[119] http://www.python.org/
[120] http://www.python.org/
[121] http://www.python.org/
[122] http://www.tango-controls.org/

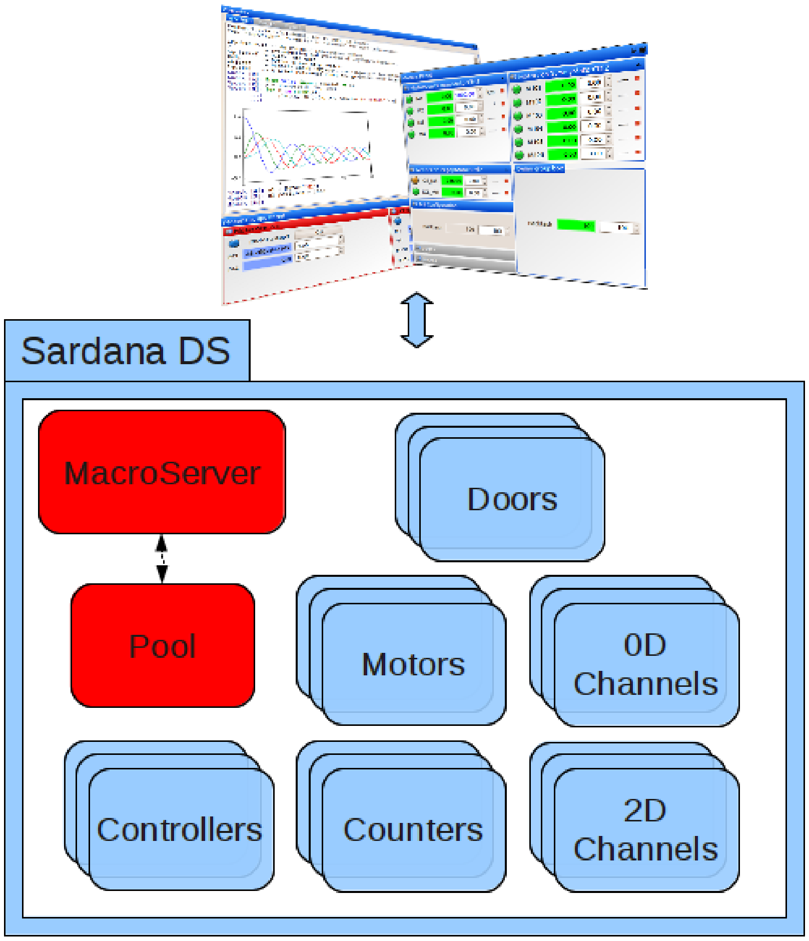


Fig. 19: A diagram representing a sardana server with its objects

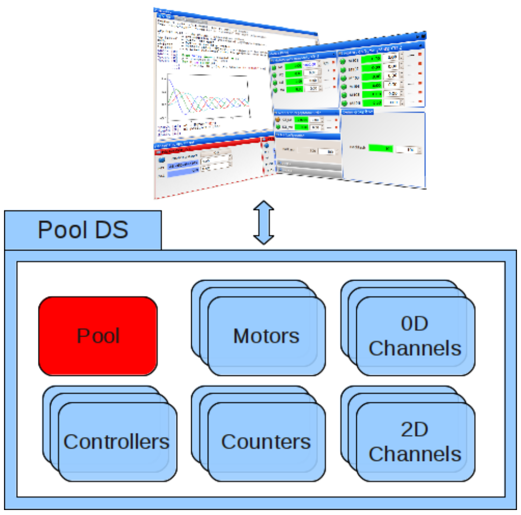


Fig. 20: 1 - Sardana configured to be a single Pool DS (no MacroServer present)

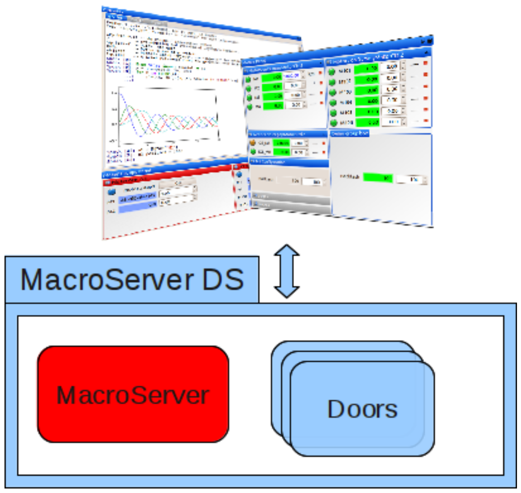


Fig. 21: 2 - Sardana configured to be a single MacroServer DS (no Pool present)

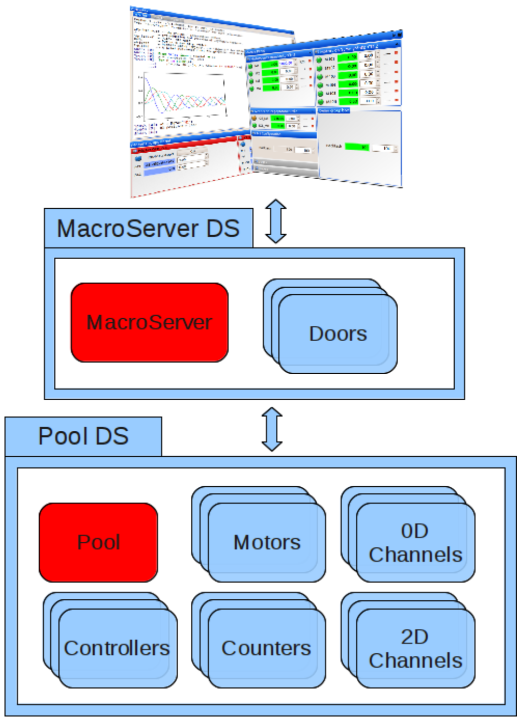


Fig. 22: 3 - Sardana configured with a MacroServer DS connecting to an underlying Pool DS

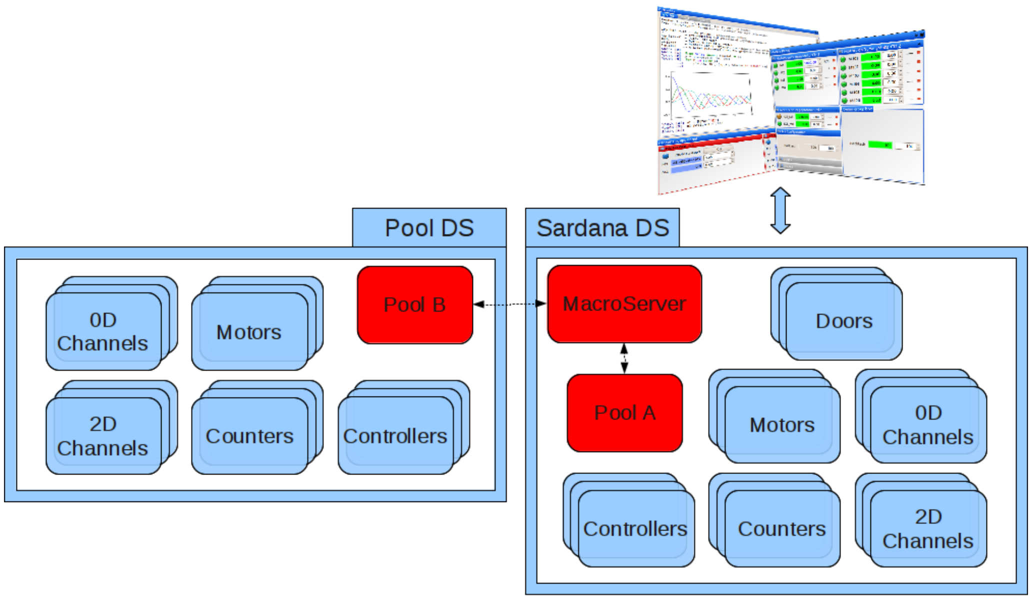


Fig. 23: 4 - Sardana configured with a Sardna DS connecting to another underlying Pool DS

When connected to a Pool device(s), the Macro Server uses the Pool device introspection *API* to discover which elements are available. The existing macros will be able to access these elements (through parameters passed to the macro or using the macro *API*) and act on them.

In order to be able to run macros, you must first connect to the Macro Server entry point object called *Door*. A single Macro Server can have many active Doors at the same time but a Door can only run one macro at a time. Each Door is exposed on the sardana server as a Tango[123] device.

You are not in any way restricted to the standard macros provided by the sardana system. You can write as many macros as you need. Writing your own macros is easy. The macro equivalent of Python[124]'s *Hello, World!* example:

```
from sardana.macroserver.macro import macro

@macro()
def hello_world(self):
    self.output("Hello, World!")
```

Here is a simple example of a macro to move any moveable element to a certain value:

```
from sardana.macroserver.macro import macro, Type

@macro([ ["moveable", Type.Moveable, None, "moveable to move"],
         ["position", Type.Float, None, "absolute position"] ])
def my_move(self, moveable, position):
    """This macro moves a moveable to the specified position"""

    moveable.move(position)
    self.output("%s is now at %s", moveable, moveable.getPosition())
```

Information on how to write your own sardana macros can be found *here*.

The complete macro *API* can be found *here*.

[123] http://www.tango-controls.org/
[124] http://www.python.org/

Pool overview

The Pool object is the sardana server object which manages all other hardware level sardana objects related with motion control and data acquisition. This object is exposed to the world as a Tango[125] device. It's *API* consists of a series of methods (Tango[126] commands) and members (Tango[127] attributes) which allow external applications to create/remove/rename and monitor the different hardware level sardana objects.

The Pool could be seen as a kind of intelligent device container to control the experiment hardware. It has two basic features which are:

1. Hardware access using dynamically created/deleted devices according to the experiment needs
2. Management of some very common and well defined actions regularly done on a laboratory/factory (motion control, data acquisition, etc.)

Hardware access

Core hardware access

Most of the times, it is possible to define a list of very common objects found in most of the experiments. Objects commonly used to drive an experiment usually fit in one of the following categories:

- ***Moveables***
    - Motor
    - Pseudo motor
    - Group of moveables
    - IORegister (a.k.a. discrete motor)
- ***Experimental channels***
    - Counter/Timer
    - 0D (Multimeter like)
    - 1D (*MCA* like)
    - 2D (*CCD* like)
    - Pseudo Counter
- *Communication channels*

Each different controlled hardware object will also be exposed as an independent Tango[128] class. The sardana device server will embed all these Tango[129] classes together. The pool Tango[130] device is the "container interface" and allows the user to create/delete classical Tango[131] devices which are instances of these embedded classes.

[125] http://www.tango-controls.org/
[126] http://www.tango-controls.org/
[127] http://www.tango-controls.org/
[128] http://www.tango-controls.org/
[129] http://www.tango-controls.org/
[130] http://www.tango-controls.org/
[131] http://www.tango-controls.org/

Controller overview

Each different hardware object is directly controlled by a software object called *controller*. This object is responsible for mapping the communication between a set of hardware objects (example motors) and the underlying hardware (example: a motor controller crate). The *controller* object is also exposed as a Tango[132] device.

Usually a controller is capable of handling several hardware objects. For example, a motor controller crate is capable of controlling several motors (generally called *axis*[136]).

The controller objects can be created/deleted/renamed dynamically in a running pool.

A specific type of controller needs to be created to handle each specific type of hardware. Therefore, to each type of hardware controller there must be associated a specific controller software component. You can write a specific controller software component (*plug-in*) that is able to communicate with the specific hardware. You can this way extend the initial pool capabilities to talk to all kinds of different hardware.

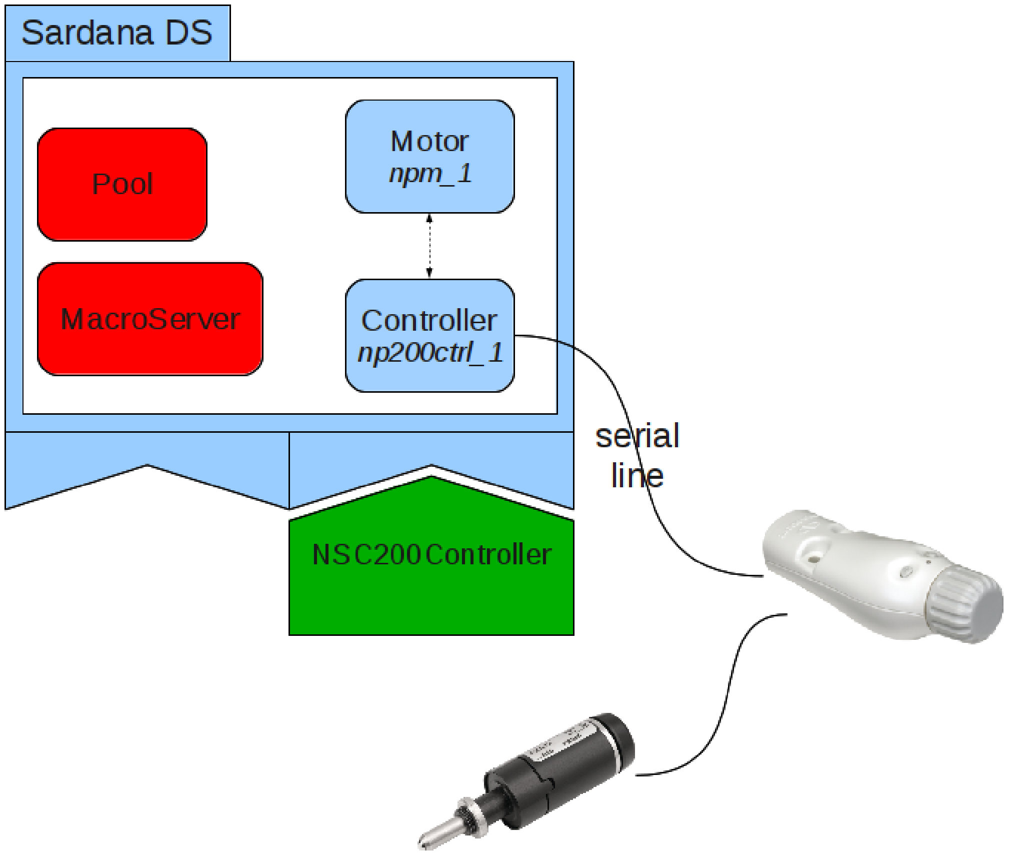


Fig. 24: A diagram representing a sardana server with a controller class *NSC200Controller*, an instance of that controller *np200ctrl_1* "connected" to a real hardware and a single motor *npm_1*.

A sardana controller is responsible for it's sardana element(s). Example: an Icepap hardware motor controller can *control* up to 128 individual motor axis. In the same way, the coresponding software motor controller *IcepapController* will *own* the individual motor axises.

These are the different types of controllers recognized by sardana:

[132] http://www.tango-controls.org/

[136] The term *axis* will be used from here on to refer to the ID of a specific hardware object (like a motor) with respect to its *controller*.

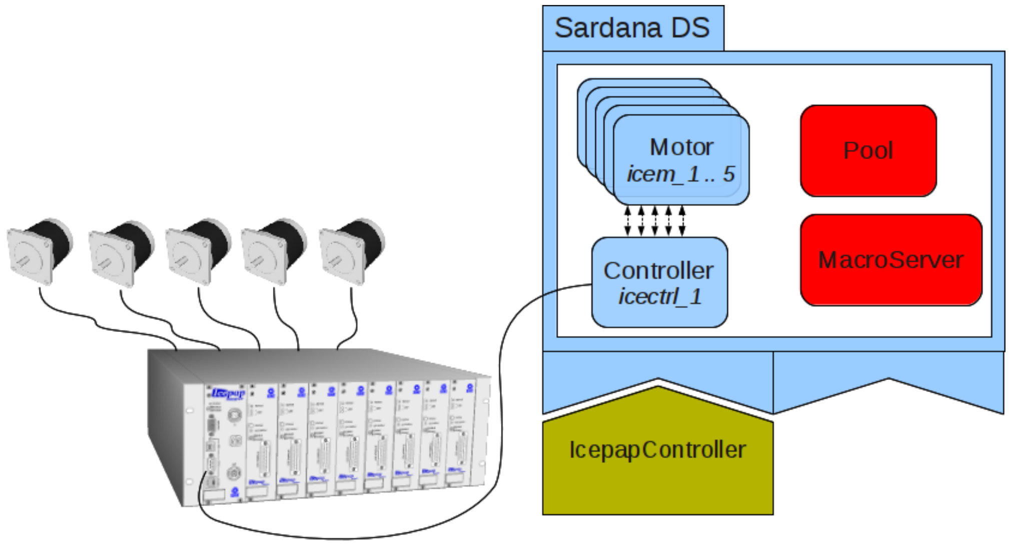


Fig. 25: A diagram representing a sardana server with a controller class *IcepapController*, an instance of that controller *icectrl_1* "connected" to a real hardware and motors *icem_[1..5]*.

`MotorController` You should use/write a `MotorController` sardana *plug-in* if the the device you want to control has a *moveable* interface. The `MotorController` actually fullfils a *changeable* interface. This means that, for example, a power supply that has a current which you want to *ramp* could also be implemented as a `MotorController`.

Example: the Newport NSC200 motor controller

`CounterTimerController` This controller type is designed to control a device capable of counting scalar values (and, optionaly have a timer).

Example: The National Instruments 6602 8-Channel Counter/Timer

`ZeroDController` This controller type is designed to control a device capable of supplying scalar values. The *API* provides a way to obtain a value over a certain acquisition time through different algorithms (average, maximum, integration).

Example: an electrometer

`OneDController` This controller type is designed to control a device capable of supplying 1D values. It has a very similar *API* to `CounterTimerController`

Example: an *MCA*

`TwoDController` This controller type is designed to control a device capable of supplying 2D values. It has a very similar *API* to `CounterTimerController`

Example: a *CCD*

`PseudoMotorController` A controller designed to export *virtual motors* that represent a new view over the actual physical motors.

Example: A slit pseudo motor controller provides *gap* and *offset* virtual motors over the physical blades

`PseudoCounterController` A controller designed to export *virtual counters* that represent a new view over the actual physical counters/0Ds.

IORegisterController A controller designed to control hardware registers.

Controller plug-ins can be written in Python[133] (and in the future in C++). Each controller code is basically a Python[134] class that needs to obey a specific *API*.

Here is an a extract of the pertinent part of a Python[135] motor controller code that is able to talk to a Newport motor controller:

```
from sardana.pool.controller import MotorController, \
    Type, Description, DefaultValue

class NSC200Controller(MotorController):
    """This class is the Tango Sardana motor controller for the Newport NewStep
    handheld motion controller NSC200.
    This controller communicates through a Device Pool serial communication
    channel."""

    ctrl_properties = \
        { 'SerialCh' : { Type : str,
                         Description : 'Communication channel name for the serial line
↪' },
          'SwitchBox': { Type : bool,
                         Description : 'Using SwitchBox',
                         DefaultValue : False},
          'ControllerNumber' : { Type : int,
                                 Description : 'Controller number',
                                 DefaultValue : 1 } }

    def __init__(self, inst, props, *args, **kwargs):
        MotorController.__init__(self, inst, props, *args, **kwargs)

        self.serial = None
        self.serial_state_event_id = -1

        if self.SwitchBox:
            self.MaxDevice = 8

    def AddDevice(self, axis):
        if axis > 1 and not self.SwitchBox:
            raise Exception("Without using a Switchbox only axis 1 is allowed")

        if self.SwitchBox:
            self._setCommand("MX", axis)

    def DeleteDevice(self, axis):
        pass

    _STATE_MAP = { NSC200.MOTOR_OFF : State.Off, NSC200.MOTOR_ON : State.On,
                   NSC200.MOTOR_MOVING : State.Moving }

    def StateOne(self, axis):
        if self.SwitchBox:
            self._setCommand("MX", axis)

        status = int(self._queryCommand("TS"))
```

(continues on next page)

133 http://www.python.org/
134 http://www.python.org/
135 http://www.python.org/

(continued from previous page)

```
        status = self._STATE_MAP.get(status, State.Unknown)
        register = int(self._queryCommand("PH"))
        lower = int(NSC200.getLimitNegative(register))
        upper = int(NSC200.getLimitPositive(register))

        switchstate = 0
        if lower == 1 and upper == 1: switchstate = 6
        elif lower == 1: switchstate = 4
        elif upper == 1: switchstate = 2
        return status, "OK", switchstate

    def ReadOne(self, axis):
        try:
            if self.SwitchBox:
                self._setCommand("MX", axis)
            return float(self._queryCommand("TP"))
        except:
            raise Exception("Error reading position, axis not available")

    def PreStartOne(self, axis, pos):
        return True

    def StartOne(self, axis, pos):
        if self.SwitchBox:
            self._setCommand("MX", axis)
        status = int(self._queryCommand("TS"))
        if status == NSC200.MOTOR_OFF:
            self._setCommand("MO","")
        self._setCommand("PA", pos)
        self._log.debug("[DONE] sending position")

    def StartAll(self):
        pass

    def AbortOne(self, axis):
        if self.SwitchBox:
            self._setCommand("MX", axis)
        self._setCommand("ST", "")
```

See also:

Writing controllers How to write controller *plug-in*s in sardana

Controller API reference the controller *API*

`Controller` the controller tango device *API*

Motor overview

The motor is one of the most used elements in sardana. A motor represents anything that can be *changed* (and can potentially take some time to do it), so, not only physical motors (like a stepper motors) fit into this category but also, for example, a power supply for which the electrical current can be modified. As it happens with the motor controller hardware and its physical motor(s), a sardana motor is always associated with its sardana motor controller.

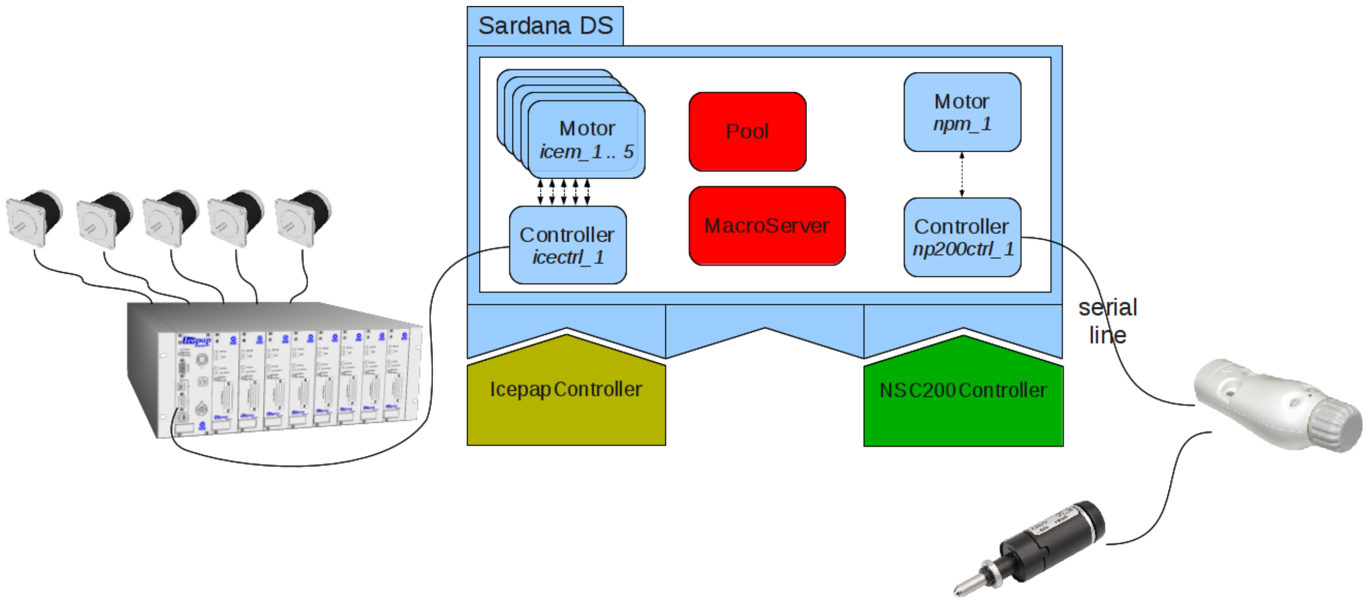


Fig. 26: A diagram representing a sardana server with a several motor controllers and their respective motors.

The *motor* object is also exposed as a Tango[137] device.

See also:

Motor API reference the motor *API*

Motor the motor tango device *API*

Pseudo motor overview

The pseudo motor interface acts like an abstraction layer for a motor or a set of motors allowing the user to control the experiment by means of an interface which is more meaningful to him(her).

One of the most basic examples is the control of a slit. The slit has two blades with one motor each. Usually the user doesn't want to control the experiment by directly handling these two motor positions since they have little meaning from the experiments perspective. Instead, it would be more useful for the user to control the experiment by means of changing the gap and offset values. In the `Slit` controller, pseudo motors gap and offset will provide the necessary interface for controlling the experiments gap and offset values respectively.

Fig. 27: An animation[139] representing a system of slits composed from horizontal blades (left and right) an vertical blades (top and bottom).

In order to translate the motor positions into the pseudo motor positions and vice versa, calculations have to be performed. The device pool provides `PseudoMotorController` class that can be overwritten to provide new calculations.

The pseudo motor position gets updated automatically every time one of its motors position gets updated e.g. when the motion is in progress.

[137] http://www.tango-controls.org/
[139] We would like to thank Dominique Heinis for sharing his expertise in blender.

The pseudo motor object is also exposed as a Tango[138] device.

See also:

Pseudo motor API reference the pseudo motor *API*

`PseudoMotor` the pseudo motor tango device *API*

Advanced topics

Drift correction

Pseudomotors which have siblings and are based on physical motors with an inaccurate or a finite precision positioning system could be affected by the drift effect.

Why does it happen?

Each move of a pseudomotor requires calculation of the physical motors positions in accordance with the current positions of its siblings. The consecutive movements of a pseudomotor can accumulate errors of the positioning system and cause drift of its siblings.

Who is affected?

- **Inaccurate positioning systems** which lead to a discrepancy between the write and the read position of the physical motors. In this case the physical motors must have a position sensor e.g. encoder but must not be configured in *closed loop* (in some special cases, where the closed loop is not precise enough, the drift effect can be observed as well). This setup can lead to the situation where write and read values of the position attribute of the physical motors are different e.g. due to the loosing steps problems or the inaccurate *step_per_unit* calibration.
- **Finite precision physical motors** e.g. *stepper* is affected by the rounding error when moving to a position which does not translate into a discrete number of steps that must be commanded to the hardware.

How is it solved in Sardana?

Sardana implements the drift correction which use is optional but enabled by default for all pseudomotors. It is based on the use of the write value, instead of the read value, of the siblings' positions, together with the new desired position of the pseudomotor being moved, during the calculation of the physical positions. The write value of the pseudomotor's position gets updated at each move of the pseudomotor or any of the underneath motors.

Note: Movements being stopped unexpectedly: abort by the user, over-travel limit or any other exceptional condition may cause considerable discrepancy in the motor's write and read positions. In the subsequent pseudomotor's move, Sardana will also correct this difference by using the write instead of read values.

The drift correction is configurable with the *DriftCorrection* property either globally (on the Pool device level) or locally (on each PseudoMotor device level).

Example

Let's use the slit pseudomotor controller to visualize the drift effect. This controller comprises two pseudomotors: gap and offset, each of them based on the same two physical motors: right and left. In this example we will simulate the inaccurate positioning of the left motor (loosing of 0.002 unit every 1 unit move).

Drift correction disabled

[138] http://www.tango-controls.org/

1. Initial state: gap and offset are at positions 0 (gap totally closed and offset at the nominal position)

   ```
   Door_lab_1 [1]: wm right left gap offset
                     right          left           gap        offset
   User
    High      Not specified  Not specified  Not specified  Not specified
    Current           0.000          0.000          0.000          0.000
    Low       Not specified  Not specified  Not specified  Not specified
   ```

2. Move gap to 1

   ```
   Door_lab_1 [2]: mv gap 1
   ```

   The calculation of the physical motors' positions gives us 0.5 for both right and left (in accordance with the current offset of 0)

   ```
   Door_lab_1 [3]: wm right left gap offset
                     right          left           gap        offset
   User
    High      Not specified  Not specified  Not specified  Not specified
    Current           0.500          0.498          0.998          0.001
    Low       Not specified  Not specified  Not specified  Not specified
   ```

   We observe that the gap pseudomotor did not reach the desired position of 1 due to the left's positioning problem. Left's position write and read discrepancy of 0.002 causes that the gap reached only 0.998 and that the offset drifted to 0.001.

3. Move gap to 2

   ```
   Door_lab_1 [4]: mv gap 2
   ```

   The calculation of the physical motors' positions gives us 1.001 for right and 0.999 for left (in accordance with the current offset of 0.001).

   ```
   Door_lab_1 [5]: wm right left gap offset
                     right          left           gap        offset
   User
    High      Not specified  Not specified  Not specified  Not specified
    Current           1.001          0.997          1.998          0.002
    Low       Not specified  Not specified  Not specified  Not specified
   ```

   We observe that the gap pseudomotor did not reach the desired position of 2 due to the left's positioning problem. Left's position write and read discrepancy of 0.002 causes that the gap reached only 1.998 and that the offset drifted again by 0.001 and the total accumulated drift is 0.002.

4. Move gap to 3

   The calculation of the physical motors' positions gives us 1.502 for right and 1.498 for left (in accordance with the current offset of 0.002).

   ```
   Door_lab_1 [6]: mv gap 3

   Door_lab_1 [7]: wm right left gap offset
                     right          left           gap        offset
   User
    High      Not specified  Not specified  Not specified  Not specified
    Current           1.502          1.496          2.998          0.003
    Low       Not specified  Not specified  Not specified  Not specified
   ```

We observe that the gap pseudomotor did not reach the desired position of 3 due to the left's positioning problem. Left's position write and read discrepancy of 0.002 causes that the gap reached only 2.998 and that the offset drifted by 0.001 and the total accumulated drift is 0.003.

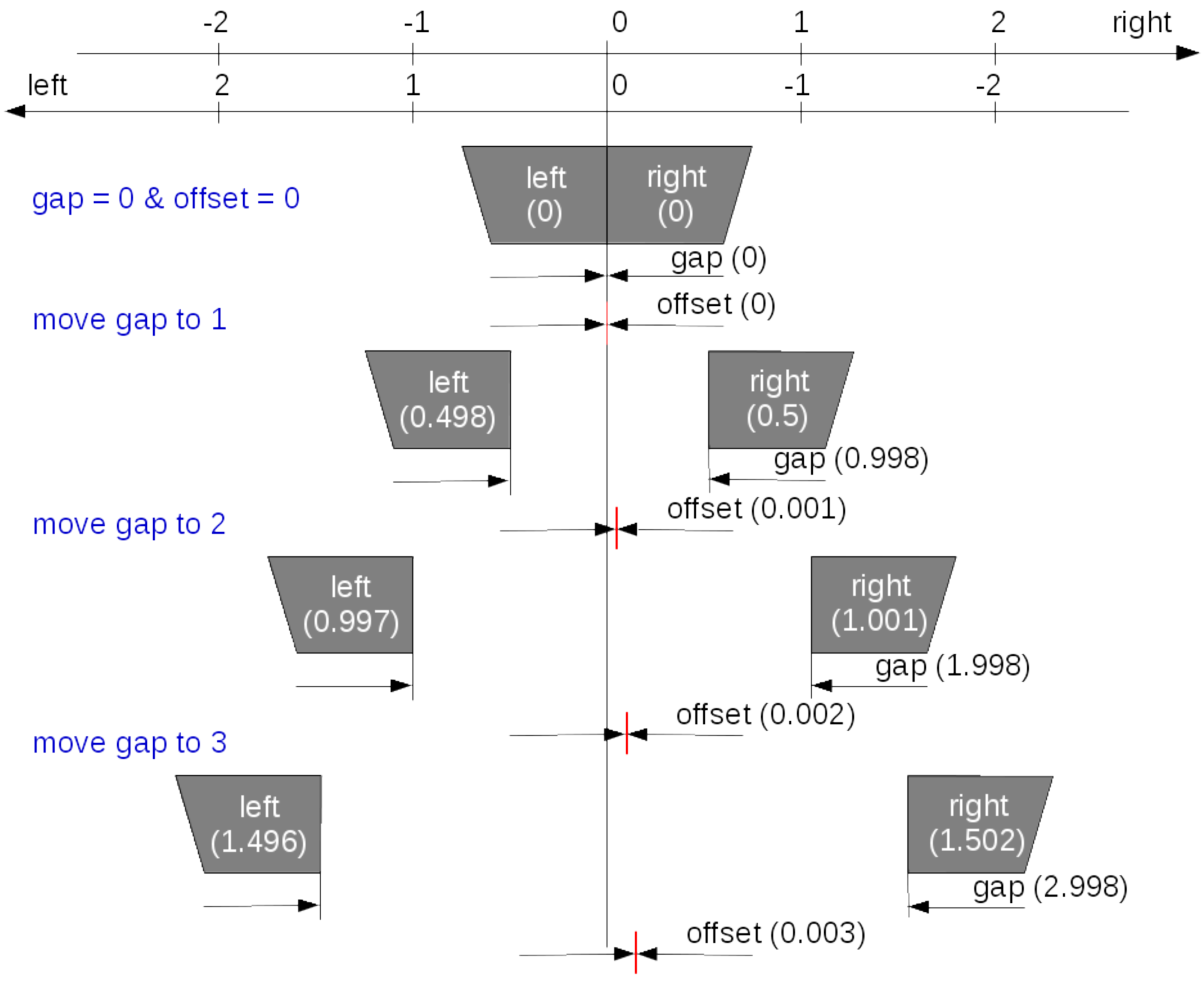


Fig. 28: This sketch demonstrates the above example where offset drifted by 0.003.

Drift correction enabled

1. Initial state: gap and offset are at positions 0 (gap totally closed and offset at the nominal position)

```
Door_lab_1 [1]: wm right left gap offset
                  right          left             gap          offset
User
 High      Not specified  Not specified  Not specified  Not specified
 Current           0.000          0.000          0.000          0.000
 Low       Not specified  Not specified  Not specified  Not specified
```

2. Move gap to 1

```
Door_lab_1 [2]: mv gap 1
```

The calculation of the physical motors' positions gives us 0.5 for both right and left (in accordance with the **last set** offset of 0).

```
Door_lab_1 [3]: wm right left gap offset
                  right           left             gap          offset
User
 High      Not specified  Not specified  Not specified  Not specified
 Current           0.500          0.498          0.998          0.001
 Low       Not specified  Not specified  Not specified  Not specified
```

We observe that the gap pseudomotor did not reach the desired position of 1 due to the left's positioning problem. Left's position write and read discrepancy of 0.002 causes that the gap reached only 0.998 and that the offset drifted to 0.001.

3. Move gap to 2

```
Door_lab_1 [4]: mv gap 2
```

The calculation of the physical motors' positions gives us 1 for right and 1 for left (in accordance to the **last set** offset 0).

```
Door_lab_1 [5]: wm right left gap offset
                  right           left             gap          offset
User
 High      Not specified  Not specified  Not specified  Not specified
 Current           1.000          0.998          1.998          0.001
 Low       Not specified  Not specified  Not specified  Not specified
```

We observe that the gap pseudomotor did not reach the desired position of 2 due to the left's positioning problem. Left's position write and read discrepancy of 0.002 causes that the gap reached only 1.998 and that the offset drifted again by 0.001 but thanks to the drift correction is maintained at this value.

4. Move gap to 3

```
Door_lab_1 [6]: mv gap 3
```

The calculation of the physical motors' positions gives us 1.5 for right and 1.5 for left (in accordance to the **last set** offset of 0).

```
Door_lab_1 [7]: wm right left gap offset
                  right           left             gap          offset
User
 High      Not specified  Not specified  Not specified  Not specified
 Current           1.500          1.498          2.998          0.001
 Low       Not specified  Not specified  Not specified  Not specified
```

We observe that the gap pseudomotor did not reach the desired position of 3 due to the left's positioning problem. Left's position write and read discrepancy of 0.002 causes that the gap reached only 2.998 and that the offset drifted again by 0.001 but thanks to the drift correction is maintained at this value.

I/O register overview

The IOR is a generic element which allows to write/read from a given hardware register a value. This value type may be one of: `int`[140], `float`[141], `bool`[142] but the hardware usually expects a fixed type for a given

[140] https://docs.python.org/dev/library/functions.html#int
[141] https://docs.python.org/dev/library/functions.html#float
[142] https://docs.python.org/dev/library/functions.html#bool

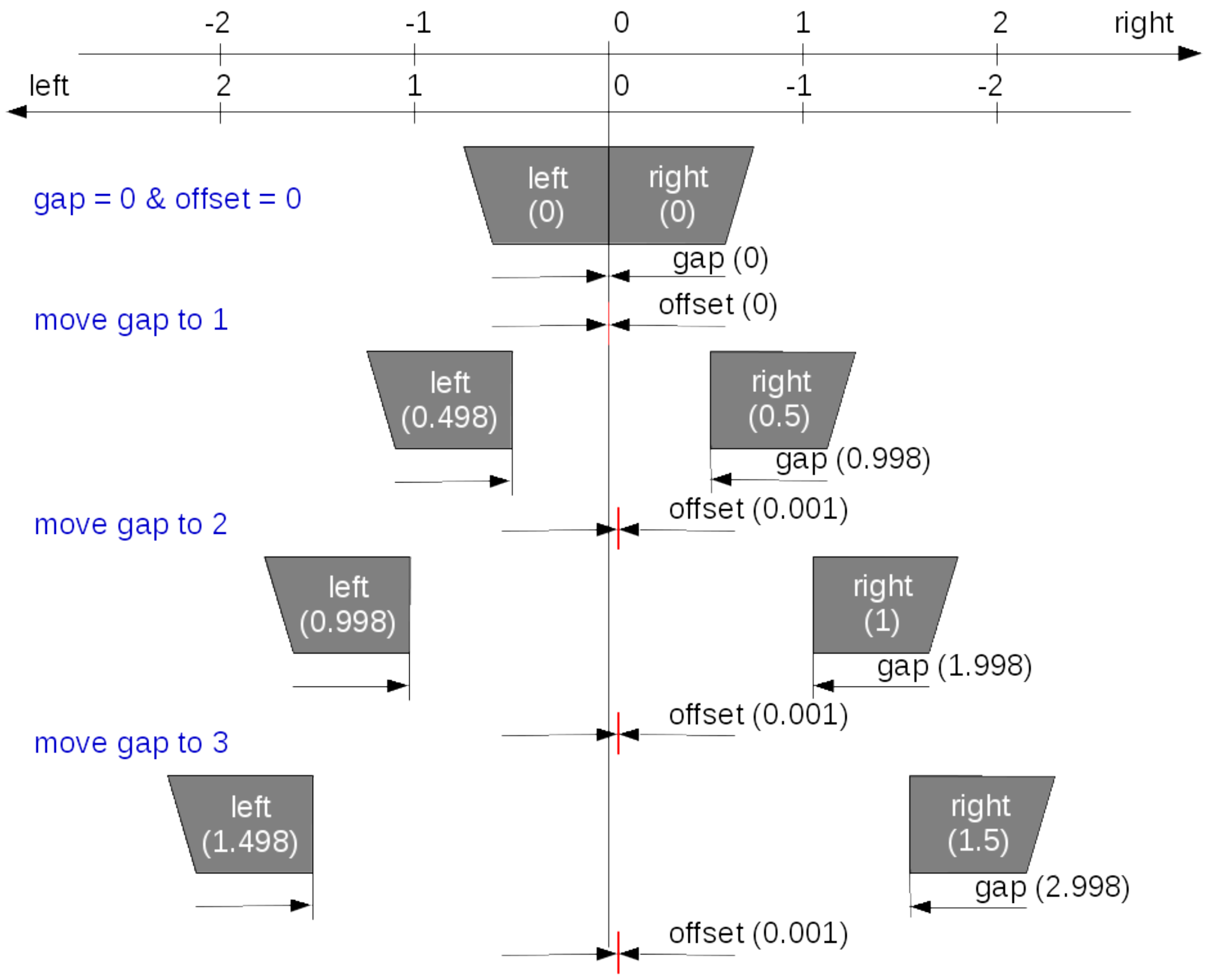


Fig. 29: This sketch demonstrates the above example where offset's drift was corrected.

register.

The IOR has a very wide range of applications it can serve to control the *PLC* registers, a discrete motor, etc.

See also:

I/O register API reference the I/O register *API*

`IORegister` the I/O register tango device *API*

Trigger/gate overview

The trigger/gate represents synchronization devices like for example the digital trigger and/or gate generators. Their main role is to synchronize acquisition of the experimental channels.

Trigger or gate characteristics could be described in either the time and/or the position configuration domains.

In the time domain, elements are configured in time units (seconds) and generation of the synchronization signals is based on passing time.

The concept of position domain is based on the relation between the trigger/gate and the moveable element. In the position domain, elements are configured in distance units of the moveable element configured as the feedback source (this could be mm, mrad, degrees, etc.). In this case generation of the synchronization signals is based on receiving updates from the source.

See also:

Trigger/Gate API reference the trigger/gate *API*

`TriggerGate` the trigger/gate tango device *API*

Counter/timer overview

The counter/timer is one of the most used elements in Sardana. A counter/timer represents an experimental channel which acquisition result is a scalar value. As indicates its name it is foreseen to interface hardware couters or timers but it also fits well with other hardware like *ADC* or electrometer.

The acquisition operation on a counter/timer is executed over the integration time specified by the user. Counter/timer can be controlled by either software or hardware synchronization (*Trigger/Gate*) and multiple repetitions, also specified by the user are, are possible within the same acquisition operation.

See also:

Counter/Timer API reference the counter/timer *API*

`CTExpChannel` the counter/timer tango device *API*

0D channel overview

The 0D experimental channel is used to access any kind of device which returns a scalar value and which are not counter/timer. Very often (but not always), this is a commercial measurement equipment connected to a GPIB bus.

In order to have as precise as possible measurement, a dedicated acquisition operation is implemented for 0D channels. This operation will simply read the data from the hardware as fast as it can (only "sleeping" 10 mS between each reading) and a computation is done on the resulting data set to return only one value. Three types of computation are foreseen. The user selects which one he needs with an attribute.

The time during which this acquisition loop will get data is controlled by the counters/timers present in the measurement group - when all of them finish acquiring the 0D acquisition operation will also stop.

See also:

0D channel API reference the 0D experiment channel *API*

ZeroDExpChannel the 0D experiment channel tango device *API*

1D channel overview

The 1D represents an experimental channel which acquisition result is a spectrum value. It is foreseen to interface with *MCA* or position sensitive detectors.

The acquisition operation on a 1D channel is executed over the integration time specified by the user. 1D channels can be controlled by either software or hardware synchronization (*Trigger/Gate*) and multiple repetitions, also specified by the user are, are possible within the same acquisition operation.

See also:

1D channel API reference the 1D experiment channel *API*

OneDExpChannel the 1D experiment channel tango device *API*

2D channel overview

The 2D represents an experimental channel which acquisition result is an image value. It is foreseen to interface with *CCD* or photon-counting array detectors.

The acquisition operation on a 2D channel is executed over the integration time specified by the user. 2D channels can be controlled by either software or hardware synchronization (*Trigger/Gate*) and multiple repetitions, also specified by the user are, are possible within the same acquisition operation.

See also:

2D channel API reference the 2D experiment channel *API*

TwoDExpChannel the 2D experiment channel tango device *API*

Pseudo counter overview

Pseudo counter acts like an abstraction layer for a counter or a set of counters allowing the user to see the experiment results by means of an interface which is more meaningful to him.

One example of a pseudo counter is `IoverI0` useful for normalizing the measurement results in order to make them comparable.

In order to translate the counter values into the pseudo counter values, calculations have to be performed. The device pool provides *`PseudoCounterController`* class that can be overwritten to provide new calculations.

The pseudo counter value gets updated automatically every time one of its counters value gets updated e.g. when the acquisition is in progress.

Each pseudo counter is represented by a Tango[143] device whose interface allows to obtain a calculation result (scalar value).

See also:

[143] http://www.tango-controls.org

Pseudo counter API reference the pseudo counter *API*

`PseudoCounter` the pseudo counter tango device *API*

Measurement group overview

The measurement group interface allows the user to access several data acquisition channels at the same time. The measurement group is the key interface to be used when acquiring the data. The Pool can have several measurement groups and use them simultaneously. When creating a measurement group, the user compose it from:

- *Counter/Timer*
- *0D*
- *1D*
- *2D*
- *Pseudo Counter*
- external attribute e.g. Tango[144]

It is not possible to have several times the same channel in a measurement group.

Configuration

In order to properly use the measurement group, each of the timerable controllers (Counter/Timer, 1D or 2D) needs to be assigned one of its channels as the timer or the monitor. The first timer or monitor becomes the master one for the whole measurement group.

By default, the data acquisition channels are synchronized by software, meaning that the acquisition will be commanded to start (or start and stop) with the software precission. In order to achieve a better synchonization the hardware triggerring (or gating) can be used by configuring a *Trigger/Gate* as the controller's synchronizer.

The measurement group configuration can by modified with the *expconf widget*.

See also:

Measurement group API reference the measurementent group *API*

`MeasurementGroup` the measurement group tango device *API*

`PoolMeasurementGroup` the measurement group class *API*

Writing macros

Writing macros

This chapter provides the necessary information to write macros in sardana. The complete macro *API* can be found *here*.

[144] http://www.tango-controls.org

What is a macro

A macro in sardana describes a specific procedure that can be executed at any time. Macros run inside the *sardana sandbox*. This simply means that each time you run a macro, the system makes sure the necessary environment for it to run safely is ready.

Macros can only be written in Python[145]. A macro can be a *function* or a *class*. In order for a *function* to be recognized as a macro, it **must** be properly *labeled* as a macro (this is done with a special `macro` *decorator*. Details are explaind below). In the same way, for a *class* to be recognized as a macro, it must inherit from a `Macro` super-class. Macros are case sensitive. This means that *helloworld* is a different macro than *HelloWorld*.

The choice between writing a macro *function* or a macro *class* depends not only on the type of procedure you want to write, but also (and probably, most importantly) on the type of programming you are most confortable with.

If you are a scientist, and you have a programming background on a functional language (like fortran, matlab, SPEC[146]), then you might prefer to write macro functions. Computer scientists (young ones, specially), on the other hand, often have a background on object oriented languages (Java, C++, C#) and feel more confortable writing macro classes.

Classes tend to scale better with the size of a program or library. By writing a macro class you can benefit from all advantages of object-oriented programming. This means that, in theory:

- it would reduce the amount of code you need to write
- reduce the complexity of your code y by dividing it into small, reasonably independent and re-usable components, that talk to each other using only well-defined interfaces
- Improvement of productivity by using easily adaptable pre-defined software components

In practice, however, and specially if you don't come from a programming background, writing classes requires a different way of thinking. It will also require you to extend your knowledge in terms of syntax of a programming language.

Furthermore, most tasks you will probably need to execute as macros, often don't fit the class paradigm that object-oriented languages offer. If you are writing a sequencial procedure to run an experiment then you are probably better of writing a python function which does the job plain and simple.

One reason to write a macro as a class is if, for example, you want to extend the behaviour of the `mv` macro. In this case, probably you would want to *extend* the existing macro by writing your own macro class which *inherits* from the original macro and this way benefit from most of the functionallity already existing in the original macro.

What should and should not be a macro

The idea of a macro is simply a piece of Python[147] code that can be executed from control system interface (*GUI*/*CLI*). Therefore, anything that you don't need to be executed by the interface should **NOT** be a macro.

When you have a big library of functions and classes, the approach to expose them to sardana should be to first carefully decide which procedures should be invoked by a *GUI*/*CLI* (namely the name of the procedure, which parameters it should receive and if it returns any value). Then write the macro(s) which invoke the code of the original library (see *Using external python libraries*). Avoid the temptation to convert the functions/classes of the original library into macros because:

[145] http://www.python.org/
[146] http://www.certif.com/
[147] http://www.python.org/

- This will most certainly break your code (any code that calls a function or class that has been converted to a macro will fail)
- It will excessively polute the macro list (imagine a *GUI* with a combo box to select which macro to execute. If you have hundreds of macros it will take forever to find the one to execute even if they are in alphabetical order)

How to start writing a macro

Since macros are essencially Python[148] code, they reside inside a Python[149] file. In sardana, we call a Python[150] file which contains macros a *macro library*.

At the time of writing, the easiest way to create a new macro is from spock (we are currently working on a macro editor *GUI*).

Preparing your text editor

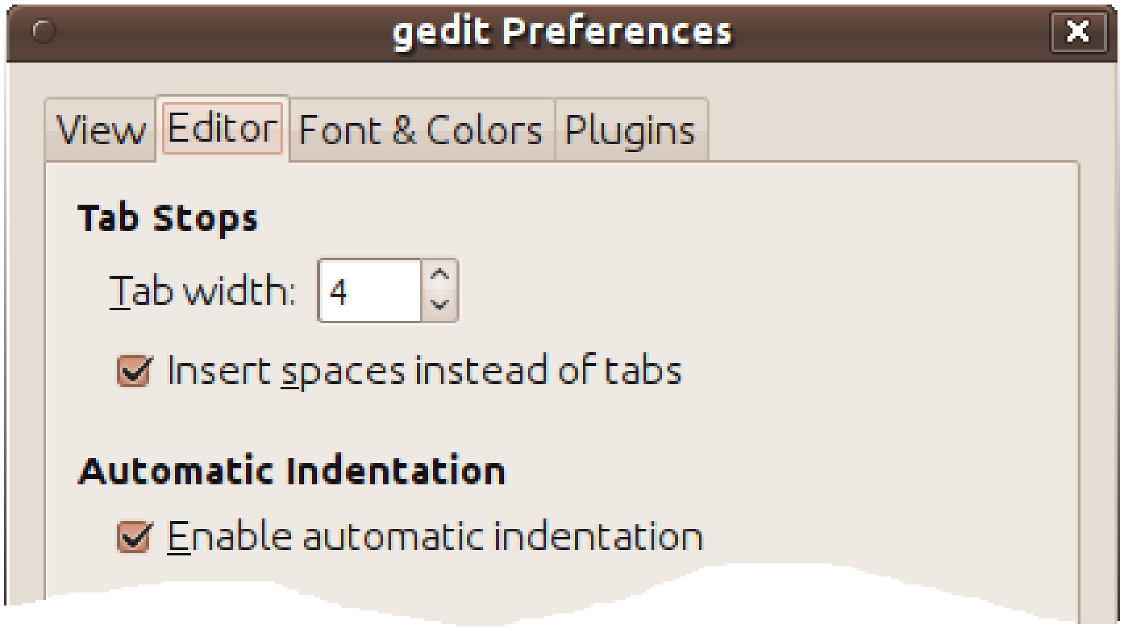
Before launching spock it is important to decide which text editor you will use to write your macros. Unless configured otherwise, spock will use the editor specified by the system environment variable `EDITOR`. If this variable is not set, it will default to vi under Linux/Unix and to notepad under Windows. The following line explains how to set the `EDITOR` environment variable to gedit under linux using bash shell:

```
$ export EDITOR=gedit
```

If you choose *gedit* it is important to properly configure it to write Python[151] code:

Go to *Edit* → *Preferences* → *Editor* and select:

- *Tab width* : 4
- *Insert spaces instead of tabs*



Tab Stops

Tab width: 4

☑ Insert spaces instead of tabs

Automatic Indentation

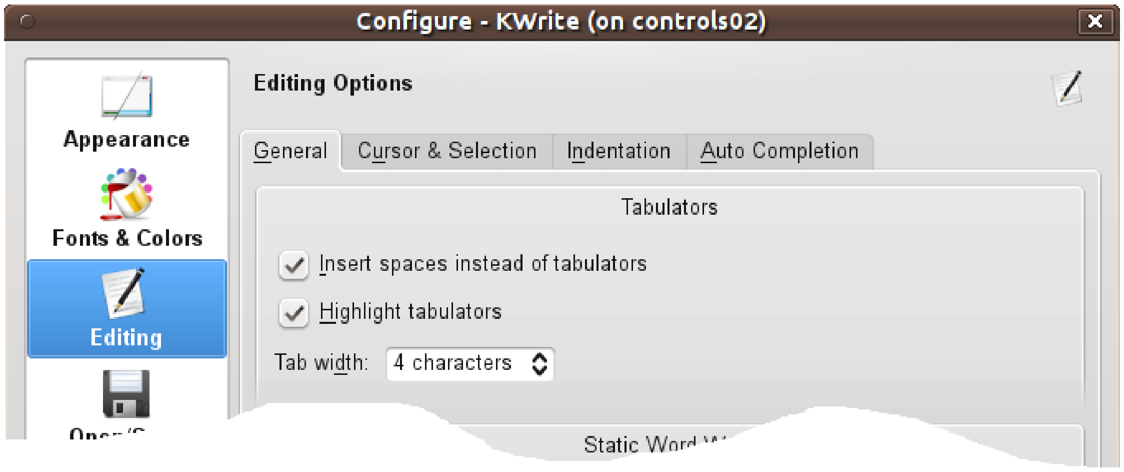
☑ Enable automatic indentation

148 http://www.python.org/
149 http://www.python.org/
150 http://www.python.org/
151 http://www.python.org/

If you choose *kwrite* it is important to properly configure it to write Python[152] code:

Go to *Settings* → *Configure editor…* and choose *Editing*:

- **In *General* tab:**
  - *Tab width* : 4
  - *Insert spaces instead of tabulators*
- **In *Indentation* tab:**
  - *Default indentation mode* : Python
  - *Indentation width* : 4


Configure - KWrite (on controls02)

Editing Options

General | Cursor & Selection | Indentation | Auto Completion

Tabulators

✓ Insert spaces instead of tabulators

✓ Highlight tabulators

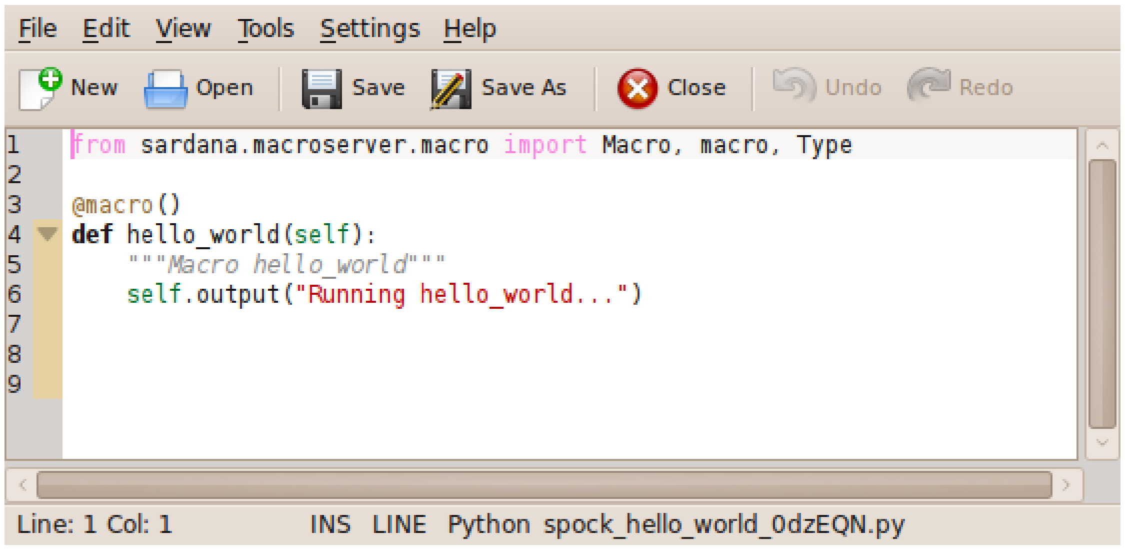
Tab width: 4 characters

Now you are ready to start writing your macro! Type *spock* on the command line. Once you are in spock, you can use the `edmac` to create/edit macros. Let's say you want to create a new macro called *hello_world* in a new macro library called *salute*. Just type in:

```
LAB-01-D01 [1]: edmac hello_world salute
Opening salute.hello_world...
Editing...
```

This will bring your favorite editor to life with a macro function template code for the macro *hello_world*.

[152] http://www.python.org/


File Edit View Tools Settings Help
New Open Save Save As Close Undo Redo

```
from sardana.macroserver.macro import Macro, macro, Type

@macro()
def hello_world(self):
    """Macro hello_world"""
    self.output("Running hello_world...")
```

Line: 1 Col: 1 INS LINE Python spock_hello_world_0dzEQN.py

The next chapter will explain how to fill this template with useful code. After you finish editing the macro, save the file, exit the editor and go back to spock. You'll be asked if you want the new code to be load on the server. Just answer 'y'.

```
LAB-01-D01 [1]: edmac hello_world salute
Openning salute.hello_world...
Editing...
Do you want to apply the new code on the server? [y] y
```

Writing a macro function

As mentioned before, macros are just simple Python[153] functions which have been *labeled* as macros. In Python[154], these labels are called *decorators*. Here is the macro function version of *Hello, World!*:

```
from sardana.macroserver.macro import macro

@macro()
def hello_world(self):
    """This is a hello world macro"""
    self.output("Hello, World!")
```

line 1 imports the *macro* symbol from the sardana macro package. `sardana.macroserver.macro` is the package which contains most symbols you will require from sardana to write your macros.

line 3 this line *decorates* de following function as a macro. It is **crucial** to use this decorator in order for your *function* to be recognized by sardana as a valid macro.

line 4 this line contains the hello_world *function* definition. Every macro needs **at least** one parameter. The first parameter is the macro execution context. It is usually called `self` but you can name it anything. This parameter gives you access to the entire context where the macro is being run. Through it, you'll be able to do all sorts of things, from sending text to the output to ask for motors or even execute other macros.

line 5 Documentation for this macro. You should **always** document your macro!

line 6 this line will print *Hello, World!* on your screen.

[153] http://www.python.org/
[154] http://www.python.org/

Note: If you already know a little about Python[155] your are probably wondering why not use `print "Hello, World!"`?

Remember that your macro will be executed by a Sardana server which may be running in a different computer than the computer you are working on. Executing a *normal print* would just print the text in the server. Therefore you need to explicitly say you want the text on the computer you are working and not the server. The way to do it is using `output()` instead of print.

If you prefer, you can use the context version of Python[156] `print()`[157] function (it is a bit more powerful than `output()`, and has a slightly different syntax)

```python
# mandatory first line in your code if you use Python < 3.0
from __future__ import print_function

from sardana.macroserver.macro import macro

@macro()
def hello_world(self):
    """This is an hello world macro"""
    self.print("Hello, World!")
```

The following footnote describes how to discover your Python[158] version[181].

Remeber that a macro is, for all purposes, a normal Python[159] *function*. This means you **CAN** inside a macro write **ANY** valid Python[160] code. This includes `for`[161] and `while`[162] loops, `if`[163] … `elif`[164] … `else`[165] conditional execution, etc…

```python
import numpy.fft

@macro()
def fft_my_wave(self):
    wave_device = self.getDevice("sys/tg_test/1")
    wave = wave_device.wave
    wave_fft = numpy.fft.fft(wave)
```

Adding parameters to your macro

Standard Python[166] allows you to specify parameters to a function by placing comma separated parameter names between the `()` in the function definition. The macro *API*, in adition, enforces you to specify some extra parameter information. At first, this may look like a useless complication, but you will apreciate clear benefits soon enough. Here are some of them:

[155] http://www.python.org/
[156] http://www.python.org/
[157] https://docs.python.org/dev/library/functions.html#print
[158] http://www.python.org/
[181] To check which version of Python you are using type on the command line `python -c "import sys; sys.stdout.write(sys.version)"`
[159] http://www.python.org/
[160] http://www.python.org/
[161] https://docs.python.org/dev/reference/compound_stmts.html#for
[162] https://docs.python.org/dev/reference/compound_stmts.html#while
[163] https://docs.python.org/dev/reference/compound_stmts.html#if
[164] https://docs.python.org/dev/reference/compound_stmts.html#elif
[165] https://docs.python.org/dev/reference/compound_stmts.html#else
[166] http://www.python.org/

- error prevention: a macro will not be allowed to run if the given parameter if of a wrong type
- *CLI*s like Spock will be able to offer autocomplete facilities (press <tab> and list of allowed parameters show up)
- *GUI*s can display list of allowed parameter values in combo boxes which gives increased usability and prevents errors
- Documentation can be generated automatically

So, here is an example on how to define a macro that needs one parameter:

```
@macro([["moveable", Type.Moveable, None, "moveable to get position"]])
def where_moveable(self, moveable):
    """This macro prints the current moveable position"""
    self.output("%s is now at %s", moveable.getName(), moveable.getPosition())
```

Here is another example on how to define a macro that needs two parameters:

- Moveable (motor, pseudo motor)
- Float (motor absolute position to go to)

```
from sardana.macroserver.macro import macro, Type

@macro([ ["moveable", Type.Moveable, None, "moveable to move"],
         ["position", Type.Float, None, "absolute position"] ])
def move(self, moveable, position):
    """This macro moves a moveable to the specified position"""
    moveable.move(position)
    self.output("%s is now at %s", moveable.getName(), moveable.getPosition())
```

The parameter information is a `list`[167] of `list`[168]s. Each `list`[169] being a composed of four elements:

- parameter name
- parameter type
- parameter default value (None means no default value)
- parameter description

Here is a list of the most common allowed parameter types:

- `Integer`: an integer number
- `Float`: a real number
- `Boolean`: a boolean True or False
- `String`: a string
- `Moveable`: a moveable element (motor, pseudo-motor)
- `Motor`: a pure motor
- `ExpChannel`: an experimental channel (counter/timer, 0D, pseudo-counter, . . .)
- `Controller`: a controller
- `ControllerClass`: an existing controller class plugin
- `MacroCode`: a macro

[167] https://docs.python.org/dev/library/stdtypes.html#list
[168] https://docs.python.org/dev/library/stdtypes.html#list
[169] https://docs.python.org/dev/library/stdtypes.html#list

- MeasurementGroup: a measurement group
- Any: anything, really

The complete list of types distributed with sardana is made up by these five simple types: Integer, Float, Boolean, String, Any, plus all available sardana interfaces (*Interface*)

Repeat parameters

A special parameter type is the repeat parameter (a.k.a. *ParamRepeat*, originating from the ParamRepeat class which usage is deprecated). The repeat parameter type is a list of parameter members. It is possible to pass from zero to multiple repetitions of the repeat parameter items at the execution time.

The repeat parameter definition allows to:

- restrict the minimum and/or maximum number of repetitions
- nest repeat parameters inside of another repeat parameters
- define multiple repeat parameters in the same macro

Repeat parameter values are passed to the macro function in the form of a list. If the repeat parameter definition contains just one member it is a plain list of items.

```
@macro([["moveables", [
             ["moveable", Type.Moveable, None, "moveable to get position"]
             ],
             None, "list of moveables to get positions"]])
def where_moveables(self, moveables):
    """This macro prints the current moveables positions"""
    for moveable in moveables:
        self.output("%s is now at %s", moveable.getName(), moveable.getPosition())
```

But if the repeat parameter definition contains more than one member each item is an internal list of the members.

```
@macro([["m_p_pairs", [
             ["moveable", Type.Moveable, None, "moveable to be moved"],
             ["position", Type.Float, None, "absolute position"]
             ],
             None, "list of moveables and positions to be moved to"]])
def move_multiple(self, m_p_pairs):
    """This macro moves moveables to the specified positions"""
    for moveable, position in m_p_pairs:
        moveable.move(position)
        self.output("%s is now at %s", moveable.getName(), moveable.getPosition())
```

A set of macro parameter examples can be found *here*.

Macro context

One of the most powerfull features of macros is that the entire context of sardana is at your disposal. Simply put, it means you have access to all sardana elements by means of the first parameter on your macro (you can give this parameter any name but usually, by convention it is called self).

self provides access to an extensive catalog of functions you can use in your macro to do all kinds of things. The complete catalog of functions can be found *here*.

Let's say you want to write a macro that explicitly moves a known *theta* motor to a certain position. You could write a macro which receives the motor as parameter but that would be a little silly since you already know beforehand which motor you will move. Instead, a better solution would be to *ask* sardana for a motor named "theta" and use it directly. Here is how you can acomplish that:

```python
@macro([["position", Type.Float, None, "absolute position"]])
def move_theta(self, position):
    """This macro moves theta to the specified position"""
    th = self.getMotor("th")
    th.move(position)
    self.output("Motor ended at %s", moveable.getPosition())
```

Calling other macros from inside your macro

One of the functions of the macro decorator is to pass the *knowledge* of all existing macros to your macro. This way, without any special imports, your macro will *know* about all other macros on the system even if they have been written in other files.

Lets recreate the two previous macros (*where_moveable* and *move*) to execute two of the macros that exist in the standard macro catalog (`wm` and `mv`)

Here is the new version of *where_moveable*

```python
@macro([["moveable", Type.Moveable, None, "moveable to get position"]])
def where_moveable(self, moveable):
    """This macro prints the current moveable position"""
    self.wm([moveable]) # self.wm(moveable) backwards compatibility - see note
```

... and the new version of *move*

```python
@macro([ ["moveable", Type.Moveable, None, "moveable to move"],
         ["position", Type.Float, None, "absolute position"] ])
def move(self, moveable, position):
    """This macro moves a moveable to the specified position"""
    self.mv([moveable, position]) # self.mv(moveable, position) backwards␣
↪compatibility - see note
    self.output("%s is now at %s", moveable.getName(), moveable.getPosition())
```

Note: Both `wm` and `mv` use *repeat parameters*. From Sardana 2.0 the repeat parameter values must be passed as lists of items. An item of a repeat parameter containing more than one member is a list. In case when a macro defines only one repeat parameter and it is the last parameter, for the backwards compatibility reasons, the plain list of items' members is allowed.

Accessing environment

The sardana server provides a global space to store variables, called *environment*. The *environment* is a dictionary[170] storing a value for each variable. This *environment* is stored persistently so if the sardana server is restarted the environment is properly restored.

Variables are case sensitive.

[170] https://docs.scipy.org/doc/numpy/glossary.html#term-dictionary

The value of an existing environment variable can be accessed using *getEnv()*. Setting the value of an environment variable is done with *setEnv()*.

For example, we know the ascan macro increments a `ScanID` environment variable each time it is executed. The following example executes a scan and outputs the new `ScanID` value:

```
@macro([["moveable", Type.Moveable, None, "moveable to get position"]])
def fixed_ascan(self, moveable):
    """This does an ascan starting at 0 ending at 100, in 10 intervals
    with integration time of 0.1s"""

    self.ascan(moveable, 0, 100, 10, 0.1)
    scan_id = self.getEnv('ScanID')
    self.output("ScanID is now %d", scan_id)
```

Logging

The Macro *API* includes a set of methods that allow you to write log messages with different levels:

- *debug()*
- *info()*
- *warning()*
- *error()*
- *critical()*
- *log()*
- *output()*

As you've seen, the special *output()* function has the same effect as a print statement (with slightly different arguments).

Log messages may have several destinations depending on how your sardana server is configured. At least, one destination of each log message is the client(s) (spock, GUI, other) which are connected to the server. Spock, for example, handles the log messages by printing to the console with different colours. By default, spock prints all log messages with level bigger than *debug()* (You can change this behaviour by typing `debug on` in spock). Another typical destination for log messages is a log file.

Here is an example on how to write a logging information message:

```
@macro()
def lets_log(self):
    self.info("Starting to execute %s", self.getName())
    self.output("Hello, World!")
    self.info("Finished to executing %s", self.getName())
```

Reports

Once the report facility has been properly configured, report messages can be sent to the previously configured report file.

There are several differences between *reporting* and *logging*. The first difference is that log messages may or may not be recorded, depending on the configured filters on the target (example: log file). A report will always be recorded.

Another difference is that report messages are not sent to the clients. The idea of a report is to silently record in a file that something as happened.

A third difference is that unlike logs, reports have no message level associated to them (actually since internally the log library is used to report messages, every report record as the predefined level *INFO* but this is just an implementation detail).

A report message can be emited at any time in the macro using the `report()` method:

```
@macro()
def lets_report(self):
    self.report("this is an official report of macro '%s'", self.getName())
```

This would generate the following report message in the report file:

> INFO 2012-07-18 09:39:34,943: this is an official report of macro 'lets_report'

Advanced macro calls

As previously explained (see *calling macros*), you can use the Macro *API* to call other macros from inside your own macro:

```
@macro([["moveable", Type.Moveable, None, "moveable to get position"]])
def fixed_ascan(self, moveable):
    """This does an ascan starting at 0 ending at 100, in 10 intervals
    with integration time of 0.1s"""
    self.ascan(moveable, 0, 100, 10, 0.1)
```

An explicit call to `execMacro()` would have the same effect:

```
@macro([["moveable", Type.Moveable, None, "moveable to get position"]])
def fixed_ascan(self, moveable):
    """This does an ascan starting at 0 ending at 100, in 10 intervals
    with integration time of 0.1s"""
    self.execMacro('ascan', moveable, '0', '100', '10', '0.2')
```

The advantage of using `execMacro()` is that it supports passing parameters with different *flavors*:

- parameters as strings:

  ```
  self.execMacro('ascan', motor.getName(), '0', '100', '10', '0.2')
  self.execMacro('mv', [[motor.getName(), '0']])
  self.execMacro('mv', motor.getName(), '0') # backwards compatibility - see note
  ```

- parameters as space separated string:

  ```
  self.execMacro('ascan %s 0 100 10 0.2' % motor.getName())
  self.execMacro('mv [[%s 0]]' % motor.getName())
  self.execMacro('mv %s 0' % motor.getName()) # backwards compatibility - see note
  self.execMacro('mv [[%s 0][%s 20]]' % (motor.getName(), motor2.getName()))
  self.execMacro('mv %s 0 %s 20' % (motor.getName(), motor2.getName())) # backwards␣
  ↪compatibility - see note
  ```

- parameters as concrete types:

  ```
  self.execMacro(['ascan', motor, 0, 100, 10, 0.2])
  self.execMacro(['mv', [[motor, 0]]])
  self.execMacro(['mv', motor, 0]) # backwards compatibility - see note
  ```

Note: Macro *mv* use *repeat parameters*. From Sardana 2.0 the repeat parameter values must be passed as lists of items. An item of a repeat parameter containing more than one member is a list. In case when a macro defines only one repeat parameter and it is the last parameter, for the backwards compatibility reasons, the plain list of items' members is allowed.

Accessing macro data

Sometimes it is desirable to access data generated by the macro we just called. For these cases, the Macro *API* provides a pair of low level methods `createMacro()` and `runMacro()` together with `data()`.

Let's say that you need access to the data generated by a scan. First you call `createMacro()` with the same parameter you would give to `execMacro()`. This will return a tuple composed from a macro object and the result of the `prepare()` method. Afterward you call `runMacro()` giving as parameter the macro object returned by `createMacro()`. In the end, you can access the data generated by the macro using `data()`:

```
@macro([["moveable", Type.Moveable, None, "moveable to get position"]])
def fixed_ascan(self, moveable):
    """This runs the ascan starting at 0 ending at 100, in 10 intervals
    with integration time of 0.1s"""

    ret = self.createMacro('ascan', moveable, '0', '100', '10', '0.2')
    # createMacro returns a tuple composed from a macro object
    # and the result of the Macro.prepare method
    my_scan, _ = ret
    self.runMacro(my_scan)
    print len(my_scan.data)
```

A set of macro call examples can be found *here*.

Writing a macro class

This chapter describes an advanced alternative to writing macros as Python[171] classes. If words like *inheritance*, *polimorphism* sound like a lawyer's horror movie then you probably should only read this if someone expert in sardana already told you that the task you intend to do cannot be accomplished by writing macro functions.

The simplest macro class that you can write **MUST** obey the following rules:

- Inherit from `Macro`
- Implement the `run()` method

The `run()` method is the place where you write the code of your macro. So, without further delay, here is the *Hello, World!* example:

```
from sardana.macroserver.macro import Macro

class HelloWorld(Macro):
    """Hello, World! macro"""

```

(continues on next page)

[171] http://www.python.org/

(continued from previous page)

```
    def run(self):
        print "Hello, World!"
```

Let's say you want to pass an integer parameter to your macro. All you have to do is declare the parameter by using the *param_def* Macro member:

```
from sardana.macroserver.macro import Macro, Type

class twice(Macro):
    """Macro twice. Prints the double of the given value"""

    param_def = [ [ "value", Type.Float, None, "value to be doubled" ] ]

    def run(self, value):
        self.output(2*value)
```

Note: As soon as you add a *param_def* you also need to modify the *run()* method to support the new paramter(s).

A set of macro parameter examples can be found *here*.

Preparing your macro for execution

Additionaly to the *run()* method, you may write a *prepare()* method where you may put code to prepare the macro for execution (for example, checking pre-conditions for running the macro). By default, the prepare method is an empty method. Here is an example on how to prepare HelloWorld to run only after year 1989:

```
import datetime
from sardana.macroserver.macro import Macro

class HelloWorld(Macro):
    """Hello, World! macro"""

    def prepare(self):
        if datetime.datetime.now() < datetime.datetime(1990,01,01):
            raise Exception("HelloWorld can only run after year 1989")

    def run(self):
        print "Hello, World!"
```

Handling macro stop and abort

While macro is being executed the user has a possibility to stop or abort it at any time. The way of performing this operation may vary between the client application being used, for example see *Stopping macros* in Spock.

It may be desired to stop or abort objects in use when the macro execution gets interrupted. This does not need to be implemented by each of the macros. One simply needs to use a special *API* to reserve this objects in the macro context and the magic will happen to stop or abort them when needed. For example if you get an object using the *getObj()* method it is automatically reserved. It is also worth mentioning the difference between the `getMotion()` and `getMotor()` methods. The first one will reserve the moveable while the second will not.

And of course we need to clarify the difference between the stop and the abort operations. It is commonly agreed that stop is more gentle and respectful than abort and is supposed to bring the system to a stable state according to a particular protocol and respecting a nominal configuration e.g. stop motors respecting the nominal deceleration time. While abort is foreseen for more emergency situations and is supposed to bring the system to a stable state as fast as possible e.g. stop motors instantly, possibly loosing track of position.

While the reserved objects are stopped or aborted immediately after user's interruption, the macro execution is not. The macro will not stop until its execution thread encounters the next *Macro API* call e.g. `output()` or `getMotor()`. There is also a special method `checkPoint()` that does nothing else but mark this place in your code as suitable for stopping or aborting.

If you want to execute a special procedure that should be executed in case of user's interruption you must override the `on_stop()` or `on_abort()` methods.

Note: Currently it is not possible to use any of the *Macro API* calls withing the `on_stop()` or `on_abort()`.

Using external python libraries

Macro libraries can use code e.g. call functions and instantiate classes defined by external python libraries. In order to import the external libraries inside the macro library, they must be available for the python interpreter running the Sardana/MacroServer server (see *Running server*).

This could be achieved in two ways:

- Adding the directory containing the external library to the *PythonPath* property of the MacroServer tango device (path separators can be `\n` or `:`).
- Adding the directory containing the external library to the *PYTHONPATH* *OS* environment variable of the Sardana/MacroServer process.

The external libraries can be reloaded at Sardana/MacroServer server runtime using the `rellib` macro.

Plotting

Remember that your macro will be executed by a Sardana server which may be running in a different computer than the computer you are working on. Executing a normal plot (from `matplotlib` or `guiqwt`[172]) would just try to show a plot in the server machine. The macro *API* provides a way to plot graphics from inside your macro whenver the client that runs the macro *understands* the plot request (don't worry, spock does understand!)

The plotting *API* is the same used by `pyplot`[173]. The *API* is accessible through the macro context (`self`). Here is an example:

```python
import math
from numpy import linspace
from scipy.integrate import quad
from scipy.special import j0

```

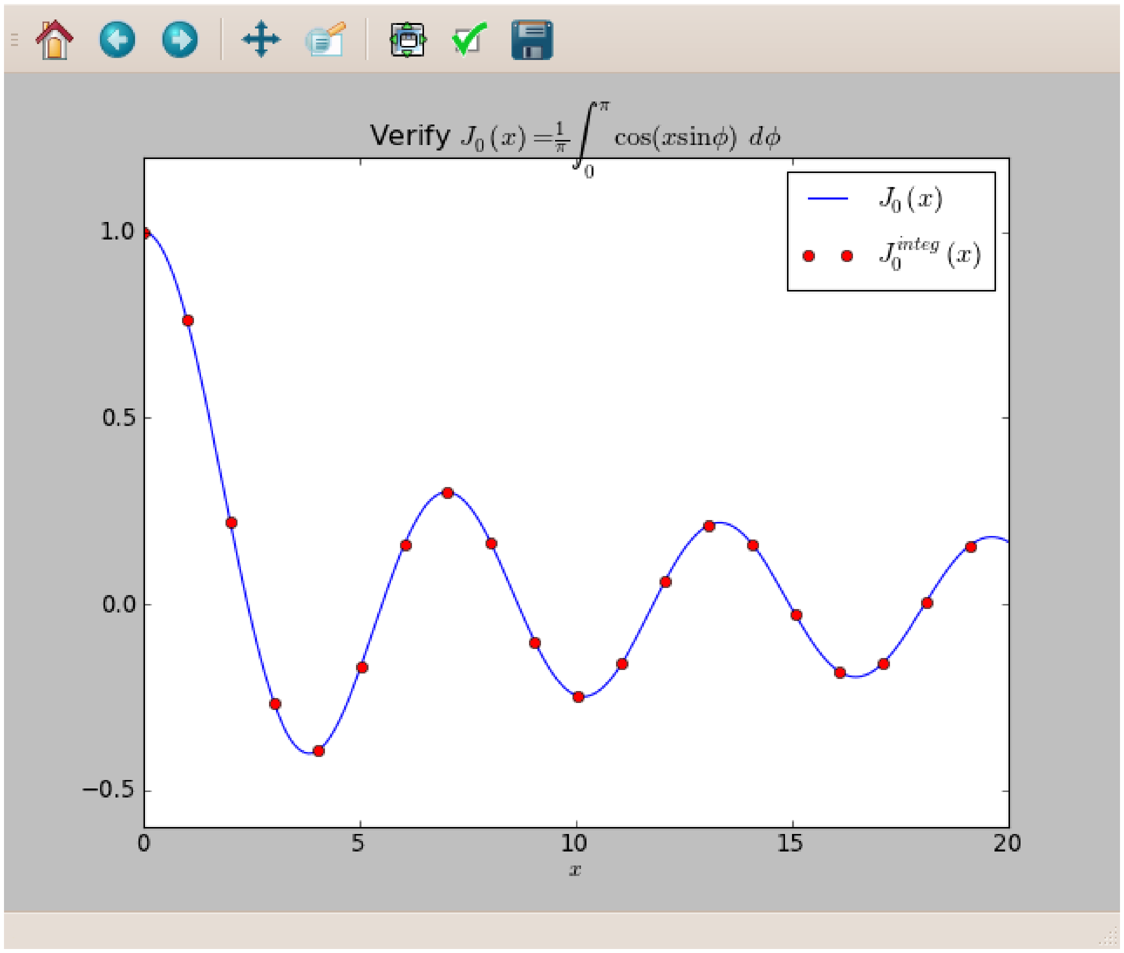
(continues on next page)

[172] https://pythonhosted.org/guiqwt/index.html#module-guiqwt
[173] https://matplotlib.org/api/_as_gen/matplotlib.pyplot.html#module-matplotlib.pyplot

(continued from previous page)

```python
from sardana.macroserver.macro import macro

def j0i(x):
    """Integral form of J_0(x)"""
    def integrand(phi):
        return math.cos(x * math.sin(phi))
    return (1.0/math.pi) * quad(integrand, 0, math.pi)[0]

@macro()
def J0_plot(self):
    """Sample J0 at linspace(0, 20, 200)"""
    x = linspace(0, 20, 200)
    y = j0(x)
    x1 = x[::10]
    y1 = map(j0i, x1)
    self.pyplot.plot(x, y, label=r'$J_0(x)$') #
    self.pyplot.plot(x1, y1, 'ro', label=r'$J_0^{integ}(x)$')
    self.pyplot.title(r'Verify $J_0(x)=\frac{1}{\pi}\int_0^{\pi}\cos(x \sin\phi)\,
↪d\phi$')
    self.pyplot.xlabel('$x$')
    self.pyplot.legend()
```

Running this macro from spock will result in something like:


Verify $J_0(x) = \frac{1}{\pi}\int_0^{\pi} \cos(x\sin\phi)\ d\phi$
$J_0(x)$
$J_0^{integ}(x)$
1.0
0.5
0.0
−0.5
0
5
10
15
20
x

Just for fun, the following macro computes a fractal and plots it as an image:

```python
import numpy

@macro([["interactions", Type.Integer, None, ""],
        ["density", Type.Integer, None, ""]])
def mandelbrot(self, interactions, density):

    x_min, x_max = -2, 1
    y_min, y_max = -1.5, 1.5

    x, y = numpy.meshgrid(numpy.linspace(x_min, x_max, density),
                          numpy.linspace(y_min, y_max, density))

    c = x + 1j * y
    z = c.copy()

    fractal = numpy.zeros(z.shape, dtype=numpy.uint8) + 255

    finteractions = float(interactions)
    for n in range(interactions):
        z *= z
        z += c
        mask = (fractal == 255) & (abs(z) > 10)
        fractal[mask] = 254 * n / finteractions
    self.pyplot.imshow(fractal)
```

And the resulting image (interactions=20, density=512):

A set of macro plotting examples can be found *here*.

Known plotting limitations

When you plot from inside a macro with `self.pyplot.plot`, the sardana server will "ask" spock to execute the desired function with the given parameters. This means that the result of plotting (a sequence of `Line2D`) is not available in the sardana server (since the actual line is in spock). The result of any function call in `self.pyplot` will always be None!

This means that the following code which works in a normal IPython[174] console will **NOT** work inside a macro:

```
LAB-01-D01 [1]: line = plot(range(10))[0]

LAB-01-D01 [2]: line.set_linewidth(5)
```

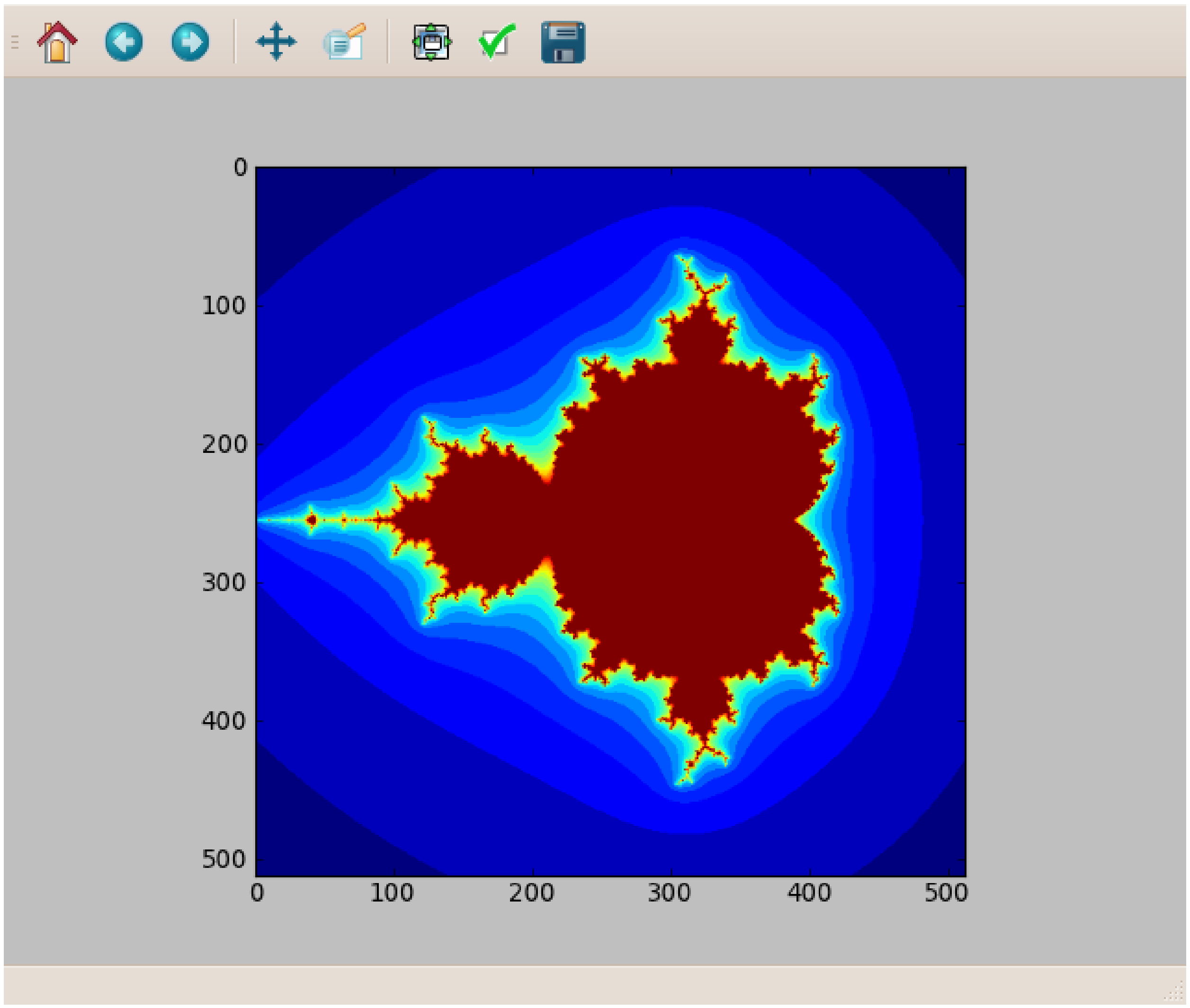
Also consider that each time you plot the complete data to be plotted is sent from the server to the client... so please avoid plotting arrays of 10,000,000 points!

Asking for user input

It is possible to ask for user input inside a macro.

Hint: Asking for input in the middle of long macros will cause the macro to stop and wait for user input. If you write a long macro that might be executed *in the middle of the night* please take the appropriate steps to

[174] http://ipython.org/


0
100
200
300
400
500
0
100
200
300
400
500

make sure you don't arrive in the morning and you are faced with a message box waiting for you to answer a question that could be avoided with a proper *default value*. To make sure your macro can run in *unattended* mode make sure that:

- it implements the interactive *interface*
- every `input()` gives a *default_value keyword argument*

(read on to see how to meet these requirements)

In pure Python[175], to ask for user input you can use the `raw_input()` (Python 2) / `input()`[176] (Python 3)

```
>>> answer = raw_input('--> ')
--> Monty Python's Flying Circus
>>> answer
"Monty Python's Flying Circus"
```

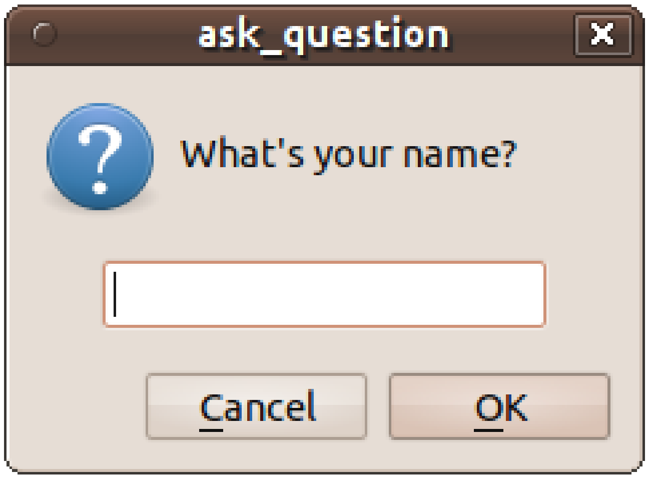
The Macro *API* provides a much more powerful version of `input()` since it can accept a wide variaty of options.

Similar to what happens with *Plotting*, when input is requested from inside a macro, the question will be sent to the client (example: spock) which ordered the macro to be executed. At this time the macro is stopped waiting for the client to answer. The client must "ask" the user for a proper value and the answer is sent back to the server which then resumes the macro execution.

Asking for user input is straightforward:

```
@macro()
def ask_name(self):
    """Macro function version to ask for user name"""

    answer = self.input("What's your name?")
    self.output("So, your name is '%s'", answer)
```

Executing this macro will make spock popup an Input Dialog Box like this one:



When you type your name and press OK the macro finishes printing the output:

[175] http://www.python.org/
[176] https://docs.python.org/dev/library/functions.html#input

```
LAB-01-D01 [1]: ask_name
Non interactive macro 'ask_name' is asking for input (please set this macro␣
↪interactive to True)
So, your name is 'Homer Simpson'
```

The macro prints a warning message saying that the macro was not declared as *interactive*. All macros that request user input **should** be declared as interactive. This is because the sardana server can run a macro in *unattended* mode. When an interactive macro is run in *unattended* mode, all `input()` instructions that have a default value will return automatically the default value without asking the user for input.

To declare a macro as interactive set the `interactive` *keyword argument* in the macro decorator to `True` (default value for `interactive` is `False`), like this:

```
@macro(interactive=True)
def ask_name(self):
    """Macro function version to ask for user name"""

    answer = self.input("What's your name?")
    self.output("So, your name is '%s'", answer)
```

To declare a macro class as interactive set the `interactive` member to `True` (default value for `interactive` is `False`), like this:

```
class ask_name(Macro):
    """Macro class version to ask for user name"""

    interactive = True

    def run(self):
        answer = self.input("What's your name?")
        self.output("So, your name is '%s'", answer)
```

a helper `imacro` decorator and a `iMacro` class exist which can be used instead of the `macro` decorator and `Macro` class to transparently declare your macro as interactive:

```
from sardana.macroserver.macro import imacro, iMacro

# interactive macro function version

@imacro()
def ask_name(self):
    """Macro function version to ask for user name"""

    answer = self.input("What's your name?")
    self.output("So, your name is '%s'", answer)

# interactive macro class version

class ask_name(iMacro):
    """Macro class version to ask for user name"""

    def run(self):
        answer = self.input("What's your name?")
        self.output("So, your name is '%s'", answer)
```

The following sub-chapters explain the different options available for macro user input.

Specifying input data type

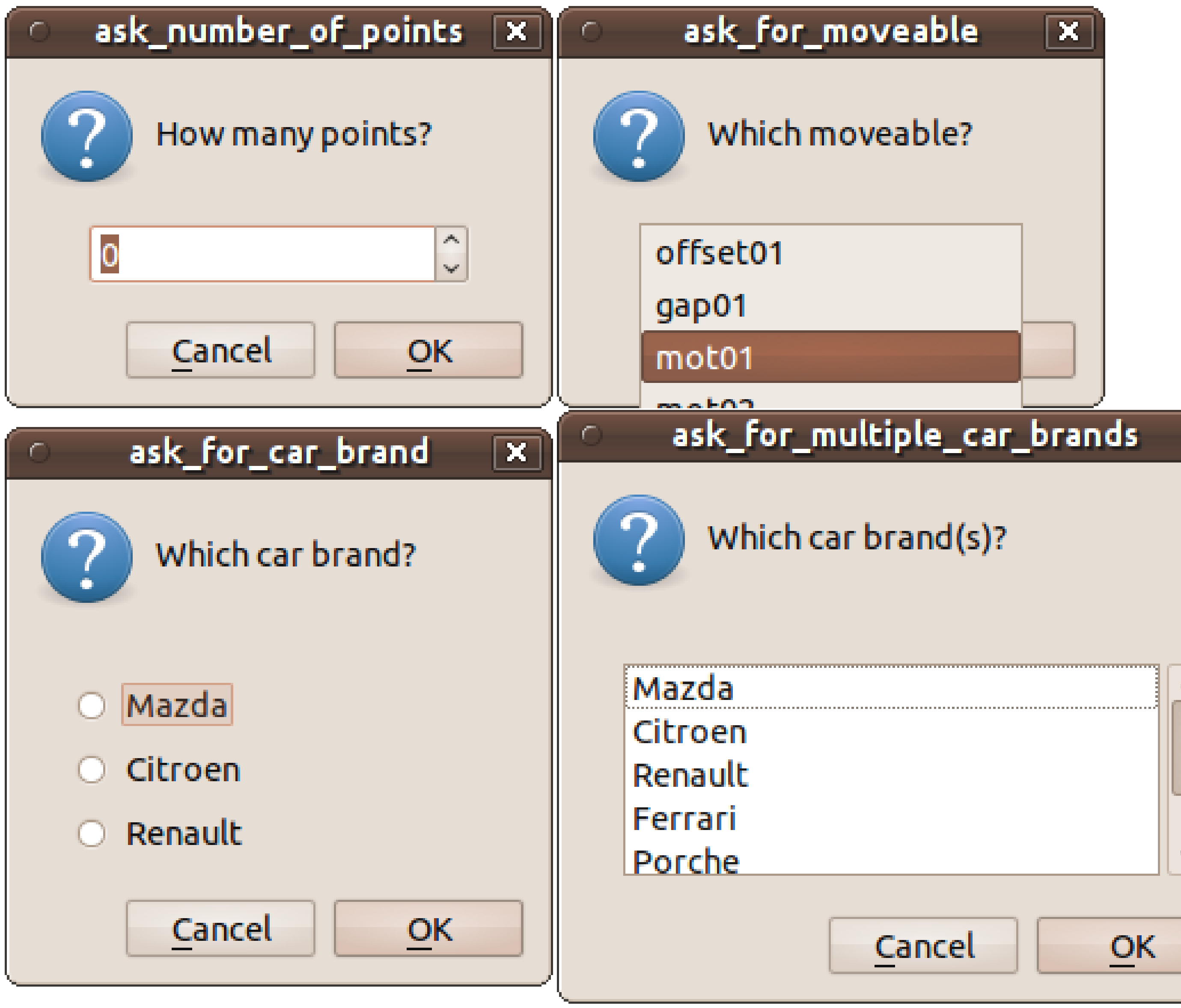
The default return type of *input* is `str`[177] which mimics the pure Python[178] input function. However, often you want to restrict the user input to a specific data type like `Integer`, `Float` or even complex object like `Moveable` or to a list of possible options.

The macro *input API* provides an easy way to do this by specifying the concrete data type in the *keyword argument data_type*. The following examples shows how to ask for an `Integer`, a `Moveable`, and single/multiple selection from a list of options:

```python
from sardana.macroserver.macro import imacro, Type

@imacro()
def ask_number_of_points(self):
    """asks user for the number of points"""

    nb_points = self.input("How many points?", data_type=Type.Integer)
    self.output("You selected %d points", nb_points)

@imacro()
def ask_for_moveable(self):
    """asks user for a motor"""

    moveable = self.input("Which moveable?", data_type=Type.Moveable)
    self.output("You selected %s which is at %f", moveable, moveable.getPosition())

@imacro()
def ask_for_car_brand(self):
    """asks user for a car brand"""

    car_brands = "Mazda", "Citroen", "Renault"
    car_brand = self.input("Which car brand?", data_type=car_brands)
    self.output("You selected %s", car_brand)

@imacro()
def ask_for_multiple_car_brands(self):
    """asks user for several car brands"""

    car_brands = "Mazda", "Citroen", "Renault", "Ferrari", "Porche", "Skoda"
    car_brands = self.input("Which car brand(s)?", data_type=car_brands,
                            allow_multiple=True)
    self.output("You selected %s", ", ".join(car_brands))
```

... and these are the corresponding dialogs that will popup in spock:

[177] https://docs.python.org/dev/library/stdtypes.html#str
[178] http://www.python.org/



Providing a default value

Providing a default value is **very important** since it will allow your macro to run in *unattended* mode. When given, the *default_value* *keyword argument* value type must be compatible with the *data_type* *keyword argument*. Providing a default value is easy. The following examples repeat the previous data type examples giving compatible default values:

```python
from sardana.macroserver.macro import imacro, Type

@imacro()
def ask_number_of_points(self):
    """asks user for the number of points"""

    nb_points = self.input("How many points?", data_type=Type.Integer,
                           default_value=100)
    self.output("You selected %d points", nb_points)
```

(continues on next page)

(continued from previous page)

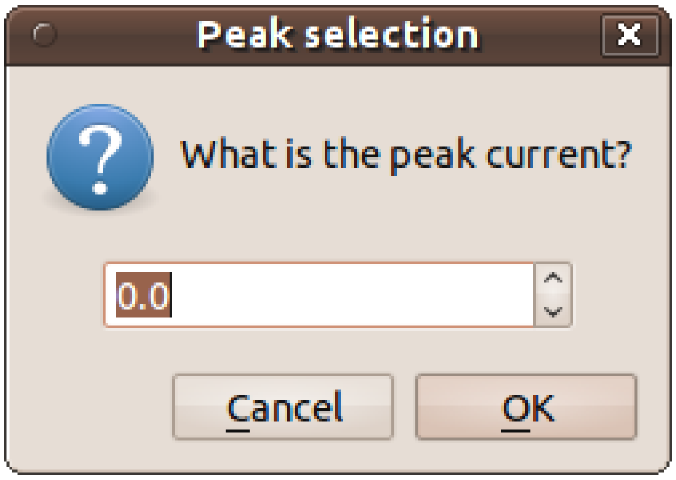
```python

@imacro()
def ask_for_moveable(self):
    """asks user for a motor"""

    moveable = self.input("Which moveable?", data_type=Type.Moveable,
                          default_value="gap01")
    self.output("You selected %s which is at %f", moveable, moveable.getPosition())

@imacro()
def ask_for_car_brand(self):
    """asks user for a car brand"""

    car_brands = "Mazda", "Citroen", "Renault"
    car_brand = self.input("Which car brand?", data_type=car_brands,
                           default_value=car_brands[1])
    self.output("You selected %s", car_brand)

@imacro()
def ask_for_multiple_car_brands(self):
    """asks user for several car brands. Default is every other car brand
    in the list"""

    car_brands = "Mazda", "Citroen", "Renault", "Ferrari", "Porche", "Skoda"
    car_brands = self.input("Which car brand(s)?", data_type=car_brands,
                            allow_multiple=True,
                            default_value=car_brands[::2])
    self.output("You selected %s", ", ".join(car_brands))
```

Giving a title

By default, the Dialog window title will contain the name of the macro which triggered user input. You can override the default behaviour with the *keyword argument title*:

```python
@imacro()
def ask_peak(self):
    """asks use for peak current of points with a custom title"""

    peak = self.input("What is the peak current?", data_type=Type.Float,
                      title="Peak selection")
    self.output("You selected a peak of %f A", peak)
```

. . . and this is the corresponding dialog:

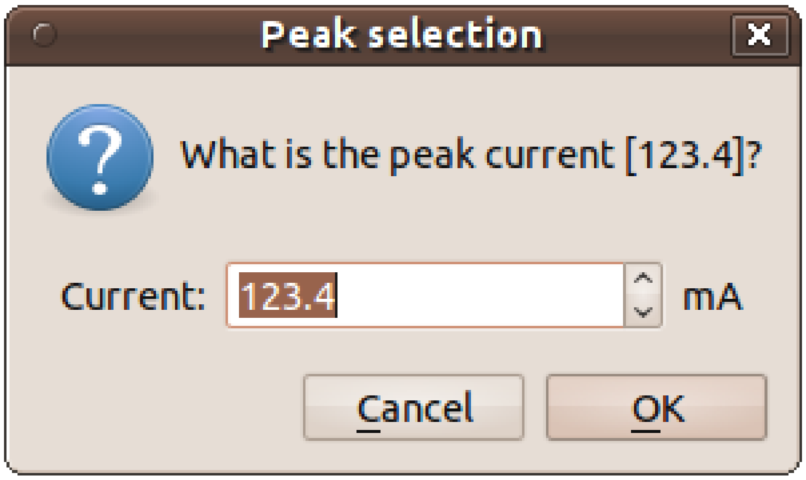


Specifying label and unit

The *key* and *unit* *keyword arguments* can be used to provide additional label and unit information respectively and prevent user mistakes:

```python
@imacro()
def ask_peak_v2(self):
    """asks use for peak current of points with a custom title,
    default value, label and units"""

    label, unit = "peak", "mA"
    peak = self.input("What is the peak current?", data_type=Type.Float,
                      title="Peak selection", key=label, unit=unit,
                      default_value=123.4)
    self.output("You selected a %s of %f %s", label, peak, unit)
```

. . . and this is the corresponding dialog:



Limiting ranges, setting decimal places and step size

When numeric input is requested, it might be useful to prevent user input outside a certain range. This can be achieved with the *minimum* and *maximum* *keyword arguments*:

```python
@imacro()
def ask_peak_v3(self):
    """asks use for peak current of points with a custom title,
    default value, label, units and ranges"""

    label, unit = "peak", "mA"
    peak = self.input("What is the peak current?", data_type=Type.Float,
                      title="Peak selection", key=label, unit=unit,
                      default_value=123.4, minimum=0.0, maximum=200.0)
    self.output("You selected a %s of %f %s", label, peak, unit)
```

An additional *step* *keyword argument* may help increase usability by setting the step size in a input spin box:

```python
@imacro()
def ask_peak_v4(self):
    """asks use for peak current of points with a custom title,
    default value, label, units, ranges and step size"""

    label, unit = "peak", "mA"
    peak = self.input("What is the peak current?", data_type=Type.Float,
                      title="Peak selection", key=label, unit=unit,
                      default_value=123.4, minimum=0.0, maximum=200.0,
                      step=5)
    self.output("You selected a %s of %f %s", label, peak, unit)
```

When asking for a decimal number, it might be useful to use the *decimals* *keyword argument* to indicate how many decimal places to show in a input spin box:

```python
@imacro()
def ask_peak_v5(self):
    """asks use for peak current of points with a custom title,
    default value, label, units, ranges, step size and decimal places"""

    label, unit = "peak", "mA"
    peak = self.input("What is the peak current?", data_type=Type.Float,
                      title="Peak selection", key=label, unit=unit,
                      default_value=123.4, minimum=0.0, maximum=200.0,
                      step=5, decimals=2)
    self.output("You selected a %s of %f %s", label, peak, unit)
```

A set of macro input examples can be found *here*.

Showing progress in long macros

Some of the macros you write may take a long time to execute. It could be useful to provide frequent feedback on the current progress of your macro to prevent users from thinking the system is blocked. The way to do this is by `yield`[179]ing a new progress number in the ode everytime you want to send a progress.

The following code shows an example:

```python
import time

@macro([["duration", Type.Integer, 1, "time to sleep (s)"]])
def nap(self, duration):
```

(continues on next page)

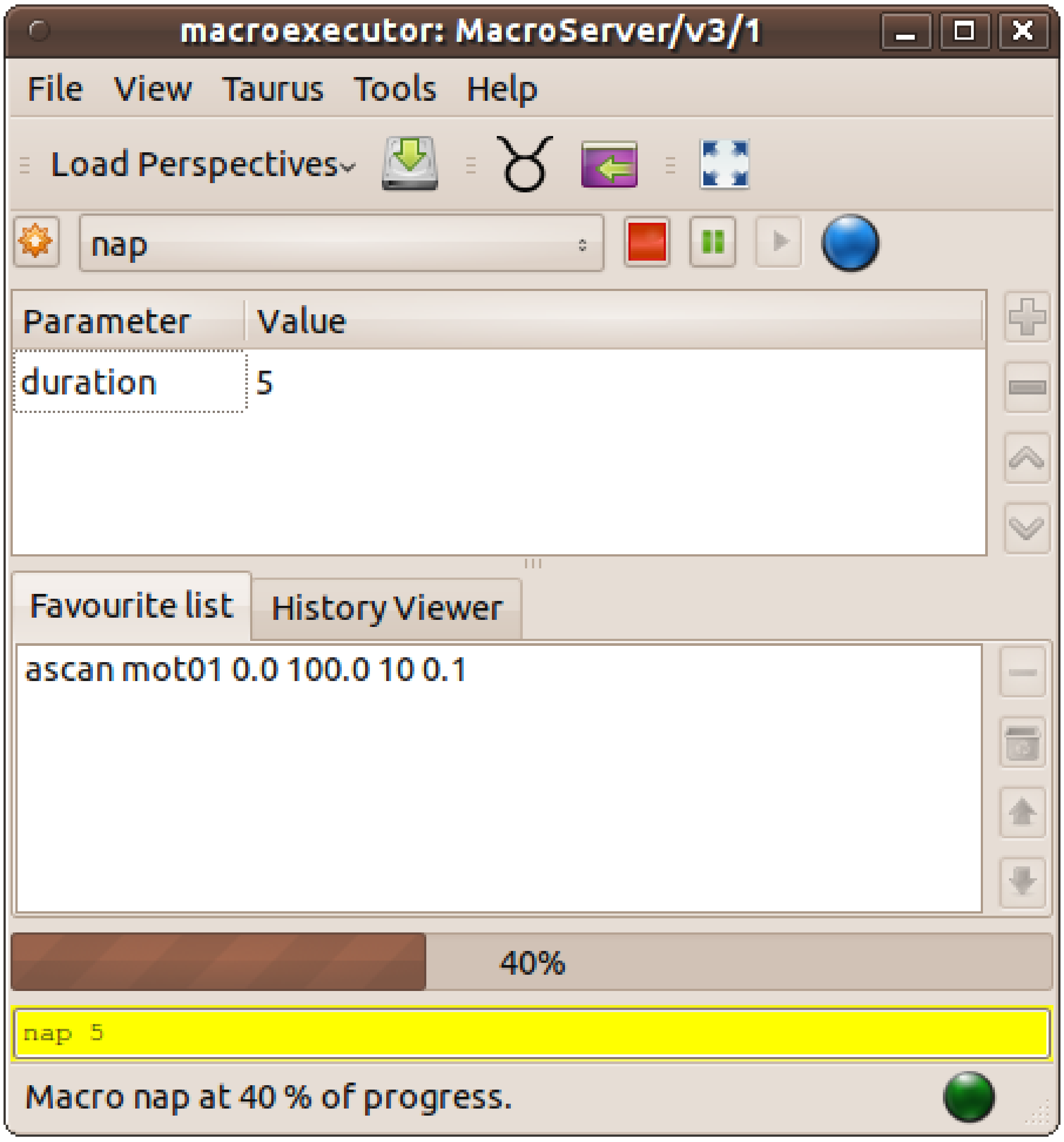
[179] https://docs.python.org/dev/reference/simple_stmts.html#yield

(continued from previous page)

```
    fduration = float(duration)
    for i in range(duration):
        time.sleep(1)
        yield (i+1) / fduration * 100
```

The important code here is line 9. Everytime the macro execution reaches this line of code, basically it tells sardana to send a progress with the desired value. By default, the value is interpreted has a percentage and should have the range between 0.0 and 100.0.

Actually, even if your macro doesn't explicitly send macro progress reports, sardana always generates a 0.0 progress at the beginning of the macro and a last 100.0 progress at the end so for example, in a *GUI*, the progress bar showing the macro progress will always reach the end (unless an error occurs) no matter how you program the progress.



Parameter	Value
duration	5

ascan mot01 0.0 100.0 10 0.1

40%

nap 5

Macro nap at 40 % of progress.

It is possible to generate a progress that doesn't fit the 0 - 100.0 range. The above macro has been modified to send a progress with a customized range:

```python
import time

@macro([["duration", Type.Integer, 1, "time to sleep (s)"]])
def nap(self, duration):

    status = { 'range' : [0, duration] }

    fduration = float(duration)
    for i in range(duration):
        time.sleep(1)
```

(continues on next page)

(continued from previous page)

```
        status['step'] = i+1
        yield status
```

You may notice that this way, the range can be changed dynamically. A progress bar in a *GUI* is programmed to adjust not only the current progress value but also the ranges so it is safe to change them if necessary.

Scan Framework

In general terms, we call *scan* to a macro that moves one or more motors and acquires data along the path of the motor(s). See the *introduction to the concept of scan in Sardana*.

While a scan macro could be written from scratch, Sardana provides a higher- level API (the *scan framework*) that greatly simplifies the development of scan macros by taking care of the details about synchronization of motors and of acquisitions.

The scan framework is implemented in the `scan` module, which provides the `GScan` base class and its specialized derived classes `SScan` and `CScan` for step and continuous scans, respectively.

Creating a scan macro consists in writing a generic macro (see *the generic macro writing instructions*) in which an instance of `GScan` is created (typically in the `prepare()` method) which is then invoked in the `run()` method.

Central to the scan framework is the `generator()` function, which must be passed to the GScan constructor. This generator is a function that allows to construct the path of the scan (see `GScan` for detailed information on the generator).

A basic example on writing a step scan

Step scans are built using an instance of the `SScan` class, which requires a step generator that defines the path for the motion. Since in a step scan the data is acquired at each step, the generator controls both the motion and the acquisition.

Note that in general, the generator does not need to generate a determinate (or even finite) number of steps. Also note that it is possible to write generators that vary their current step based on the acquired values (e.g., changing step sizes as a function of some counter reading).

The `ascan_demo` macro illustrates the most basic features of a step scan:

```
class ascan_demo(Macro):
    """
    This is a basic reimplementation of the ascan` macro for demonstration
    purposes of the Generic Scan framework. The "real" implementation of
    :class:`sardana.macroserver.macros.ascan` derives from
    :class:`sardana.macroserver.macros.aNscan` and provides some extra features.
    """

    hints = { 'scan' : 'ascan_demo'} #this is used to indicate other codes that the␣
↪macro is a scan
    env = ('ActiveMntGrp',) #this hints that the macro requires the ActiveMntGrp␣
↪environment variable to be set

    param_def = [
       ['motor',       Type.Moveable, None, 'Motor to move'],
```

(continues on next page)

(continued from previous page)

```
        ['start_pos',  Type.Float,    None, 'Scan start position'],
        ['final_pos',  Type.Float,    None, 'Scan final position'],
        ['nr_interv',  Type.Integer,  None, 'Number of scan intervals'],
        ['integ_time', Type.Float,    None, 'Integration time']
    ]

    def prepare(self, motor, start_pos, final_pos, nr_interv, integ_time, **opts):
        #parse the user parameters
        self.start = numpy.array([start_pos], dtype='d')
        self.final = numpy.array([final_pos], dtype='d')
        self.integ_time = integ_time

        self.nr_points = nr_interv+1
        self.interv_size = ( self.final - self.start) / nr_interv
        self.name='ascan_demo'
        env = opts.get('env',{}) #the "env" dictionary may be passed as an option

        #create an instance of GScan (in this case, of its child, SScan
        self._gScan=SScan(self, generator=self._generator, moveables=[motor], env=env)


    def _generator(self):
        step = {}
        step["integ_time"] =  self.integ_time #integ_time is the same for all steps
        for point_no in xrange(self.nr_points):
            step["positions"] = self.start + point_no * self.interv_size #note that␣
↪this is a numpy array
            step["point_id"] = point_no
            yield step

    def run(self,*args):
        for step in self._gScan.step_scan(): #just go through the steps
            yield step

    @property
    def data(self):
        return self._gScan.data #the GScan provides scan data
```

The `ascan_demo` shows only basic features of the scan framework, but it already shows that writing a step scan macro is mostly just a matter of writing a generator function.

It also shows that the `scan.gscan.GScan.data()` method can be used to provide the needed return value of *`data()`*

A basic example on writing a continuous scan

Continuous scans are built using an instance of the `CScan` class. Since in the continuous scans the acquisition and motion are decoupled, CScan requires two independent generators:

- a *waypoint generator*: which defines the path for the motion in a very similar way as the step generator does for a step scan. The steps generated by this generator are also called "waypoints".
- a *period generator* which controls the data acquisition steps.

Essentially, `CScan` implements the continuous scan as an acquisition loop (controlled by the period generator) nested within a motion loop (controlled by the waypoint generator). Note that each loop is run on an independent thread, and only limited communication occurs between the two (basically the acquisition starts at the beginning of each movement and ends when a waypoint is reached).

The `ascanc_demo` macro illustrates the most basic features of a continuous scan::

```
class ascanc_demo(Macro):
    """
    This is a basic reimplementation of the ascanc` macro for demonstration
    purposes of the Generic Scan framework. The "real" implementation of
    :class:`sardana.macroserver.macros.ascanc` derives from
    :class:`sardana.macroserver.macros.aNscan` and provides some extra features.
    """

    hints = { 'scan' : 'ascanc_demo'} #this is used to indicate other codes that the␣
↪macro is a scan
    env = ('ActiveMntGrp',) #this hints that the macro requires the ActiveMntGrp␣
↪environment variable to be set

    param_def = [
       ['motor',      Type.Moveable, None, 'Motor to move'],
       ['start_pos',  Type.Float,    None, 'Scan start position'],
       ['final_pos',  Type.Float,    None, 'Scan final position'],
       ['integ_time', Type.Float,    None, 'Integration time']
    ]

    def prepare(self, motor, start_pos, final_pos, integ_time, **opts):
        self.name='ascanc_demo'
        #parse the user parameters
        self.start = numpy.array([start_pos], dtype='d')
        self.final = numpy.array([final_pos], dtype='d')
        self.integ_time = integ_time
        env = opts.get('env',{}) #the "env" dictionary may be passed as an option

        #create an instance of GScan (in this case, of its child, CScan
        self._gScan = CScan(self,
                            waypointGenerator=self._waypoint_generator,
                            periodGenerator=self._period_generator,
                            moveables=[motor],
                            env=env)

    def _waypoint_generator(self):
        #a very simple waypoint generator! only start and stop points!
        yield {"positions":self.start, "waypoint_id": 0}
        yield {"positions":self.final, "waypoint_id": 1}


    def _period_generator(self):
        step = {}
        step["integ_time"] =  self.integ_time
        point_no = 0
        while(True): #infinite generator. The acquisition loop is started/stopped at␣
↪begin and end of each waypoint
            point_no += 1
            step["point_id"] = point_no
            yield step

    def run(self,*args):
        for step in self._gScan.step_scan():
            yield step
```

See also:

for another example of a continuous scan implementation (with more elaborated waypoint generator), see the code of `meshc`

Hooks support in scans

In general, the Hooks API provided by the `Hookable` base class allows a macro to run other code (the hook callable) at certain points of its execution. The hooks use a "hints" mechanism to pass the receiving macro some extra information on how/when they should be executed. The hints are strings, and its content is not fixed by the API, being up to each macro to identify, use and/or ignore them.

You can find some examples of the use of hooks in the `hooks` module.

In the case of the scan macros, the hooks can be either registered directly via the Hooks API or passed as key:values of the "step" dictionary returned by the scan `generator()` (see `GScan` for more details).

The hints for a given hook are used by the scan framework to select the moment of the scan execution that the given hook is run. The following is a list of hint strings that scan macros support (other hints are ignored):

- 'pre-scan-hooks' : before starting the scan.
- 'pre-move-hooks' : for steps: before starting to move.
- 'post-move-hooks': for steps: after finishing the move.
- 'pre-acq-hooks' : for steps: before starting to acquire.
- 'post-acq-hooks' : for steps: after finishing acquisition but before recording the step.
- 'post-step-hooks' : for steps: after finishing recording the step.
- 'post-scan-hooks' : after finishing the scan

See the code of `hooked_scan` for a macro that demonstrates the use of the hook points of a scan.

Other examples of the `hooks` module can be illustrative.

Also, note that the Taurus MacroExecutor widget allows the user to dynamically add hooks to existing macros before execution.

More examples

Other macros in the `examples` module illustrate more features of the scan framework.

See also the code of the standard scan macros in the `scan` module.

Finally, the documentation and code of `GScan`, `SScan` and `CScan` may be helpful.

Writing controllers

This chapter provides the necessary information to write controllers in sardana.

An overview of the pool controller concept can be found *here*.

The complete controller *API* can be found *here*.

First, the common interface to all controller types is explained. After, a detailed chapter will focus on each specific controller type:

What is a controller

A controller in sardana is a piece of software capable of *translating* between the sardana *API* and a specific hardware *API*. Sardana expects a controller to obey a specific *API* in order to be able to properly configure and operate with it. The hardware *API* used by the controller could be anything, from a pure serial line to shared memory or a remote server written in Tango[183], Taco[184] or even EPICS[185].

Controllers can only be written in Python[186] (in future also C++ will be possible). A controller **must** be a *class* inheriting from one of the existing controller types:

- `MotorController`
- `CounterTimerController`
- `ZeroDController`
- `OneDController`
- `TwoDController`
- `IORegisterController`
- `TriggerGateController`
- `PseudoMotorController`
- `PseudoCounterController`

A controller is designed to incorporate a set of generic individual elements. Each element has a corresponding *axis*. For example, in a motor controller the elements will be motors, but in a counter/timer controller the elements will be experimental channels.

Some controller classes are designed to target a specific type of hardware. Other classes of controllers, the *pseudo* classes, are designed to provide a high level view over a set of underlying lower level controller elements.

We will focus first on writing low level hardware controllers since they share some of the *API* and after on the *pseudo* controllers.

Controller - The basics

The first thing to do is to import the necessary symbols from sardana library. As you will see, most symbols can be imported through the `sardana.pool.controller` module:

```python
import springfieldlib

from sardana.pool.controller import MotorController

class SpringfieldMotorController(MotorController):
    """A motor controller intended for demonstration purposes only"""
    pass
```

The common *API* to all low level controllers includes the set of methods to:

1. construct the controller
2. add/delete a controller element[202]
3. obtain the state of controller element(s)[203]

[183] http://www.tango-controls.org/
[184] http://www.esrf.eu/Infrastructure/Computing/TACO/
[185] http://www.aps.anl.gov/epics/
[186] http://www.python.org/
[202] Pseudo controllers don't need to manage their individual axis. Therefore, for pseudos you will not implement these methods
[203] For pseudo controllers, sardana will calculate the state of each pseudo axis based on the state of the elements that serve as input to the pseudo controller. Therefore, for pseudos you will not implement these methods

4. define, set and get extra axis attributes
5. define, set and get extra controller attributes
6. define, set and get extra controller properties

In the following chapters the examples will be based on a motor controller scenario.

The examples use a `springfieldlib` module which emulates a motor hardware access library.

The `springfieldlib` can be downloaded from `here`.

The Springfield motor controller can be downloaded from `here`.

Constructor

The constructor consists of the `__init__()` method. This method is called when you create a new controller of that type and every time the sardana server is started. It will also be called if the controller code has changed on the file and the new code is reloaded into sardana.

It is **NOT** mandatory to override the `__init__()` from *MotorController*. Do it only if you need to add some initialization code. If you do it, it is **very important** to follow the two rules:

1. use the method signature: `__init__(self, inst, props, *args, **kwargs)`
2. always call the super class constructor

The example shows how to implement a constructor for a motor controller:

```python
class SpringfieldMotorController(MotorController):

    def __init__(self, inst, props, *args, **kwargs):
        super(SpringfieldMotorController, self).__init__(inst, props, *args, **kwargs)

        # initialize hardware communication
        self.springfield = springfieldlib.SpringfieldMotorHW()

        # do some initialization
        self._motors = {}
```

Add/Delete axis

Each individual element in a controller is called *axis*. An axis is represented by a number. A controller can support one or more axes. Axis numbers don't need to be sequencial. For example, at one time you may have created for your motor controller instance only axis 2 and 5.

Two methods are called when creating or removing an element from a controller. These methods are *AddDevice()* and *DeleteDevice()*. The *AddDevice()* method is called when a new axis belonging to the controller is created in sardana. The *DeleteDevice()* method is called when an axis belonging to the controller is removed from sardana. These methods are also called when the sardana server is started and if the controller code has changed on the file and the new code is reloaded into sardana.

The example shows an example how to implement these methods on a motor controller:

```python
class SpringfieldMotorController(MotorController):

    def AddDevice(self, axis):
        self._motors[axis] = True
```

(continues on next page)

(continued from previous page)

```
    def DeleteDevice(self, axis):
        del self._motor[axis]
```

Get axis state

To get the state of an axis, sardana calls the *StateOne()* method. This method receives an axis as parameter and should return either:

- state (*State*) or
- **a sequence of two elements:**
  - state (*State*)
  - status (str[187])

(For motor controller see *get motor state*):

The state should be a member of *State* (For backward compatibility reasons, it is also supported to return one of PyTango.DevState). The status could be any string.

If you return a *State* object, sardana will compose a status string with:

> <axis name> is in <state name>

Here is an example of the possible implementation of *StateOne()* :

```
from sardana import State

class SpringfieldMotorController(MotorController):

    StateMap = {
        1 : State.On,
        2 : State.Moving,
        3 : State.Fault,
    }

    def StateOne(self, axis):
        springfield = self.springfield
        state = self.StateMap[ springfield.getState(axis) ]
        status = springfield.getStatus(axis)
        return state, status
```

Extra axis attributes

Each axis is associated a set of standard attributes. These attributes depend on the type of controller (example, a motor will have velocity, acceleration but a counter won't).

Additionally, you can specify an additional set of extra attributes on each axis.

Lets suppose that a Springfield motor controller can do close loop on hardware. We could define an extra motor attribute on each axis that (de)actives close loop on demand.

[187] https://docs.python.org/dev/library/stdtypes.html#str

The first thing to do is to specify which are the extra attributes. This is done through the `axis_attributes`. This is basically a dictionary where the keys are attribute names and the value is a dictionary describing the folowing properties for each attribute:

config. parameter	Mandatory	Key	Default value	Example
data type & format	Yes	`Type`	—	`int`[188]
data access	No	`Access`	`ReadWrite`	`ReadOnly`
description	No	`Description`	"" (empty string)	"the motor encoder source"
default value	No	`DefaultValue`	—	12345
getter method name	No	`FGet`	"get" + <name>	"getEncoderSource"
setter method name	No	`FSet`	"set" + <name>	"setEncoderSource"
memorize value	No	`Memorize`	`Memorized`	`NotMemorized`
max dimension size	No	`MaxDimSize`	Scalar: `()`; 1D: `(2048,)`; 2D: `(2048, 2048)`	`(2048,)`

Here is an example of how to specify the scalar, boolean, read-write *CloseLoop* extra attribute in a Springfield motor controller:

```
from sardana import DataAccess
from sardana.pool.controller import Type, Description, DefaultValue, Access, FGet,␣
↪FSet

class SpringfieldMotorController(MotorController):

    axis_attributes = {
        "CloseLoop" : {
                Type         : bool,
                Description  : "(de)activates the motor close loop algorithm",
                DefaultValue : False,
            },
    }

    def getCloseLoop(self, axis):
        return self.springfield.isCloseLoopActive(axis)

    def setCloseLoop(self, axis, value):
        self.springfield.setCloseLoop(axis, value)
```

When sardana needs to read the close loop value, it will first check if the controller has the method specified by the `FGet` keyword (we didn't specify it in `axis_attributes` so it defaults to *getCloseLoop*). It will then call this controller method which should return a value compatible with the attribute data type.

As an alternative, to avoid filling the controller code with pairs of get/set methods, you can choose not to write the getCloseLoop and setCloseLoop methods. This will trigger sardana to call the `GetAxisExtraPar()` /`SetAxisExtraPar()` pair of methods. The disadvantage is you will end up with a forest of `if`[189] … `elif`[190] … `else`[191] statements. Here is the alternative implementation:

[188] https://docs.python.org/dev/library/functions.html#int
[189] https://docs.python.org/dev/reference/compound_stmts.html#if
[190] https://docs.python.org/dev/reference/compound_stmts.html#elif
[191] https://docs.python.org/dev/reference/compound_stmts.html#else

```
from sardana import DataAccess
from sardana.pool.controller import Type, Description, DefaultValue, Access, FGet,␣
↪FSet

class SpringfieldMotorController(MotorController):

    axis_attributes = {
        "CloseLoop" : {
                Type         : bool,
                Description  : "(de)activates the motor close loop algorithm",
                DefaultValue : False,
            },
    }

    def GetAxisExtraPar(self, axis, parameter):
        if parameter == 'CloseLoop':
            return self.springfield.isCloseLoopActive(axis)

    def SetAxisExtraPar(self, axis, parameter, value):
        if parameter == 'CloseLoop':
             self.springfield.setCloseLoop(axis, value)
```

Sardana gives you the choice: we leave it up to you to decide which is the better option for your specific case.

Extra controller attributes

Besides extra attributes per axis, you can also define extra attributes at the controller level. In order to do that you have to specify the extra controller attribute(s) within the *ctrl_attributes* member. The syntax for this dictionary is the same as the one used for *axis_attributes*.

Here is an example on how to specify a read-only float matrix attribute called *ReflectionMatrix* at the controller level:

```
class SpringfieldMotorController(MotorController):

    ctrl_attributes = {
        "ReflectionMatrix" : {
                Type         : ( (float,), ),
                Description  : "The reflection matrix",
                Access : DataAccess.ReadOnly,
            },
    }

    def getReflectionMatrix(self):
        return ( (1.0, 0.0), (0.0, 1.0) )
```

Or, similar to what you can do with axis attributes:

```
class SpringfieldMotorController(MotorController):

    ctrl_attributes = \
    {
        "ReflectionMatrix" : {
                Type         : ( (float,), ),
                Description  : "The reflection matrix",
```

(continues on next page)

(continued from previous page)

```
                Access : DataAccess.ReadOnly,
            },
    }

    def GetCtrlPar(self, name):
        if name == "ReflectionMatrix":
            return ( (1.0, 0.0), (0.0, 1.0) )
```

Extra controller properties

A more static form of attributes can be defined at the controller level. These *properties* are loaded into the controller at the time of object construction. They are accessible to your controller at any time but it is not possible for a user from outside to modify them. The way to define `ctrl_properties` is very similar to the way you define extra axis attributes or extra controller attributes.

Here is an example on how to specify a host and port properties:

```
class SpringfieldMotorController(MotorController):

    ctrl_properties = \
    {
        "host" : {
                Type : str,
                Description : "host name"
            },
        "port" : {
                Type : int,
                Description : "port number",
                DefaultValue: springfieldlib.SpringfieldMotorHW.DefaultPort
           },
    }

    def __init__(self, inst, props, *args, **kwargs):
        super(SpringfieldMotorController, self).__init__(inst, props, *args, **kwargs)

        host = self.host
        port = self.port

        # initialize hardware communication
        self.springfield = springfieldlib.SpringfieldMotorHW(host=host, port=port)

        # do some initialization
        self._motors = {}
```

As you can see from lines 15 and 16, to access your controller properties simply use `self.<property name>`. Sardana assures that every property has a value. In our case, when a SpringfieldMotorController is created, if port property is not specified by the user (example: using the `defctrl` macro in spock), sardana assignes the default value `springfieldlib.SpringfieldMotorHW.DefaultPort`. On the other hand, since host has no default value, if it is not specified by the user, sardana will complain and fail to create and instance of SpringfieldMotorController.

Error handling

When you write a controller it is important to properly handle errors (example: motor power overload, hit a limit switch, lost of communication with the hardware).

These are the two basic sardana rules you should have in mind:

1. The exceptions which are not handled by the controller are handled by sardana, usually by re-raising the exception (when sardana runs as a Tango[192] DS a translation is done from the Python[193] exception to a Tango[194] exception). The *StateOne()* method is handled a little differently: the state is set to `Fault` and the status will contain the exception information.
2. When the methods which are supposed to return a value (like *GetAxisPar()*) don't return a value compatible with the expected data type (including `None`[195]) a `TypeError`[196] exception is thrown.

In every method you should carefully choose how to do handle the possible exceptions/errors.

Usually, catch and handle is the best technique since it is the code of your controller which knows exactly the workings of the hardware. You can discriminate errors and decide a proper handle for each. Essencially, this technique consists of:

1. catching the error (if an exception: with `try`[197] ... `except`[198] clause, if an expected return of a function: with a `if`[199] ... `elif`[200] ... `else`[201] statement, etc)
2. raise a proper exception (could be the same exception that has been catched) or, if in *StateOne()*, return the apropriate error state (`Fault`, `Alarm`) and a descriptive status.

Here is an example: if the documentation of the underlying library says that:

> *reading the motor closeloop raises CommunicationFailed if it is not possible to communicate with the Springfield hardware*
>
> *reading the motor state raises MotorPowerOverload if the motors has a power overload or a MotorTempTooHigh when the motor temperature is too high*

then you should handle the exception in the controller and return a proper state information:

```
def getCloseLoop(self, axis):
    # Here the "proper exception" to raise in case of error is actually the
    # one that is raised from the springfield library so handling the
    # exception is transparent. Nice!
    return self.springfield.isCloseLoopActive(axis)

def StateOne(self, axis):
    springfield = self.springfield

    try:
        state = self.StateMap[ springfield.getState(axis) ]
        status = springfield.getStatus(axis)
    except springfieldlib.MotorPowerOverload:
        state = State.Fault
```

(continues on next page)

[192] http://www.tango-controls.org/
[193] http://www.python.org/
[194] http://www.tango-controls.org/
[195] https://docs.python.org/dev/library/constants.html#None
[196] https://docs.python.org/dev/library/exceptions.html#TypeError
[197] https://docs.python.org/dev/reference/compound_stmts.html#try
[198] https://docs.python.org/dev/reference/compound_stmts.html#except
[199] https://docs.python.org/dev/reference/compound_stmts.html#if
[200] https://docs.python.org/dev/reference/compound_stmts.html#elif
[201] https://docs.python.org/dev/reference/compound_stmts.html#else

(continued from previous page)

```
        status = "Motor has a power overload"
    except springfieldlib.MotorTempTooHigh:
        temp = springfield.getTemperature(axis)
        state = State.Alarm
        status = "Motor temperature is too high (%f degrees)" % temp

    limit_switches = MotorController.NoLimitSwitch
    hw_limit_switches = springfield.getLimits(axis)
    if hw_limit_switches[0]:
        limit_switches |= MotorController.HomeLimitSwitch
    if hw_limit_switches[1]:
        limit_switches |= MotorController.UpperLimitSwitch
    if hw_limit_switches[2]:
        limit_switches |= MotorController.LowerLimitSwitch
    return state, status, limit_switches
```

Hiding the exception is usually a **BAD** technique since it prevents the user from finding what was the cause of the problem. You should only use it in extreme cases (example: if there is a bug in sardana which crashes the server if you try to properly raise an exception, then you can **temporarely** use this technique until the bug is solved).

Example:

```
def getCloseLoop(self, axis):
    # BAD error handling technique
    try:
        return self.springfield.isCloseLoopActive(axis)
    except:
        pass
```

How to write a motor controller

The basics

An example of a hypothetical *Springfield* motor controller will be build incrementally from scratch to aid in the explanation.

By now you should have read the general controller basics chapter. You should now have a MotorController with a proper constructor, add and delete axis methods:

```
import springfieldlib

from sardana.pool.controller import MotorController

class SpringfieldMotorController(MotorController):

    def __init__(self, inst, props, *args, **kwargs):
        super(SpringfieldMotorController, self).__init__(inst, props, *args, **kwargs)

        # initialize hardware communication
        self.springfield = springfieldlib.SpringfieldMotorHW()

        # do some initialization
        self._motors = {}
```

(continues on next page)

(continued from previous page)

```
    def AddDevice(self, axis):
        self._motors[axis] = True

    def DeleteDevice(self, axis):
        del self._motor[axis]
```

The *get axis state* method has some details that will be explained below.

The examples use a `springfieldlib` module which emulates a motor hardware access library.

The `springfieldlib` can be downloaded from `here`.

The Springfield motor controller can be downloaded from `here`.

The following code describes a minimal *Springfield* base motor controller which is able to return both the state and position of a motor as well as move a motor to the desired position:

```
class SpringfieldBaseMotorController(MotorController):
    """The most basic controller intended from demonstration purposes only.
    This is the absolute minimum you have to implement to set a proper motor
    controller able to get a motor position, get a motor state and move a
    motor.

    This example is so basic that it is not even directly described in the
    documentation"""

    MaxDevice = 128

    def __init__(self, inst, props, *args, **kwargs):
        """Constructor"""
        super(SpringfieldBaseMotorController, self).__init__(
            inst, props, *args, **kwargs)
        self.springfield = springfieldlib.SpringfieldMotorHW()

    def ReadOne(self, axis):
        """Get the specified motor position"""
        return self.springfield.getPosition(axis)

    def StateOne(self, axis):
        """Get the specified motor state"""
        springfield = self.springfield
        state = springfield.getState(axis)
        if state == 1:
            return State.On, "Motor is stopped"
        elif state == 2:
            return State.Moving, "Motor is moving"
        elif state == 3:
            return State.Fault, "Motor has an error"

    def StartOne(self, axis, position):
        """Move the specified motor to the specified position"""
        self.springfield.move(axis, position)

    def StopOne(self, axis):
        """Stop the specified motor"""
        self.springfield.stop(axis)
```

This code is shown only to demonstrate the minimal controller *API*. The advanced motor controller chapters describe how to account for more complex behaviour like reducing the number of hardware accesses or synchronize motion of multiple motors.

Get motor state

To get the state of a motor, sardana calls the `StateOne()` method. This method receives an axis as parameter and should return either:

- state (`State`) or
- **a sequence of two elements:**
  - state (`State`)
  - status (`str`[204]) *or* limit switches (`int`[205])
- **a sequence of three elements:**
  - state (`State`)
  - status (`str`[206])
  - limit switches (`int`[207])

The state should be a member of `State` (For backward compatibility reasons, it is also supported to return one of `PyTango.DevState`). The status could be any string. The limit switches is a integer with bits representing the three possible limits: home, upper and lower. Sardana provides three constants which can be *or*ed together to provide the desired limit switch:

- `NoLimitSwitch`
- `HomeLimitSwitch`
- `UpperLimitSwitch`
- `LowerLimitSwitch`

To say both home and lower limit switches are active (rare!) you can do:

```
limit_switches = MotorController.HomeLimitSwitch | MotorController.LowerLimitSwitch
```

If you don't return a status, sardana will compose a status string with:

> <axis name> is in <state name>

If you don't return limit switches, sardana will assume all limit switches are off.

Here is an example of the possible implementation of `StateOne()`:

```
from sardana import State

class SpringfieldMotorController(MotorController):

    StateMap = {
        1 : State.On,
        2 : State.Moving,
        3 : State.Fault,
    }

```

(continues on next page)

[204] https://docs.python.org/dev/library/stdtypes.html#str
[205] https://docs.python.org/dev/library/functions.html#int
[206] https://docs.python.org/dev/library/stdtypes.html#str
[207] https://docs.python.org/dev/library/functions.html#int

(continued from previous page)

```
    def StateOne(self, axis):
        springfield = self.springfield
        state = self.StateMap[ springfield.getState(axis) ]
        status = springfield.getStatus(axis)

        limit_switches = MotorController.NoLimitSwitch
        hw_limit_switches = springfield.getLimits(axis)
        if hw_limit_switches[0]:
            limit_switches |= MotorController.HomeLimitSwitch
        if hw_limit_switches[1]:
            limit_switches |= MotorController.UpperLimitSwitch
        if hw_limit_switches[2]:
            limit_switches |= MotorController.LowerLimitSwitch
        return state, status, limit_switches
```

Get motor position

To get the motor position, sardana calls the `ReadOne()` method. This method receives an axis as parameter and should return a valid position. Sardana interprets the returned position as a *dial position*.

Here is an example of the possible implementation of `ReadOne()`:

```
class SpringfieldMotorController(MotorController):

    def ReadOne(self, axis):
        position = self.springfield.getPosition(axis)
        return position
```

Move a motor

When an order comes for sardana to move a motor, sardana will call the `StartOne()` method. This method receives an axis and a position. The controller code should trigger the hardware motion. The given position is always the *dial position*.

Here is an example of the possible implementation of `StartOne()`:

```
class SpringfieldMotorController(MotorController):

    def StartOne(self, axis, position):
        self.springfield.move(axis, position)
```

As soon as `StartOne()` is invoked, sardana expects the motor to be moving. It enters a high frequency motion loop which asks for the motor state through calls to `StateOne()`. It will keep the loop running as long as the controller responds with `State.Moving`. If `StateOne()` raises an exception or returns something other than `State.Moving`, sardana will assume the motor is stopped and exit the motion loop.

For a motion to work properly, it is therefore, **very important** that `StateOne()` responds correctly.

Stop a motor

It is possible to stop a motor when it is moving. When sardana is ordered to stop a motor motion, it invokes the `StopOne()` method. This method receives an axis parameter. The controller should make sure the desired motor is *gracefully* stopped, if possible, respecting the configured motion parameters (like deceleration and base_rate).

Here is an example of the possible implementation of `StopOne()`:

```
class SpringfieldMotorController(MotorController):

    def StopOne(self, axis):
        self.springfield.stop(axis)
```

Abort a motor

In a danger situation (motor moving a table about to hit a wall), it is desirable to abort a motion *as fast as possible*. When sardana is ordered to abort a motor motion, it invokes the `AbortOne()` method. This method receives an axis parameter. The controller should make sure the desired motor is stopped as fast as it can be done, possibly losing track of position.

Here is an example of the possible implementation of `AbortOne()`:

```
class SpringfieldMotorController(MotorController):

    def AbortOne(self, axis):
        self.springfield.abort(axis)
```

Note: The default implementation of `StopOne()` calls `AbortOne()` so, if your controller cannot distinguish stopping from aborting, it is sufficient to implement `AbortOne()`.

Standard axis attributes

By default, sardana expects every axis to have a set of attributes:

- acceleration
- deceleration
- velocity
- base rate
- steps per unit

To set and retrieve the value of these attributes, sardana invokes pair of methods: `GetAxisPar()` /`SetAxisPar()`

Here is an example of the possible implementation:

```
class SpringfieldMotorController(MotorController):

    def GetAxisPar(self, axis, name):
        springfield = self.springfield
        name = name.lower()
        if name == "acceleration":
            v = springfield.getAccelerationTime(axis)
        elif name == "deceleration":
            v = springfield.getDecelerationTime(axis)
```

(continues on next page)

(continued from previous page)

```
        elif name == "base_rate":
            v = springfield.getMinVelocity(axis)
        elif name == "velocity":
            v = springfield.getMaxVelocity(axis)
        elif name == "step_per_unit":
            v = springfield.getStepPerUnit(axis)
        return v

    def SetAxisPar(self, axis, name, value):
        springfield = self.springfield
        name = name.lower()
        if name == "acceleration":
            springfield.setAccelerationTime(axis, value)
        elif name == "deceleration":
            springfield.setDecelerationTime(axis, value)
        elif name == "base_rate":
            springfield.setMinVelocity(axis, value)
        elif name == "velocity":
            springfield.setMaxVelocity(axis, value)
        elif name == "step_per_unit":
            springfield.setStepPerUnit(axis, value)
```

See also:

What to do when… What to do when your hardware motor controller doesn't support steps per unit

Define a position

Sometimes it is useful to reset the current position to a certain value. Imagine you are writing a controller for a hardware controller which handles stepper motors. When the hardware is asked for a motor position it will probably answer some value from an internal register which is incremented/decremented each time the motor goes up/down a step. Probably this value as physical meaning so the usual procedure is to move the motor to a known position (home switch, for example) and once there, set a meaningful position to the current position. Some motor controllers support reseting the internal register to the desired value. If your motor controller can do this the implementation is as easy as writing the `DefinePosition()` and call the proper code of your hardware library to do it:

```
class SpringfieldMotorController(MotorController):

    def DefinePosition(self, axis, position):
        self.springfield.setCurrentPosition(axis, position)
```

See also:

What to do when…

What to do when your hardware motor controller doesn't support defining the position

What to do when…

This chapter describes common difficult situations you may face when writing a motor controller in sardana, and possible solutions to solve them.

my controller doesn't support steps per unit Many (probably, most) hardware motor controllers don't support steps per unit at the hardware level. This means that your sardana controller should be able to emulate steps per unit at the software level. This can be easily done, but it requires you to make some changes in your code.

We will assume now that the Springfield motor controller doesn't support steps per unit feature. The first that needs to be done is to modify the `AddDevice()` method so it is able to to store the resulting conversion factor between the hardware read position and the position the should be returned (the *step_per_unit*). The `ReadOne()` also needs to be rewritten to make the proper calculation. Finally `GetAxisPar()` / `SetAxisPar()` methods need to be rewritten to properly get/set the step per unit value:

```python
class SpringfieldMotorController(MotorController):

    def AddDevice(self, axis):
        self._motor[axis] = dict(step_per_unit=1.0)

    def ReadOne(self, axis):
        step_per_unit = self._motor[axis]["step_per_unit"]
        position = self.springfield.getPosition(axis)
        return position / step_per_unit

    def GetAxisPar(self, axis, name):
        springfield = self.springfield
        name = name.lower()
        if name == "acceleration":
            v = springfield.getAccelerationTime(axis)
        elif name == "deceleration":
            v = springfield.getDecelerationTime(axis)
        elif name == "base_rate":
            v = springfield.getMinVelocity(axis)
        elif name == "velocity":
            v = springfield.getMaxVelocity(axis)
        elif name == "step_per_unit":
            v = self._motor[axis]["step_per_unit"]
        return v

    def SetAxisPar(self, axis, name, value):
        springfield = self.springfield
        name = name.lower()
        if name == "acceleration":
            springfield.setAccelerationTime(axis, value)
        elif name == "deceleration":
            springfield.setDecelerationTime(axis, value)
        elif name == "base_rate":
            springfield.setMinVelocity(axis, value)
        elif name == "velocity":
            springfield.setMaxVelocity(axis, value)
        elif name == "step_per_unit":
            self._motor[axis]["step_per_unit"] = value
```

my controller doesn't support defining the position Some controllers may not be able to reset the position to a different value. In these cases, your controller code should be able to emulate such a feature. This can be easily done, but it requires you to make some changes in your code.

We will now assume that the Springfield motor controller doesn't support steps per unit feature. The first thing that needs to be done is to modify the `AddDevice()` method so it is able to store the resulting offset between the hardware read position and the position the should be returned (the *define_position_offset*). The `ReadOne()` also needs to be rewritten to take the *define_position_offset* into account. Finally `DefinePosition()` needs to be written to update the *define_position_offset* to the desired value:

```
class SpringfieldMotorController(MotorController):

    def AddDevice(self, axis):
        self._motor[axis] = dict(define_position_offset=0.0)

    def ReadOne(self, axis):
        dp_offset = self._motor[axis]["define_position_offset"]
        position = self.springfield.getPosition(axis)
        return position + dp_offset

    def DefinePosition(self, axis, position):
        current_position = self.springfield.getPosition(axis)
        self._motor[axis]["define_position_offset"] = position - current_position
```

Advanced topics

Timestamp a motor position

When you read the position of a motor from the hardware sometimes it is necessary to associate a timestamp with that position so you can track the position of a motor in time.

If sardana is executed as a Tango device server, reading the position attribute from the motor device triggers the execution of your controller's *ReadOne()* method. Tango responds with the value your controller returns from the call to *ReadOne()* and automatically assigns a timestamp. However this timestamp has a certain delay since the time the value was actually read from hardware and the time Tango generates the timestamp.

To avoid this, sardana supports returning in *ReadOne()* an object that contains both the value and the timestamp instead of the usual `numbers.Number`[208]. The object must be an instance of *SardanaValue*.

Here is an example of associating a timestamp in *ReadOne()*:

```
import time
from sardana.pool.controller import SardanaValue

class SpringfieldMotorController(MotorController):

   def ReadOne(self, axis):
       return SardanaValue(value=self.springfield.getPosition(axis),
                           timestamp=time.time())
```

If your controller communicates with a Tango device, Sardana also supports returning a `DeviceAttribute` object. Sardana will use this object's value and timestamp. Example:

```
class TangoMotorController(MotorController):

   def ReadOne(self, axis):
       return self.device.read_attribute("position")
```

[208] https://docs.python.org/dev/library/numbers.html#numbers.Number

Multiple motion synchronization

This chapter describes an extended *API* that allows you to better synchronize motions involing more than one motor, as well as optimize hardware communication (in case the hardware interface also supports this).

Often it is the case that the experiment/procedure the user runs requires to move more than one motor at the same time. Imagine that the user requires motor at axis 1 to be moved to 100mm and motor axis 2 to be moved to -20mm. Your controller will receive two consecutive calls to `StartOne()`:

```
StartOne(1, 100)
StartOne(2, -20)
```

and each StartOne will probably connect to the hardware (through serial line, socket, Tango[209] or EPICS[210]) and ask the motor to be moved. This will do the job but, there will be a slight desynchronization between the two motors because hardware call of motor 1 will be done before hardware call to motor 2.

Sardana provides an extended *start motion* which gives you the possibility to improve the syncronization (and probably reduce communications) but your hardware controller must somehow support this feature as well.

The complete start motion *API* consists of four methods:

- `PreStartAll()`
- `PreStartOne()`
- `StartOne()`
- `StartAll()`

Except for `StartOne()`, the implemenation of all other start methods is optional and their default implementation does nothing (`PreStartOne()` actually returns `True`).

So, actually, the complete algorithm for motor motion in sardana is:

```
/FOR/ Each controller(s) implied in the motion
     - Call PreStartAll()
/END FOR/

/FOR/ Each motor(s) implied in the motion
     - ret = PreStartOne(motor to move, new position)
     - /IF/ ret is not true
        /RAISE/ Cannot start. Motor PreStartOne returns False
     - /END IF/
     - Call StartOne(motor to move, new position)
/END FOR/

/FOR/ Each controller(s) implied in the motion
     - Call StartAll()
/END FOR/
```

So, for the example above where we move two motors, the complete sequence of calls to the controller is:

```
PreStartAll()

if not PreStartOne(1, 100):
    raise Exception("Cannot start. Motor(1) PreStartOne returns False")
```

(continues on next page)

[209] http://www.tango-controls.org/
[210] http://www.aps.anl.gov/epics/

(continued from previous page)

```
if not PreStartOne(2, -20):
    raise Exception("Cannot start. Motor(2) PreStartOne returns False")

StartOne(1, 100)
StartOne(2, -20)

StartAll()
```

Sardana assures that the above sequence is never interrupted by other calls, like a call from a different user to get motor state.

Suppose the springfield library tells us in the documentation that:

> … to move multiple motors at the same time use:
>
> ```
> moveMultiple(seq<pair<axis, position>>)
> ```
>
> Example:
>
> ```
> moveMultiple([[1, 100], [2, -20]])
> ```

We can modify our motor controller to take profit of this hardware feature:

```
class SpringfieldMotorController(MotorController):

    def PreStartAll(self):
        # clear the local motion information dictionary
        self._moveable_info = []

    def StartOne(self, axis, position):
        # store information about this axis motion
        motion_info = axis, position
        self._moveable_info.append(motion_info)

    def StartAll(self):
        self.springfield.moveMultiple(self._moveable_info)
```

In case of stopping/aborting of the motors (or any other stoppable/abortable elements) the synchronization may be as important as in case of starting them. Let's take an example of a motorized two-legged table and its translational movement. A desynchronized stop/abort of the motors may introduce an extra angle of the table that in very specific cases may be not desired e.g. activation of the safety limits, closed loop errors, etc.

In this case the complete algorithm for stopping/aborting the motor motion in sardana is:

```
/FOR/ Each controller(s) implied in the motion

    - Call PreStopAll()

    /FOR/ Each motor of the given controller implied in the motion
        - ret = PreStopOne(motor to stop)
        - /IF/ ret is not true
            /RAISE/ Cannot stop. Motor PreStopOne returns False
        - /END IF/
        - Call StopOne(motor to stop)
    /END FOR/
```

(continues on next page)

(continued from previous page)

```
    - Call StopAll()

/END FOR/
```

Each of the hardware controller method calls is protected in case of errors so the stopping/aborting algorithm tries to stop/abort as many axes/controllers.

A similar principle applies when sardana asks for the state and position of multiple axis. The two sets of methods are, in these cases:

- `PreStateAll()`
- `PreStateOne()`
- `StateAll()`
- `StateOne()`
- `PreReadAll()`
- `PreReadOne()`
- `ReadAll()`
- `ReadOne()`

The main differences between these sets of methods and the ones from start motion is that `StateOne()` / `ReadOne()` methods are called **AFTER** the corresponding `StateAll()` / `ReadAll()` counterparts and they are expeced to return the state/position of the requested axis.

The internal sardana algorithm to read position is:

```
/FOR/ Each controller(s) implied in the reading (executed concurrently)

     - Call PreReadAll()

    /FOR/ Each motor(s) of the given controller implied in the reading
         - PreReadOne(motor to read)
    /END FOR/

    - Call ReadAll()

    /FOR/ Each motor(s) of the given controller implied in the reading
         - ReadOne(motor to read)
    /END FOR/

/END FOR/
```

Here is an example assuming the springfield library tells us in the documentation that:

> . . . to read the position of multiple motors at the same time use:
>
> ```
> getMultiplePosition(seq<axis>) -> dict<axis, position>
> ```
>
> Example:
>
> ```
> positions = getMultiplePosition([1, 2])
> ```

The new improved code could look like this:

```
class SpringfieldMotorController(MotorController):

    def PreRealAll(self):
        # clear the local position information dictionary
```

(continues on next page)

(continued from previous page)

```
        self._position_info = []

    def PreReadOne(self, axis):
        self._position_info.append(axis)

    def ReadAll(self):
        self._positions = self.springfield.getMultiplePosition(self._position_info)

    def ReadOne(self, axis):
        return self._positions[axis]
```

How to write a counter/timer controller

The basics

An example of a hypothetical *Springfield* counter/timer controller will be build incrementally from scratch to aid in the explanation.

By now you should have read the general controller basics chapter. You should be able to create a CounterTimerController with:

- a proper constructor,
- add and delete axis methods
- get axis state

```
import springfieldlib

from sardana.pool.controller import CounterTimerController

from sardana import State

class SpringfieldCounterTimerController(CounterTimerController):

    def __init__(self, inst, props, *args, **kwargs):
        super(SpringfieldCounterTimerController, self).__init__(inst, props, *args,␣
↪**kwargs)

        # initialize hardware communication
        self.springfield = springfieldlib.SpringfieldCounterHW()

        # do some initialization
        self._counters = {}

    def AddDevice(self, axis):
        self._counters[axis] = True

    def DeleteDevice(self, axis):
        del self._counters[axis]

    StateMap = {
        1 : State.On,
        2 : State.Moving,
        3 : State.Fault,
```

(continues on next page)

(continued from previous page)

```
    }

    def StateOne(self, axis):
        springfield = self.springfield
        state = self.StateMap[ springfield.getState(axis) ]
        status = springfield.getStatus(axis)
        return state, status
```

The examples use a `springfieldlib` module which emulates a counter/timer hardware access library.

The `springfieldlib` can be downloaded from `here`.

The Springfield counter/timer controller can be downloaded from `here`.

The following code describes a minimal *Springfield* base counter/timer controller which is able to return both the state and value of an individual counter as well as to start an acquisition:

```
class SpringfieldBaseCounterTimerController(CounterTimerController):
    """The most basic controller intended from demonstration purposes only.
    This is the absolute minimum you have to implement to set a proper counter
    controller able to get a counter value, get a counter state and do an
    acquisition.

    This example is so basic that it is not even directly described in the
    documentation"""

    def __init__(self, inst, props, *args, **kwargs):
        """Constructor"""
        super(SpringfieldBaseCounterTimerController,
              self).__init__(inst, props, *args, **kwargs)
        self.springfield = springfieldlib.SpringfieldCounterHW()

    def ReadOne(self, axis):
        """Get the specified counter value"""
        return self.springfield.getValue(axis)

    def StateOne(self, axis):
        """Get the specified counter state"""
        springfield = self.springfield
        state = springfield.getState(axis)
        if state == 1:
            return State.On, "Counter is stopped"
        elif state == 2:
            return State.Moving, "Counter is acquiring"
        elif state == 3:
            return State.Fault, "Counter has an error"

    def StartOne(self, axis, value=None):
        """acquire the specified counter"""
        self.springfield.StartChannel(axis)

    def LoadOne(self, axis, value, repetitions):
        self.springfield.LoadChannel(axis, value)

    def StopOne(self, axis):
        """Stop the specified counter"""
        self.springfield.stop(axis)
```

Get counter state

To get the state of a counter, sardana calls the `StateOne()` method. This method receives an axis as parameter and should return either:

- state (`State`) or
- **a sequence of two elements:**
  - state (`State`)
  - status (`str`[211])

The state should be a member of `State` (For backward compatibility reasons, it is also supported to return one of `PyTango.DevState`). The status could be any string.

Load a counter

To load a counter with either the integration time or the monitor counts, sardana calls the `LoadOne()` method. This method receives axis, value and repetitions parameters. For the moment let's focus on the first two of them.

Here is an example of the possible implementation of `LoadOne()`:

```
class SpringfieldCounterTimerController(CounterTimerController):

    def LoadOne(self, axis, value, repetitions):
        self.springfield.LoadChannel(axis, value)
```

Get counter value

To get the counter value, sardana calls the `ReadOne()` method. This method receives an axis as parameter and should return a valid counter value. Sardana notifies the pseudo counters about the new counter value so they can be updated (see *Pseudo counter overview* for more details).

Here is an example of the possible implementation of `ReadOne()`:

```
class SpringfieldCounterTimerController(CounterTimerController):

    def ReadOne(self, axis):
        value = self.springfield.getValue(axis)
        return value
```

Start a counter

When an order comes for sardana to start a counter, sardana will call the `StartOne()` method. This method receives an axis as parameter. The controller code should trigger the hardware acquisition.

Here is an example of the possible implementation of `StartOne()`:

```
class SpringfieldCounterTimerController(CounterTimerController):

    def StartOne(self, axis, value):
        self.springfield.StartChannel(axis)
```

[211] https://docs.python.org/dev/library/stdtypes.html#str

As soon as `StartOne()` is invoked, sardana expects the counter to be acquiring. It enters a high frequency acquisition loop which asks for the counter state through calls to `StateOne()`. It will keep the loop running as long as the controller responds with `State.Moving`. If `StateOne()` raises an exception or returns something other than `State.Moving`, sardana will assume the counter is stopped and exit the acquisition loop.

For an acquisition to work properly, it is therefore, **very important** that `StateOne()` responds correctly.

Stop a counter

It is possible to stop a counter when it is acquiring. When sardana is ordered to stop a counter acquisition, it invokes the `StopOne()` method. This method receives an axis parameter. The controller should make sure the desired counter is *gracefully* stopped.

Here is an example of the possible implementation of `StopOne()`:

```
class SpringfieldCounterTImerController(CounterTimerController):

    def StopOne(self, axis):
        self.springfield.StopChannel(axis)
```

Abort a counter

In an emergency situation, it is desirable to abort an acquisition *as fast as possible*. When sardana is ordered to abort a counter acquisition, it invokes the `AbortOne()` method. This method receives an axis parameter. The controller should make sure the desired counter is stopped as fast as it can be done.

Here is an example of the possible implementation of `AbortOne()`:

```
class SpringfieldCounterTimerController(CounterTimerController):

    def AbortOne(self, axis):
        self.springfield.AbortChannel(axis)
```

Timer and monitor roles

Usually counters can work in either of two modes: timer or monitor. In both of them, one counter in a group is assigned a special role to control when the rest of them should stop counting. The stopping condition is based on the integration time in case of the timer or on the monitor counts in case of the monitor. The assignment of this special role is based on the measurement group *Configuration*. The controller receives this configuration (axis number) via the controller parameter `timer` and `monitor`. The currently used acquisition mode is set via the controller parameter `acquisition_mode`.

Advanced topics

Timestamp a counter value

When you read the value of a counter from the hardware sometimes it is necessary to associate a timestamp with that value so you can track the value of a counter in time.

If sardana is executed as a Tango device server, reading the value attribute from the counter device triggers the execution of your controller's `ReadOne()` method. Tango responds with the value your controller returns from the call to `ReadOne()` and automatically assigns a timestamp. However this timestamp has a certain delay since the time the value was actually read from hardware and the time Tango generates the timestamp.

To avoid this, sardana supports returning in `ReadOne()` an object that contains both the value and the timestamp instead of the usual `numbers.Number`[212]. The object must be an instance of `SardanaValue`.

Here is an example of associating a timestamp in `ReadOne()`:

```
import time
from sardana.pool.controller import SardanaValue

class SpringfieldCounterTimerController(CounterTimerController):

   def ReadOne(self, axis):
       return SardanaValue(value=self.springfield.getValue(axis),
                           timestamp=time.time())
```

If your controller communicates with a Tango device, Sardana also supports returning a `DeviceAttribute` object. Sardana will use this object's value and timestamp. Example:

```
class TangoCounterTimerController(CounterTimerController):

   def ReadOne(self, axis):
       return self.device.read_attribute("value")
```

Multiple acquisition synchronization

This chapter describes an extended *API* that allows you to better synchronize acquisitions involving more than one counter, as well as optimize hardware communication (in case the hardware interface also supports this).

Often it is the case that the experiment/procedure the user runs requires to acquire more than one counter at the same time (see *Measurement group overview*). Imagine that the user requires counter at axis 1 and counter at axis 2 to be acquired. Your controller will receive two consecutive calls to `StartOne()`:

```
StartOne(1)
StartOne(2)
```

and each StartOne will probably connect to the hardware (through serial line, socket, Tango[213] or EPICS[214]) and ask the counter to be started. This will do the job but, there will be a slight desynchronization between the two counters because hardware call of counter 1 will be done before hardware call to counter 2.

Sardana provides an extended *start acquisition* which gives you the possibility to improve the synchronization (and probably reduce communications) but your hardware controller must somehow support this feature as well.

The complete start acquisition *API* consists of four methods:

- `PreStartAll()`
- `PreStartOne()`
- `StartOne()`
- `StartAll()`

[212] https://docs.python.org/dev/library/numbers.html#numbers.Number
[213] http://www.tango-controls.org/
[214] http://www.aps.anl.gov/epics/

Except for `StartOne()`, the implementation of all other start methods is optional and their default implementation does nothing (`PreStartOne()` actually returns `True`).

So, actually, a simplified algorithm for counter acquisition start in sardana is:

```
/FOR/ Each controller(s) implied in the acquisition
     - Call PreStartAll()
/END FOR/

/FOR/ Each counter(s) implied in the acquisition
     - ret = PreStartOne(counter to acquire, new position)
     - /IF/ ret is not true
        /RAISE/ Cannot start. Counter PreStartOne returns False
     - /END IF/
     - Call StartOne(counter to acquire, new position)
/END FOR/

/FOR/ Each controller(s) implied in the acquisition
     - Call StartAll()
/END FOR/
```

So, for the example above where we acquire two counters, the complete sequence of calls to the controller is:

```
PreStartAll()

if not PreStartOne(1):
    raise Exception("Cannot start. Counter(1) PreStartOne returns False")
if not PreStartOne(2):
    raise Exception("Cannot start. Counter(2) PreStartOne returns False")

StartOne(1)
StartOne(2)

StartAll()
```

Sardana assures that the above sequence is never interrupted by other calls, like a call from a different user to get counter state.

Suppose the springfield library tells us in the documentation that:

> ... to acquire multiple counters at the same time use:
>
> ```
> startCounters(seq<axis>)
> ```
>
> Example:
>
> ```
> startCounters([1, 2])
> ```

We can modify our counter controller to take profit of this hardware feature:

```
class SpringfieldCounterTimerController(MotorController):

    def PreStartAll(self):
        # clear the local acquisition information dictionary
        self._counters_info = []

    def StartOne(self, axis):
        # store information about this axis motion
```

(continues on next page)

(continued from previous page)

```
        self._counters_info.append(axis)

    def StartAll(self):
        self.springfield.startCounters(self._counters_info)
```

Hardware synchronization

The synchronization achieved in *Multiple acquisition synchronization* may not be enough when it comes to acquiring with multiple controllers at the same time or to executing multiple acquisitions in a row. Some of the controllers can be synchronized on an external hardware event and in this case several important aspects needs to be taken into account.

Synchronization type

First of all the controller needs to know which type of synchronization will be used. This is assigned on the measurement group *Configuration* level. The controller receives one of the `AcqSynch` values via the controller parameter `synchronization`.

The selected mode will change the behavior of the counter after the `StartOne()` is invoked. In case one of the software modes was selected, the counter will immediately start acquiring. In case one of the hardware modes was selected, the counter will immediately get armed for the hardware events, and will wait with the acquisition until they occur.

Here is an example of the possible implementation of `SetCtrlPar()`:

```
from sardana.pool import AcqSynch

class SpringfieldCounterTimerController(CounterTimerController):

    SynchMap = {
        AcqSynch.SoftwareTrigger : 1,
        AcqSynch.SoftwareGate : 2,
        AcqSynch.HardwareTrigger: 3,
        AcqSynch.HardwareGate: 4
    }

    def SetCtrlPar(self, name, value):
        super(SpringfieldMotorController, self).SetCtrlPar(name, value)
        synchronization = SynchMap[value]
        if name == "synchronization":
            self.springfield.SetSynchronization(synchronization)
```

Multiple acquisitions

It is a very common scenario to execute multiple hardware synchronized acquisitions in a row. One example of this type of measurements are the *Continuous scans*. The controller receives the number of acquisitions via the third argument of the `LoadOne()` method.

Here is an example of the possible implementation of `LoadOne()`:

```
class SpringfieldCounterTimerController(CounterTimerController):

    def LoadOne(self, axis, value, repetitions):
        self.springfield.LoadChannel(axis, value)
        self.springfield.SetRepetitions(repetitions)
        return value
```

Get counter values

During the hardware synchronized acquisitions the counter values are usually stored in the hardware buffers. Sardana enters a high frequency acquisition loop after the `StartOne()` is invoked which, apart of asking for the counter state through calls to the `StateOne()` method, will try to retrieve the counter values using the `ReadOne()` method. It will keep the loop running as long as the controller responds with `State.Moving`. Sardana executes one extra readout after the state has changed in order to retrieve the final counter values.

The `ReadOne()` method is used indifferently of the selected synchronization but its return values should depend on it and can be:

- a single counter value: either `float`[215] or `SardanaValue` in case of the `SoftwareTrigger` or `SoftwareGate` synchronization
- a sequence of counter values: either `float`[216] or `SardanaValue` in case of the `HardwareTrigger` or `HardwareGate` synchronization

Sardana assumes that the counter values are returned in the order of acquisition and that there are no gaps in between them.

Todo: document how to skip the readouts while acquiring

How to write a 0D controller

Todo: complete 0D controller howto

Get 0D state

To get the state of a 0D, sardana calls the `StateOne()` method. During the acquisition loop this method is called only once when it is about to exit. This method receives an axis as parameter and should return either:

- state (`State`) or
- **a sequence of two elements:**
    - state (`State`)
    - status (`str`[217])

[215] https://docs.python.org/dev/library/functions.html#float
[216] https://docs.python.org/dev/library/functions.html#float
[217] https://docs.python.org/dev/library/stdtypes.html#str

The state should be a member of *State* (For backward compatibility reasons, it is also supported to return one of `PyTango.DevState`). The status could be any string.

If you don't return a status, sardana will compose a status string with:

> <axis name> is in <state name>

The controller could return on of the four states **On**, **Alarm**, **Fault** or **Unknown**. Apart of that sardana could set **Moving** or **Fault** state to the 0D. The Moving state is set during the acquisition loop to indicate that it is acquiring data. The Fault state is set when the controller software is not available (impossible to load it). The controller should return Fault if a fault is reported from the hardware controller or if the controller software returns an unforeseen state. The controller should return Unknown state if an exception occurs during the communication between the pool and the hardware controller.

How to write a 1D controller

The basics

Todo: document 1D controller howto

How to write a 2D controller

The basics

Todo: document 2D controller howto

How to write a trigger/gate controller

The basics

An example of a hypothetical *Springfield* trigger/gate controller will be build incrementally from scratch to aid in the explanation.

By now you should have read the general controller basics chapter. You should be able to create a TriggerGateController with:

- a proper constructor
- add and delete axis methods
- get axis state

```
import springfieldlib

from sardana.pool.controller import TriggerGateController

class SpringfieldTriggerGateController(TriggerGateController):

    def __init__(self, inst, props, *args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        super(SpringfieldTriggerGateController, self).__init__(inst, props, *args,␣
↪**kwargs)

        # initialize hardware communication
        self.springfield = springfieldlib.SpringfieldTriggerHW()

        # do some initialization
        self._triggers = {}

    def AddDevice(self, axis):
        self._triggers[axis] = True

    def DeleteDevice(self, axis):
        del self._triggers[axis]

    StateMap = {
        1 : State.On,
        2 : State.Moving,
        3 : State.Fault,
    }

    def StateOne(self, axis):
        springfield = self.springfield
        state = self.StateMap[ springfield.getState(axis) ]
        status = springfield.getStatus(axis)
        return state, status
```

The examples use a `springfieldlib` module which emulates a trigger/gate hardware access library.

The `springfieldlib` can be downloaded from `here`.

The Springfield trigger/gate controller can be downloaded from `here`.

The following code describes a minimal *Springfield* base trigger/gate controller which is able to return the state of an individual trigger as well as to start a synchronization:

```
class SpringfieldBaseTriggerGateController(TriggerGateController):
    """The most basic controller intended from demonstration purposes only.
    This is the absolute minimum you have to implement to set a proper trigger
    controller able to get a trigger value, get a trigger state and do an
    acquisition.

    This example is so basic that it is not even directly described in the
    documentation"""

    def __init__(self, inst, props, *args, **kwargs):
        """Constructor"""
        super(SpringfieldBaseTriggerGateController, self).__init__(
            inst, props, *args, **kwargs)
        self.springfield = springfieldlib.SpringfieldTriggerHW()

    def StateOne(self, axis):
        """Get the specified trigger state"""
        springfield = self.springfield
        state = springfield.getState(axis)
        if state == 1:
            return State.On, "Trigger is stopped"
```

(continues on next page)

(continued from previous page)

```
        elif state == 2:
            return State.Moving, "Trigger is running"
        elif state == 3:
            return State.Fault, "Trigger has an error"

    def StartOne(self, axis, value=None):
        """acquire the specified trigger"""
        self.springfield.StartChannel(axis)

    def SynchOne(self, axis, synchronization):
        self.springfield.SynchChannel(axis, synchronization)

    def StopOne(self, axis):
        """Stop the specified trigger"""
        self.springfield.stop(axis)
```

Get trigger state

To get the state of a trigger, sardana calls the `StateOne()` method. This method receives an axis as parameter and should return either:

- state (`State`) or
- **a sequence of two elements:**
  - state (`State`)
  - status (`str`[218])

The state should be a member of `State` (For backward compatibility reasons, it is also supported to return one of `PyTango.DevState`). The status could be any string.

Load synchronization description

To load a trigger with the synchronization description sardana calls the `SynchOne()` method. This method receives axis and synchronization parameters.

Here is an example of the possible implementation of `SynchOne()`:

```
class SpringfieldTriggerGateController(TriggerGateController):

    def SynchOne(self, axis, synchronization):
        self.springfield.SynchChannel(axis, synchronization)
```

Synchronization description

Synchronization is a data structure following a special convention. It is composed from the groups of equidistant intervals described by: the initial point and delay, total and active intervals and the number of repetitions. These information can be expressed in different synchronization domains if necessary: time and/or position.

[218] https://docs.python.org/dev/library/stdtypes.html#str

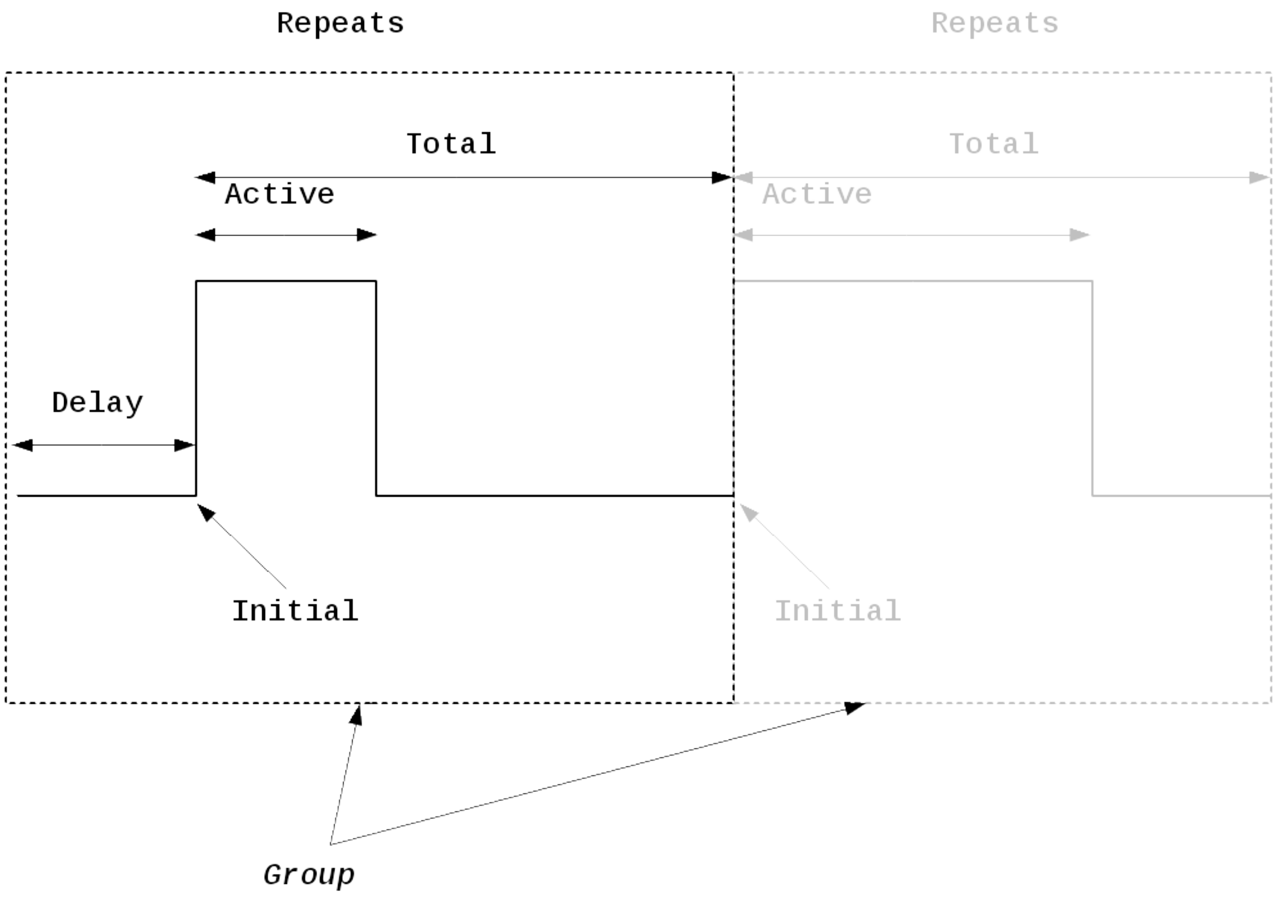


Fig. 30: This sketch depicts parameters describing a group.

Sardana defines two enumeration classes to help in manipulations of the synchronization description. The `SynchParam` defines the parameters used to describe a group. The `SynchDomain` defines the possible domains in which a parameter may be expressed.

The following code demonstrates creation of a synchronization description expressed in time and position domains (moveable's velocity = 10 units/second and acceleration time = 0.1 second). It will generate 10 synchronization pulses of length 0.1 second equally spaced on a distance of 100 units.

```
from sardana.pool import SynchParam, SynchDomain

synchronization = [
    {
        SynchParam.Delay:   {SynchDomain.Time: 0.1, SynchDomain.Position: 0.5},
        SynchParam.Initial: {SynchDomain.Time: None, SynchDomain.Position: 0},
        SynchParam.Active:  {SynchDomain.Time: 0.1, SynchDomain.Position: 1},
        SynchParam.Total:   {SynchDomain.Time: 1, SynchDomain.Position: 10},
        SynchParam.Repeats: 10,
    }
]
```

Start a trigger

When an order comes for sardana to start a trigger, sardana will call the `StartOne()` method. This method receives an axis as parameter. The controller code should trigger the hardware acquisition.

Here is an example of the possible implementation of `StartOne()`:

```
class SpringfieldTriggerGateController(TriggerGateController):

    def StartOne(self, axis):
        self.springfield.StartChannel(axis)
```

As soon as `StartOne()` is invoked, sardana expects the trigger to be running. It enters a high frequency synchronization loop which asks for the trigger state through calls to `StateOne()`. It will keep the loop running as long as the controller responds with `State.Moving`. If `StateOne()` raises an exception or returns something other than `State.Moving`, sardana will assume the trigger is stopped and exit the synchronization loop.

For an synchronization to work properly, it is therefore, **very important** that `StateOne()` responds correctly.

Stop a trigger

It is possible to stop a trigger when it is running. When sardana is ordered to stop a trigger synchronization, it invokes the `StopOne()` method. This method receives an axis parameter. The controller should make sure the desired trigger is *gracefully* stopped.

Here is an example of the possible implementation of `StopOne()`:

```
class SpringfieldTriggerGateController(TriggerGateController):

    def StopOne(self, axis):
        self.springfield.StopChannel(axis)
```

Abort a trigger

In an emergency situation, it is desirable to abort a synchronization *as fast as possible*. When sardana is ordered to abort a trigger synchronization, it invokes the `AbortOne()` method. This method receives an axis parameter. The controller should make sure the desired trigger is stopped as fast as it can be done.

Here is an example of the possible implementation of `AbortOne()`:

```
class SpringfieldTriggerGateController(TriggerGateController):

    def AbortOne(self, axis):
        self.springfield.AbortChannel(axis)
```

How to write an I/O register controller

The basics

Todo: document IORegister controller howto

How to write a pseudo motor controller

The basics

Todo: document pseudo motor controller howto

How to write a pseudo counter controller

The basics

An example of a X-ray beam position monitor (XBPM) pseudo counter controller will be build incrementally from scratch to aid in the explanation. Its purpose is to provide an easy feedback about the beam position in the vertical and horizontal axes as well as the total intensity of the beam.

By now you should have read the general controller basics chapter. Let's start from writing a `PseudoCounterController` subclass with a proper constructor and the roles defined.

```
from sardana.pool.controller import PseudoCounterController

class XBPMPseudoCounterController(PseudoCounterController):

    counter_roles = ('top', 'bottom', 'right', 'left')
    pseudo_counter_roles = ('vertical', 'horizontal', 'total')

    def __init__(self, inst, props, *args, **kwargs):
        super(XBPMPseudoCounterController, self).__init__(inst, props, *args,␣
→**kwargs)
```

The `counter_roles` and `pseudo_counter_roles` tuples contains names of the counter and pseudo counter roles respectively. These names are used when creating the controller instance and their order is important when writing the controller itself. Each controller will define its own roles.

The constructor does nothing apart of calling the parent class constructor but could be used to implement any necessary initialization.

The pseudo counter calculations are implemented in the `calc()` method:

```
def calc(self, index, counter_values):
    top, bottom, right, left = counter_values

    if index == 1: # vertical
        vertical = (top - bottom)/(top + bottom)
        return vertical
    elif index == 2: # horizontal
        horizontal = (right - left)/(right + left)
        return horizontal
    elif index == 3: # total
        total = (top + bottom + right + left) / 4
        return total
```

From the implementation we can conclude that the vertical pseudo counter will give values from -1 to 1 depending on the beam position in the vertical dimension. If the beam passes closer to the top sensor, the value will be more positive. If the beam passes closer to the bottom sensor the value will be more negative. The value close to the zero indicates the beam centered in the middle. Similarly behaves the horizontal pseudo counter. The total pseudo counter is the mean value of all the four sensors and indicates the beam intensity.

Including external variables in the calculation

The pseudo counter calculation may require an arbitrary variable which is not a counter value. One can use Taurus[219] or PyTango[220] libraries in order to read their attributes and use them in the calculation. It is even possible to write pseudo counters not based at all on the counters. In this case it is enough to define an empty `counter_roles` tuple.

Writing recorders

Overview

Sardana macros may produce data and users are usually interested in storing or visualizing it. Sardana delegates this work to the recorders. A good example of the recorder usage are the scan macros developed with the *Scan Framework*. Recorders are in charge of writing data to its destinations, for example a file, the Spock output or to plot it on a graph.

What is a recorder?

Recorder class is a Sardana element managed by the MacroServer. It is identified by its name, and is located in a recorder library - another Sardana element which is also identified by its name. Recorders are developed as Python classes, and recorder libraries are just Python modules aggregating these classes.

[219] http://packages.python.org/taurus/
[220] http://packages.python.org/PyTango/

Type of recorders

Sardana defines some standard recorders e.g. the Spock output recorder or the SPEC file recorder. From the other hand users may define their custom recorders. Sardana provides the following standard recorders (grouped by types):

- **file [*]**
    - FIO_FileRecorder
    - NXscan_FileRecorder
    - SPEC_FileRecorder
- **shared memory [*]**
    - SPSRecorder
    - ShmRecorder
- **output**
    - JsonRecorder [*]
    - OutputRecorder

[*] Scan Framework provides mechanisms to enable and select this recorders using the environment variables.

Writing a custom recorder

Todo: document how to write custom recorders

Configuration

Custom recorders may be added to the Sardana system by placing the recorder library module in a directory which is specified by the MacroServer *RecorderPath* property. RecorderPath property may contain an ordered, colon-separated list of directories. In case of overriding recorders by name or by file extension (in case of the file recorders), recorders located in the first paths are of higher priority than the ones from the last paths.

Three types of overriding may occur:

By recorder library name If Python modules with the same name are located in different directories, the library located in the the higher priority directory will be loaded.

By recorder name If two recorder classes with the same name appear in two different modules, only the recorder from the library located in the higher priority module will be loaded. If both modules are located in the same directory, the behavior is undetermined.

By file extension If two different recorders supporting the same file extension appear in two different modules, the one from the higher priority path will be used when selection is based on the extension (but both will be available for the selection by name). If both of these recorders' modules are located in the same directory, the system will assign a list of recorders to a given extension. Then the application is in charge of deciding which one to use.

As previously mentioned recorders are selectable by either the recorder name or the extension. During the MacroServer startup the extension to recorder map is generated while loading the recorder libraries. This dynamically created map may be overridden by the custom map defined in the *sardanacustomsettings* module (SCAN_RECORDER_MAP variable with a dictionary where key is the scan file extension e.g. ".h5" and value is the recorder name e.g. "MyCustomRecorder", where both keys and values are of type string). The SCAN_RECORDER_MAP will make an union with the dynamically created map taking precedence in case the extensions repeats in both of them.

Sardana Testing

Sardana Testing

Sardana Test Framework

A testing framework allowing to test the Sardana features is included with the Sardana distribution. It is useful for test-driven development and it allows to find bugs in the code.

The first implementation of the Framework is an outcome of the Sardana Enhancement Proposal 5 (SEP5)[221].

Ideally, whenever possible, bug reports should be accompanied by a test revealing the bug.

The first tests implemented are focused on Unit Tests, but the same framework should be used for integration and system tests as well.

The sardana.test module includes testsuite.py. This file provides an auto-discovering suite for all tests implemented in Sardana.

The following are some key points to keep in mind when using this framework:

- The Sardana Test Framework is based on `unittest`[222] which should be imported from `taurus.external` in order to be compatible with all versions of python supported by Taurus.
- all test-related code is contained in submodules named *test* which appear in any module of Sardana.
- **test-related code falls in one of these three categories:**
  - Actual test code (classes that derive from unittest.TestCase)
  - Utility classes/functions (code to simplify development of test code)
  - Resources (accessory files required by some test). They are located in subdirectories named *res* situated inside the folders named *test*.

For a more complete description of the conventions on how to write tests with the Sardana Testing Framework, please refer to the [SEP5](http://sourceforge.net/p/sardana/wiki/SEP5/).

Sardana Test Framework for testing macros

Sardana Test Framework provides tools for testing macros. These tools come from sardana.macroserver.macros.test module

Tests meant to be incorporated in the Sardana distribution must be portable. For this reason it is strongly encouraged to use only elements created by the sar_demo macro. Only in the case where this is not possible, one may create specific elements for a test; these elements must be removed at the end of the test execution (e.g. using the tearDown method).

[221] http://sourceforge.net/p/sardana/wiki/SEP5/
[222] https://docs.python.org/dev/library/unittest.html#module-unittest

The module `sardana.macroserver.macros.test` provides utilities to simplify the tests for macro execution and macro stop. Macro test classes can inherit from `RunMacroTestCase`, `RunStopMacroTestCase` or `BaseMacroTestCase`.

Another utility provided is the option to execute the same test with many different macro input parameters. This is done by decorating the test class with any of the decorators of the the macro tests family.

This decorator is provided by `sardana.macroserver.macros.test`.

Specificities:

- Macros such as 'lsm' inherit from RunMacroTestCase as it is interesting to test if the macros can be executed. Helper methods (such as `RunMacroTestCase.macro_runs()`) can be overriden when programming new test cases. New helpers can be created as well.
- Scan macros inherits from RunStopMacroTestCase as it is interesting to test both: if the macros can be executed and if they can be aborted.

Links

For a more complete description of the conventions used when writing tests, see: http://sourceforge.net/p/sardana/wiki/SEP5/

For more information about unittest framework: http://docs.python.org/2/library/unittest.html

Run tests from command line

Run test suite

Running the Sardana test suite from command line can be done in two different ways:

1. Sardana tests can be executed using the *setuptools* test command prior to the installation by executing the following command from within the sardana project directory:

    python setup.py test

   This will execute only a subset of all the sardana tests - the unit test suite. The functional tests, that require the *sar_demo test environment*, are excluded on purpose.

2. The complete Sardana test suite, that includes the unit and the functional tests can be executed only after the Sardana installation by executing the *sardanatestsuite* script.

Run a single test

Executing a single test from command line is done by doing:

    python -m unittest test_name

Where test_name is the test module that has to be run.

That can be done with more verbosity by indicating the option -v.

    python -m unittest -v test_name

sar_demo test environment

Some of the Sardana tests e.g. the ones that test the macros, require a running Sardana instance with the sar_demo macro executed previosly. By default the tests will try to connect to the *door/demo1/1* door in order to run the macros there. The default door name can be changed in the *sardanacustomsettings* module.

Test-driven development example

In this section it is presented a practical example of how to code a macro by doing test-driven development thanks to the tools provided by the Sardana Test Framework.

Consider that we want to write a new macro named "sqrtmac" for calculating the square root of an input number. The "sqrtmac" specifications are:

1. Its data must be given in the form {'in':x,'out':s}
2. Its output ('out') must be the square root of the input data ('in').
3. Macro must raise an Exception of type ValueError if negative numbers are given as input.

Test development

First we design the tests according to the specifications considering the features that are required for the macro. For doing so we will need some imports in order to be able to use the base classes and decorators. In this case the important base class is RunMacroTestCase, and we import testRun and testFail to be used as decorators:

```
"""Tests for sqrt macro"""
import numpy as np
import unittest
from sardana.macroserver.macros.test import RunMacroTestCase, testRun, testFail
```

Now we will write a basic test, that will check the execution of the sqrtmac for a given input x = 12345.678. For doing so, we inherit from unittest and from RunMacroTestCase. In this implementation we will calculate in the test the sqrt of the input parameter and then, using assertEqual, we will verify that this value is equal to the output of the macro. The helper method macro_runs is used for executing the macro:

```
"""Tests for a macro calculating the sqrt of an input number"""
import numpy as np
import unittest
from sardana.macroserver.macros.test import RunMacroTestCase, testRun, testFail


class sqrtmacTest(RunMacroTestCase, unittest.TestCase):
    """Test of sqrt macro. It verifies that macro sqrt can be executed.
    """
    macro_name = "sqrtmac"

    def test_sqrtmac(self):

        macro_params = [str(x)]
        self.macro_runs(macro_params)

        data=self.macro_executor.getData()
        expected_output = 49
```

(continues on next page)

(continued from previous page)

```
        msg = 'Macro output does not equals the expected output'
        self.assertEqual(data['in'] ,float(macro_params[0]), msg)
        self.assertEqual(data['out'] ,expected_output, msg)
```

Now, two new tests are added thanks to the decorator and the helper functions. In this case we will use the decorator @testRun. The same test case can be launched with different sets of parameters. One decorator is used for each set of parameters.

One of the tests will run the sqrtmac macro for an input value of 9 and verify that the macro has been executed without problems.

Another test added will run the sqrt for an input of 2.25 and will verify its input and output values against the expected values which we pass to the decorator. A wait_timeout of 5s will be given; this means, that if the test does not finish within 5 seconds, the current test will give an error and the following test will be executed:

```
"""Tests for a macro calculating the sqrt of an input number"""
import numpy as np
import unittest
from sardana.macroserver.macros.test import RunMacroTestCase, testRun, testFail


@testRun(macro_params=['9'])
@testRun(macro_params=['2.25'], data={'in':2.25,'out':1.5}, wait_timeout=5)
class sqrtmacTest(RunMacroTestCase, unittest.TestCase):
    """Test of sqrt macro. It verifies that macro sqrt can be executed.
    """
    macro_name = "sqrtmac"

    def test_sqrtmac(self):

        macro_params = [str(x)]
        self.macro_runs(macro_params)

        data=self.macro_executor.getData()
        expected_output = 49

        msg = 'Macro output does not equals the expected output'
        self.assertEqual(data['in'] ,float(macro_params[0]), msg)
        self.assertEqual(data['out'] ,expected_output, msg)
```

The following test implemented must check that the macro is raising an Exception if negative numbers are passed as input. The type of exception raised must be a ValueError. For developing this test we will use the decorator testFail which allows to test if a macro is raising an Exception before finishing its execution. The final implementation of our test file test_sqrt.py is as follows:

```
"""Tests for a macro calculating the sqrt of an input number"""
import numpy as np
import unittest
from sardana.macroserver.macros.test import RunMacroTestCase, testRun, testFail

@testRun(macro_params=['9'])
@testRun(macro_params=['2.25'], data={'in':2.25,'out':1.5}, wait_timeout=5)
@testFail(macro_params=['-3.0'], exception=ValueError, wait_timeout=5)
class sqrtmacTest(RunMacroTestCase, unittest.TestCase):
```

(continues on next page)

(continued from previous page)

```
    """Test of sqrt macro. It verifies that macro sqrt can be executed.
    """
    macro_name = "sqrtmac"

    def test_sqrtmac(self):

        macro_params = [str(x)]
        self.macro_runs(macro_params)

        data=self.macro_executor.getData()
        expected_output = 49

        msg = 'Macro output does not equals the expected output'
        self.assertEqual(data['in'] ,float(macro_params[0]), msg)
        self.assertEqual(data['out'] ,expected_output, msg)
```

Macro development

Thanks to the test that we have designed precedently we can now implement the macro and check if it is developed according to the specifications.

We do a first implementation of the macro by calculating the square root of an input number. Then we will execute the test and analyze the results. The first implementation looks like this:

```
import numpy as np
from sardana.macroserver.macro import Macro, Type

class sqrtmac(Macro):
    """Macro sqrtmac"""

    param_def = [ [ "value", Type.Float, 9,
                    "input value for which we want the square root"] ]
    result_def = [ [ "result", Type.Float, None,
                     "square root of the input value"] ]

    def run (self, n):
        ret = np.sqrt(n)
        return ret
```

An its ouput on the screen:

```
sardana/src/sardana/macroserver/macros/test> python -m unittest -v test_sqrtmac
test_sqrtmac (test_sqrtmac.sqrtmacTest) ... ERROR
test_sqrtmac_macro_fails (test_sqrtmac.sqrtmacTest)
Testing sqrtmac with macro_fails(macro_params=['-3.0'], exception=<type 'exceptions.
↪ValueError'>, wait_timeout=5) ... FAIL
test_sqrtmac_macro_runs (test_sqrtmac.sqrtmacTest)
Testing sqrtmac with macro_runs(macro_params=['2.25'], wait_timeout=5, data={'out': 1.
↪5, 'in': 2.25}) ... ERROR
test_sqrtmac_macro_runs_2 (test_sqrtmac.sqrtmacTest)
Testing sqrtmac with macro_runs(macro_params=['9']) ... ok

======================================================================
ERROR: test_sqrtmac (test_sqrtmac.sqrtmacTest)
----------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
            .
            .
            .
    desc = Exception: Macro 'sqrtmac' does not produce any data


======================================================================
ERROR: test_sqrtmac_macro_runs (test_sqrtmac.sqrtmacTest)
Testing sqrtmac with macro_runs(macro_params=['2.25'], wait_timeout=5, data={'out': 1.
↪5, 'in': 2.25})
----------------------------------------------------------------------
Traceback (most recent call last):
            .
            .
            .
    desc = Exception: Macro 'sqrtmac' does not produce any data


======================================================================
FAIL: test_sqrtmac_macro_fails (test_sqrtmac.sqrtmacTest)
Testing sqrtmac with macro_fails(macro_params=['-3.0'], exception=<type 'exceptions.
↪ValueError'>, wait_timeout=5)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/siciliarep/tmp/mrosanes/workspace/GIT/projects/sardana/src/sardana/
↪macroserver/macros/test/base.py", line 144, in newTest
    return helper(**helper_kwargs)
  File "/siciliarep/tmp/mrosanes/workspace/GIT/projects/sardana/src/sardana/
↪macroserver/macros/test/base.py", line 271, in macro_fails
    self.assertEqual(state, 'exception', msg)
AssertionError: Post-execution state should be "exception" (got "finish")

----------------------------------------------------------------------
Ran 4 tests in 0.977s

FAILED (failures=1, errors=2)
```

At this moment two tests are giving an error because 'sqrtmac' does not produce data, and one test is failing because the exception is not treat. The test that is giving 'Ok' is only testing that the macro can be executed.

The second step will be to set the input and output data of the macro and execute the test again:

```python
import numpy as np
from sardana.macroserver.macro import Macro, Type

class sqrtmac(Macro):
    """Macro sqrtmac"""

    param_def = [ [ "value", Type.Float, 9,
                    "input value for which we want the square root"] ]
    result_def = [ [ "result", Type.Float, None,
                     "square root of the input value"] ]

    def run (self, n):
        ret = np.sqrt(n)
        self.setData({'in':n,'out':ret})
```

(continues on next page)

(continued from previous page)

```
        return ret
```

An its ouput on the screen:

```
sardana/macroserver/macros/test> python -m unittest -v test_sqrtmac
test_sqrtmac (test_sqrtmac.sqrtmacTest) ... ok
test_sqrtmac_macro_fails (test_sqrtmac.sqrtmacTest)
Testing sqrtmac with macro_fails(macro_params=['-3.0'], exception=<type 'exceptions.
↪ValueError'>, wait_timeout=5) ... FAIL
test_sqrtmac_macro_runs (test_sqrtmac.sqrtmacTest)
Testing sqrtmac with macro_runs(macro_params=['2.25'], wait_timeout=5, data={'out': 1.
↪5, 'in': 2.25}) ... ok
test_sqrtmac_macro_runs_2 (test_sqrtmac.sqrtmacTest)
Testing sqrtmac with macro_runs(macro_params=['9']) ... ok

======================================================================
FAIL: test_sqrtmac_macro_fails (test_sqrtmac.sqrtmacTest)
Testing sqrtmac with macro_fails(macro_params=['-3.0'], exception=<type 'exceptions.
↪ValueError'>, wait_timeout=5)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/siciliarep/tmp/mrosanes/workspace/GIT/projects/sardana/src/sardana/
↪macroserver/macros/test/base.py", line 142, in newTest
    return helper(**helper_kwargs)
  File "/siciliarep/tmp/mrosanes/workspace/GIT/projects/sardana/src/sardana/
↪macroserver/macros/test/base.py", line 267, in macro_fails
    self.assertEqual(state, 'exception', msg)
AssertionError: Post-execution state should be "exception" (got "finish")

----------------------------------------------------------------------
Ran 4 tests in 0.932s

FAILED (failures=1)
```

As we can see, the test_sqrtmac_macro_fails is Failing, because the case of negative numbers is still not suppported. The rest of tests that are testing the execution and the expected output values are OK.

Finally we arrive to the complete implementation of the macro taking into account the Exception that should be raised if we enter a negative number as input parameter. For coding this macro test-driven development has been used:

```
import numpy as np
from sardana.macroserver.macro import Macro, Type

class sqrtmac(Macro):
    """Macro sqrtmac"""

    param_def = [ [ "value", Type.Float, 9,
                    "input value for which we want the square root"] ]
    result_def = [ [ "result", Type.Float, None,
                     "square root of the input value"] ]

    def run (self, n):
        if (n<0):
            raise ValueError("Negative numbers are not accepted.")
```

(continues on next page)

(continued from previous page)

```
        ret = np.sqrt(n)
        self.setData({'in':n,'out':ret})
        return ret
```

An the output on the console after executing the test looks like this:

```
sardana/macroserver/macros/test> python -m unittest -v test_sqrtmac
test_sqrtmac (test_sqrtmac.sqrtmacTest) ... ok
test_sqrtmac_macro_fails (test_sqrtmac.sqrtmacTest)
Testing sqrtmac with macro_fails(macro_params=['-3.0'], exception=<type 'exceptions.
↪ValueError'>, wait_timeout=5) ... ok
test_sqrtmac_macro_runs (test_sqrtmac.sqrtmacTest)
Testing sqrtmac with macro_runs(macro_params=['2.25'], wait_timeout=5, data={'out': 1.
↪5, 'in': 2.25}) ... ok
test_sqrtmac_macro_runs_2 (test_sqrtmac.sqrtmacTest)
Testing sqrtmac with macro_runs(macro_params=['9']) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.928s

OK
```

Sardana Unit Test Examples

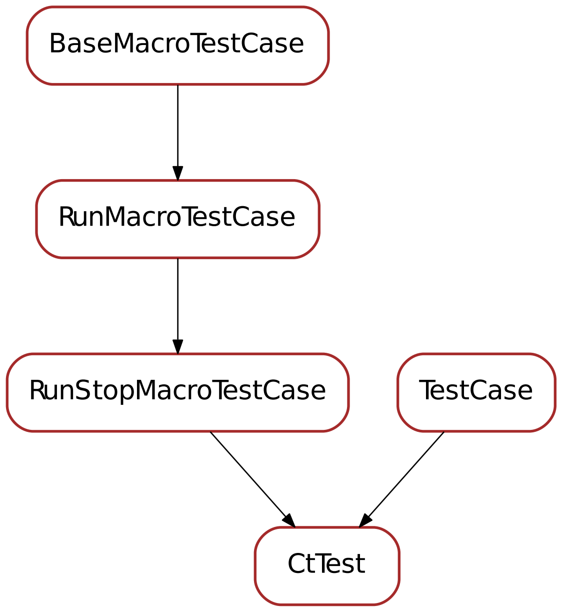
`test_ct`

Tests for ct macros

Classes

- *CtTest*

CtTest



class CtTest (**a, **kw*)

Test of ct macro. It verifies that macro ct can be executed. It inherits from RunStopMacroTestCase and from unittest.TestCase. It tests two executions of the ct macro with two different input parameters. Then it does another execution and it tests if the execution can be aborted.

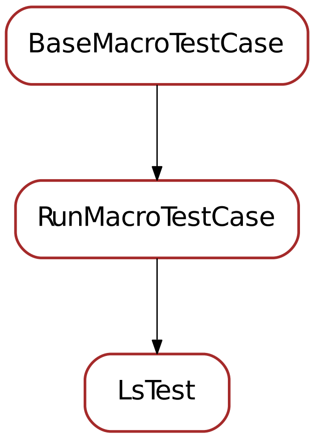
`test_list`

Tests for list macros

Classes

- *LsTest*
- *LsmTest*
- *LspmTest*
- *LsctrlTest*
- *LsctTest*
- *Ls0dTest*
- *Ls1dTest*
- *Ls2dTest*

LsTest



class LsTest

Base class for testing macros used to list elements. See *RunMacroTestCase* for requirements. LsTest use the lists of elem_type generated by *SarDemoEnv* as reference for compare with the output of the tested ls macro.

LsTest provide the class member:

- **elem_type (str): Type of the element to validate (mandatory).** Must be a valid type for *SarDemoEnv* class.

It provides the helper method:

- *check_elements()*

elem_type = None

check_elements(*list1*, *list2*)

A helper method to evaluate if all elements of list1 are in list2. :params list1: (seq<str>) List of elements to evaluate. :params list2: (seq<str>) List of elements for validate.

macro_runs(***kwargs*)

Reimplementation of macro_runs method for ls macros. It verifies that elements (elem_type) gotten by parsing the macro executor log output are in the correspondent list (elem_type) of SardanaEnv.

assertFinished(*msg*)

Asserts that macro has finished.

door_name = 'door/demo1/1'

macro_fails(*macro_name=None*, *macro_params=None*, *wait_timeout=inf*, *exception=None*)

Check that the macro fails to run for the given input parameters

Parameters

- **macro_name** – (str) macro name (takes precedence over macro_name class member)

- **macro_params** – (seq<str>) input parameters for the macro
- **wait_timeout** – maximum allowed time for the macro to fail. By default infinite timeout is used.
- **exception** – (str or Exception) if given, an additional check of the type of the exception is done. (IMPORTANT: this is just a comparison of str representations of exception objects)

macro_name = None

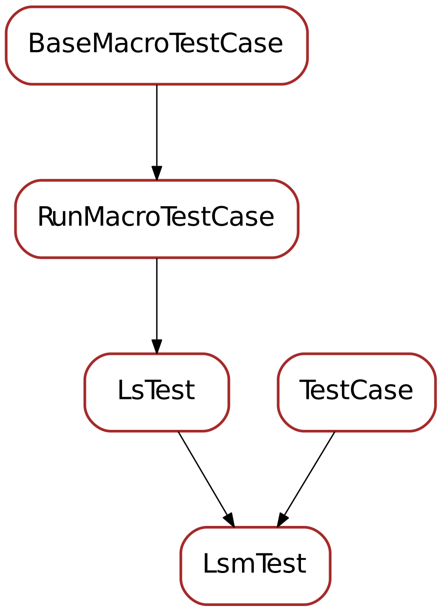
setUp()

Preconditions: - Those from `BaseMacroTestCase` - the macro executor registers to all the log levels

tearDown()

The macro_executor instance must be removed

LsmTest



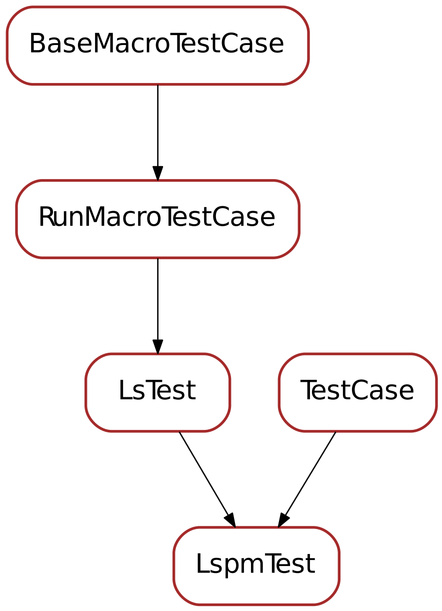
class **LsmTest**(**a, **kw*)

Class used for testing the 'lsm' macro. It verifies that all motors created by sar_demo are listed after execution of the macro 'lsm'.

macro_name = 'lsm'

elem_type = 'moveable'

LspmTest



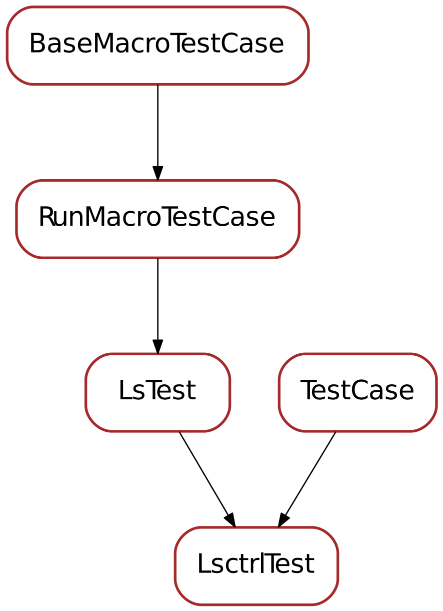
class LspmTest(**a, **kw*)

Class used for testing the 'lspm' macro. It verifies that all pseudomotors created by sar_demo are listed after execution of the macro 'lspm'.

macro_name = 'lspm'

elem_type = 'pseudomotor'

LsctrlTest



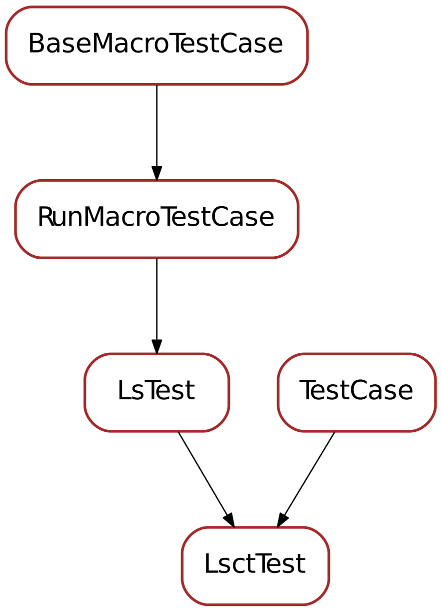
class LsctrlTest(**a, **kw*)

Class used for testing the 'lsctrl' macro. It verifies that all controllers created by sar_demo are listed after execution of the macro 'lsctrl'.

macro_name = 'lsctrl'

elem_type = 'controller'

LsctTest



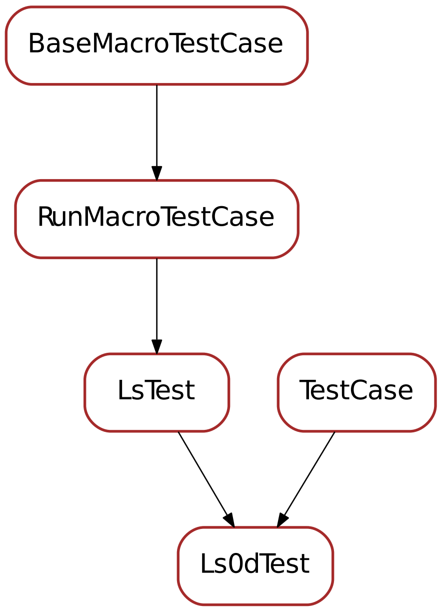
class LsctTest (**a, **kw*)

Class used for testing the 'lsct' macro. It verifies that all ct created by sar_demo are listed after execution of the macro 'lsct'.

macro_name = 'lsct'

elem_type = 'ctexpchannel'

Ls0dTest



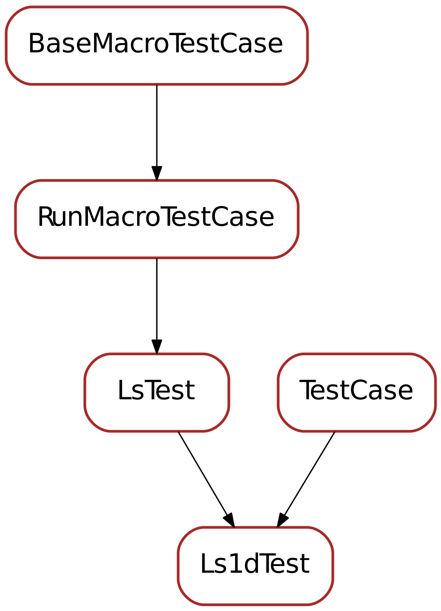
class Ls0dTest(**a, **kw*)

Class used for testing the 'ls0d' macro. It verifies that all 0d created by sar_demo are listed after execution of the macro 'ls0d'.

macro_name = 'ls0d'

elem_type = 'zerodexpchannel'

Ls1dTest



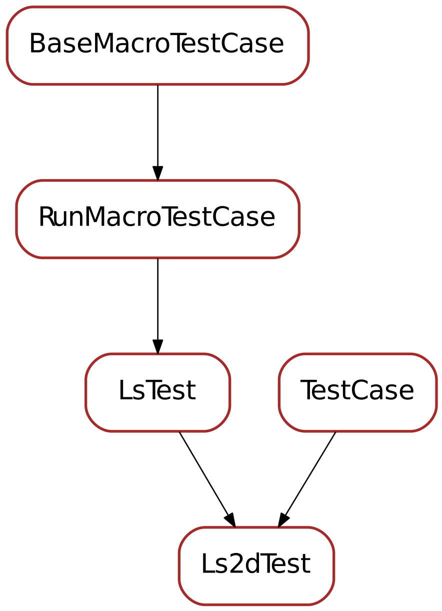
class Ls1dTest (**a, **kw*)

Class used for testing the 'ls1d' macro. It verifies that all 1d created by sar_demo are listed after execution of the macro 'ls1d'.

macro_name = 'ls1d'

elem_type = 'onedexpchannel'

Ls2dTest



class Ls2dTest (**a, **kw*)

Class used for testing the 'ls2d' macro. It verifies that all 2d created by sar_demo are listed after execution of the macro 'ls2d'.

macro_name = 'ls2d'

elem_type = 'twodexpchannel'

test_scan

Tests for scan macros

Functions

parsing_log_output (*log_output*)

A helper method to parse log output of an executed scan macro. :params log_output: (seq<str>) Result of macro_executor.getLog('output') (see description in *BaseMacroExecutor*).

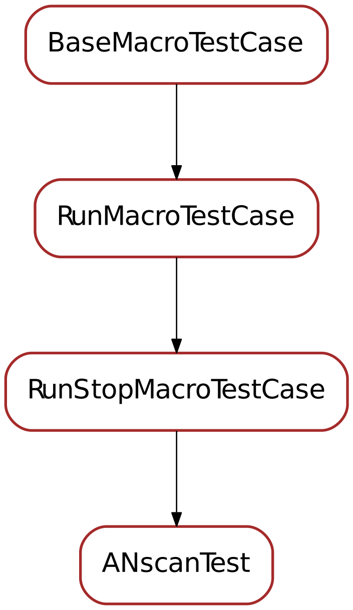
Returns (seq<number>) The numeric data of a scan.

Classes

- *ANscanTest*
- *DNscanTest*

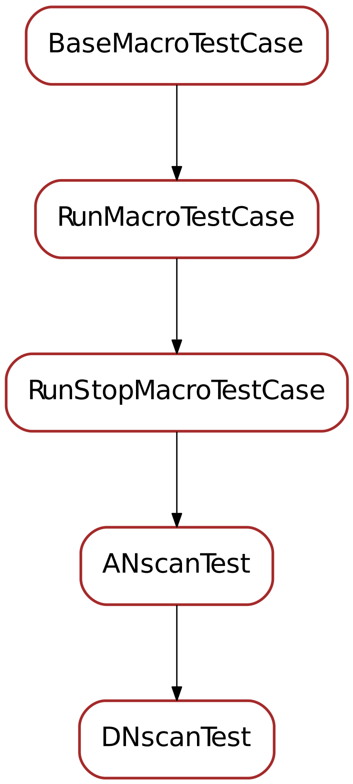
- *DNscancTest*
- *AscanTest*
- *DscanTest*
- *MeshTest*

ANscanTest



class ANscanTest
Not yet implemented. Once implemented it will test anscan. See *RunStopMacroTestCase* for requirements.

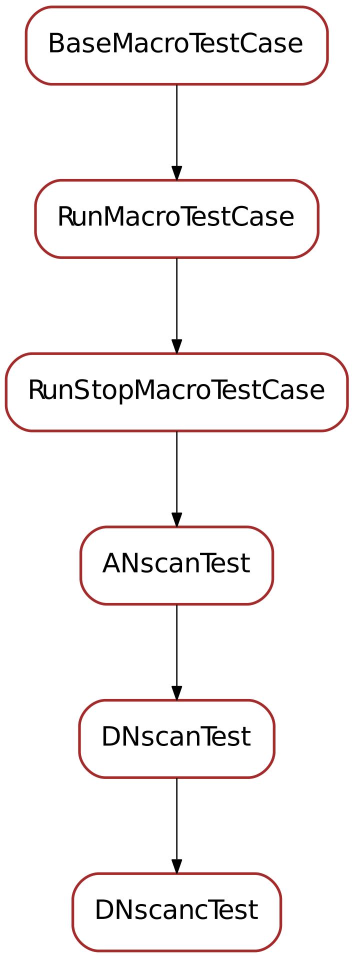
DNscanTest



class DNscanTest

Not yet implemented. Once implemented it will test the macro dnscanc. See *ANscanTest* for requirements.

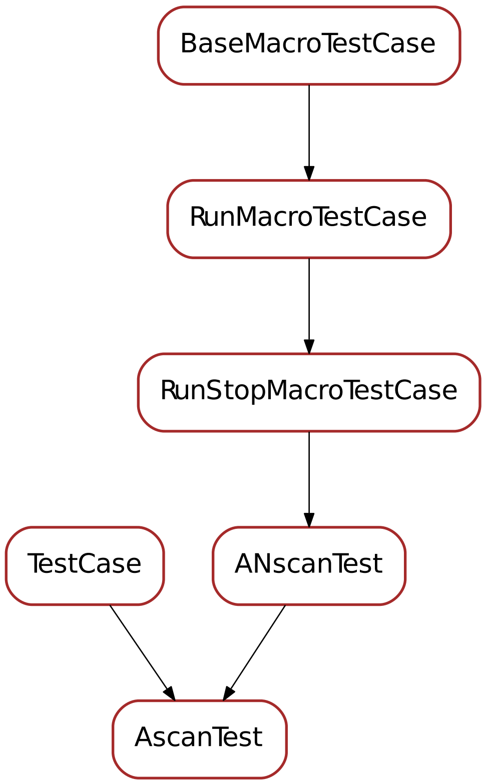
DNscancTest



class DNscancTest

Not yet implemented. Once implemented it will test the macro dnscanc. See *DNscanTest* for requirements.

AscanTest



class AscanTest(**a, **kw*)

Test of ascan macro. See `ANscanTest` for requirements. It verifies that macro ascan can be executed and stoped and tests the output of the ascan using data from log system and macro data.

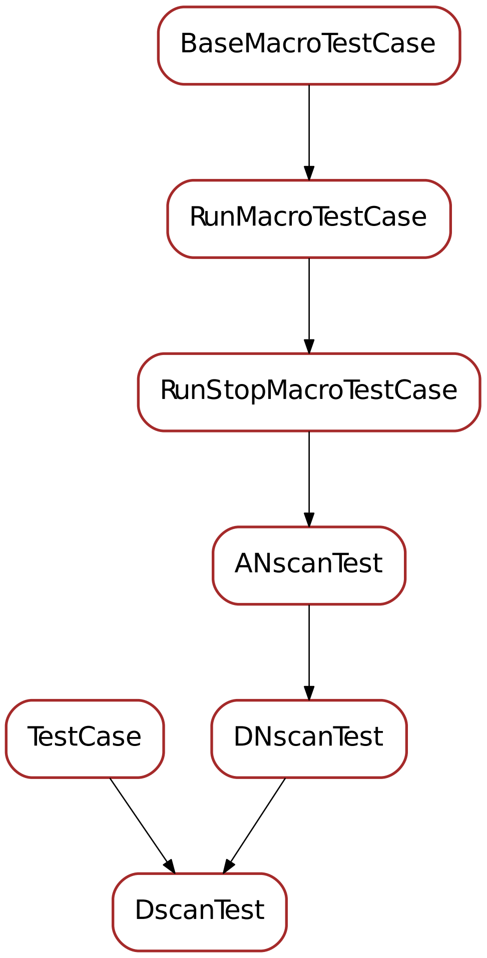
macro_name = 'ascan'

macro_runs(*macro_params=None, wait_timeout=30.0*)

Reimplementation of macro_runs method for ascan macro. It verifies using double checking, with log output and data from the macro:

- The motor initial and final positions of the scan are the ones given as input.
- Intervals in terms of motor position between one point and the next one are equidistant.

DscanTest

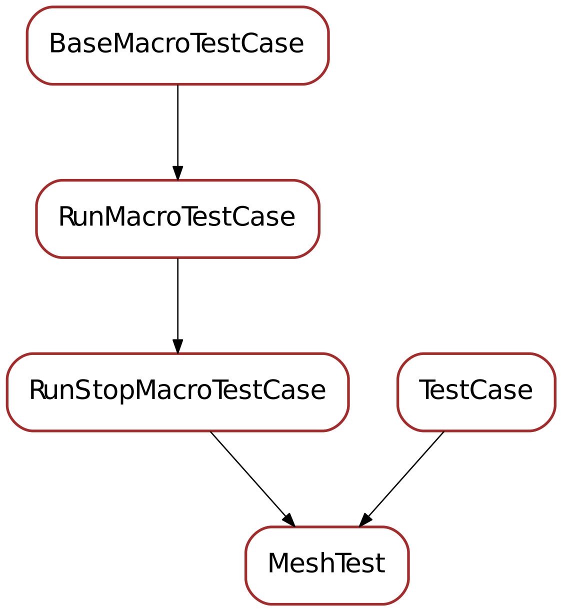


class DscanTest(**a*, ***kw*)

Test of dscan macro. It verifies that macro dscan can be executed and stoped. See `DNscanTest` for requirements.

macro_name = 'dscan'

MeshTest



class MeshTest (**a, **kw*)

Test of mesh macro. It verifies that macro mesh can be executed and stoped. See *RunStopMacroTestCase* for requirements.

macro_name = 'mesh'

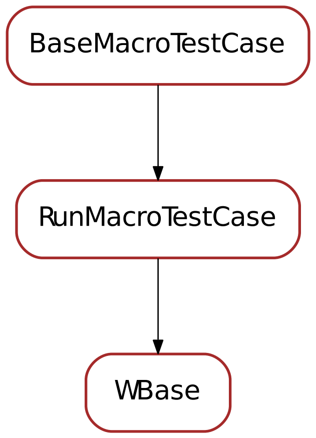
test_wm

Tests for wm macros

Classes

- *WBase*
- *WmTest*

WBase



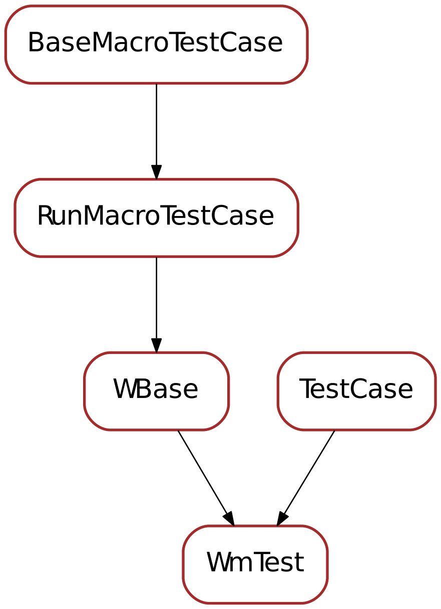
class WBase

Base class for testing macros used to read position.

macro_runs(***kw*)

Testing the execution of the 'wm' macro and verify that the log 'output' exists.

WmTest



class WmTest(**a*, ***kw*)

Test of wm macro. It verifies that the macro 'wm' can be executed. It inherits from WmBase and from unittest.TestCase. It tests the execution of the 'wm' macro and verifies that the log 'output' exists.

macro_name = 'wm'

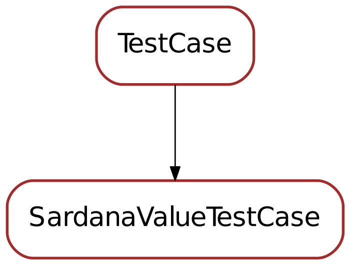
test_sardanavalue

Unit tests for sardanavalue module

Classes

- *SardanaValueTestCase*

SardanaValueTestCase



class SardanaValueTestCase(**a*, ***kw*)
Instantiating in different ways a Sardana Value and perform some verifications.

testInstanceCreation()
Instantiate in different ways a SardanaValue object.

testSardanaValueWithExceptionInfo()
Verify the creation of SardanaValue when exc_info != None. Verify that 'Error' is contained in the returned string.

testSardanaValueWithNoExceptionInfo()
Verify the creation of SardanaValue when exc_info is not specified and we give a value as argument of the SardanaValue constructor. SardanaValue representation shall contain its value.

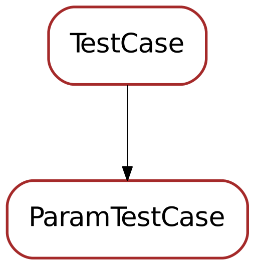
`test_parameter`

test_parameter module documentation

Classes

- *ParamTestCase*

ParamTestCase



class ParamTestCase(**a, **kw*)
Instantiate in different ways a Param object and verify that they are correct instances from the class Param.

testInstanceCreation()
Instantiate in different ways a Param object.

Sardana API

APIs

Macro API reference

Macro class

class Macro(**args, **kwargs*)
The Macro base class. All macros should inherit directly or indirectly from this class.

Init
internal variable

Running
internal variable

Pause
internal variable

Stop
internal variable

Fault
internal variable

Finished
internal variable

Ready
internal variable

Abort
internal variable

Exception
internal variable

All = 'All'
Constant used to specify all elements in a parameter

BlockStart = '<BLOCK>'
internal variable

BlockFinish = '</BLOCK>'
internal variable

param_def = []
This property holds the macro parameter description. It consists of a sequence of parameter information objects. A parameter information object is either:

1. a simple parameter object
2. a parameter repetition object

A simple parameter object is a sequence of:

1. a string representing the parameter name
2. a member of `Macro.Type` representing the parameter data type
3. a default value for the parameter or None if there is no default value
4. a string with the parameter description

Example:

```
param_def = ( ('value', Type.Float, None, 'a float parameter' ) )
```

A parameter repetition object is a sequence of:

1. a string representing the parameter repetition name
2. a sequence of parameter information objects
3. a dictionary representing the parameter repetition semantics or None to use the default parameter repetition semantics. Dictionary keys are:
   - *min* - integer representing minimum number of repetitions or None for no minimum.
   - *max* - integer representing maximum number of repetitions or None for no maximum.

   Default parameter repetition semantics is `{ 'min': 1, 'max' : None }` (in other words, "at least one repetition" semantics)

Example:

```
param_def = (
    ( 'motor_list', ( ( 'motor', Type.Motor, None, 'motor name') ), None,
↪'List of motors')
)
```

result_def = []
This property holds the macro result description. It a single parameter information object.

See also:

param_def

hints = {}
Hints to give a client to perform special tasks. Example: scan macros give hints on the types of hooks they support. A *GUI* can use this information to allow a scan to have sub-macros executed as hooks.

env = ()
a set of mandatory environment variable names without which your macro cannot run

interactive = False
decide if the macro should be able to receive input from the user [default: False]. A macro which asks input but has this flag set to False will print a warning message each time it is executed

run(**args*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

prepare(**args*, ***kwargs*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

on_abort()
Macro API. Hook executed when an abort occurs. Overwrite as necessary. Default implementation does nothing

on_pause()
Macro API. Hook executed when a pause occurs. Overwrite as necessary. Default implementation does nothing

on_stop()
Macro API. Hook executed when a stop occurs. Overwrite as necessary. Default implementation calls `on_abort()`

checkPoint(***kwargs*)
Macro API. Empty method that just performs a checkpoint. This can be used to check for the stop. Usually you won't need to call this method

pausePoint(***kwargs*)
Macro API. Will establish a pause point where called. If an external source as invoked a pause then, when this this method is called, it will be block until the external source calls resume. You may want to call this method if your macro takes a considerable time to execute and you may whish to pause it at some time. Example:

```
for i in range(10000):
    time.sleep(0.1)
    self.output("At step %d/10000", i)
    self.pausePoint()
```

Parameters timeout (`float`[223]) – timeout in seconds [default: None, meaning wait forever]

macros
Macro API. An object that contains all macro classes as members. With the returning object you can invoke other macros. Example:

```
m = self.macros.ascan('th', '0', '90', '10', '2')
scan_data = m.data
```

[223] https://docs.python.org/dev/library/functions.html#float

getMacroStatus(***kwargs*)

Macro API. Returns the current macro status. Macro status is a `dict`[224] where keys are the strings:

- *id* - macro ID (internal usage only)
- *range* - the full progress range of a macro (usually a `tuple`[225] of two numbers (0, 100))
- *state* - the current macro state, a string which can have values *start*, *step*, *stop* and *abort*
- *step* - the current step in macro. Should be a value inside the allowed macro range

Returns the macro status

Return type `dict`[226]

getName(***kwargs*)

Macro API. Returns this macro name

Returns the macro name

Return type `str`[227]

getID(***kwargs*)

Macro API. Returns this macro id

Returns the macro id

Return type `str`[228]

getParentMacro(***kwargs*)

Macro API. Returns the parent macro reference.

Returns the parent macro reference or None if there is no parent macro

Return type *Macro*

getDescription(***kwargs*)

Macro API. Returns a string description of the macro.

Returns the string description of the macro

Return type `str`[229]

getParameters(***kwargs*)

Macro API. Returns a the macro parameters. It returns a list containning the parameters with which the macro was executed

Returns the macro parameters

Return type `list`[230]

getExecutor(***kwargs*)

Macro API. Returns the reference to the object that invoked this macro. Usually is a MacroExecutor object.

Returns the reference to the object that invoked this macro

[224] https://docs.python.org/dev/library/stdtypes.html#dict
[225] https://docs.python.org/dev/library/stdtypes.html#tuple
[226] https://docs.python.org/dev/library/stdtypes.html#dict
[227] https://docs.python.org/dev/library/stdtypes.html#str
[228] https://docs.python.org/dev/library/stdtypes.html#str
[229] https://docs.python.org/dev/library/stdtypes.html#str
[230] https://docs.python.org/dev/library/stdtypes.html#list

Return type `MacroExecutor`

getDoorObj(***kwargs*)
Macro API. Returns the reference to the Door that invoked this macro.

Returns the reference to the Door that invoked this macro.

Rype `Door`

getManager(***kwargs*)
Macro API. Returns the manager for this macro (usually a MacroServer)

Returns the MacroServer

Return type *MacroServer*

manager
Macro API. Returns the manager for this macro (usually a MacroServer)

Returns the MacroServer

Return type *MacroServer*

getMacroServer(***kwargs*)
Macro API. Returns the MacroServer for this macro

Returns the MacroServer

Return type *MacroServer*

macro_server
Macro API. Returns the MacroServer for this macro

Returns the MacroServer

Return type *MacroServer*

getDoorName(***kwargs*)
Macro API. Returns the string with the name of the Door that invoked this macro.

Returns the string with the name of the Door that invoked this macro.

Return type `str`[231]

getCommand(***kwargs*)
Macro API. Returns the string used to execute the macro. Ex.: 'ascan M1 0 1000 100 0.8'

Returns the macro command.

Return type `str`[232]

getDateString(***kwargs*)
Macro API. Helper method. Returns the current date in a string.

Parameters **time_format** (`str`[233]) – the format in which the date should be returned (optional, default value is '%a %b %d %H:%M:%S %Y'

Returns the current date

Return type `str`[234]

outputDate(***kwargs*)
Macro API. Helper method. Outputs the current date into the output buffer

[231] https://docs.python.org/dev/library/stdtypes.html#str
[232] https://docs.python.org/dev/library/stdtypes.html#str
[233] https://docs.python.org/dev/library/stdtypes.html#str
[234] https://docs.python.org/dev/library/stdtypes.html#str

Parameters **time_format** (str[235]) – (str) the format in which the date should be returned (optional, default value is '%a %b %d %H:%M:%S %Y'

sendRecordData(***kwargs*)
Macro API. Sends the given data to the RecordData attribute of the Door

Parameters **data** – (sequence) the data to be sent

plot(***kwargs*)
Macro API. Sends the plot command to the client using the 'RecordData' DevEncoded attribute. The data is encoded using the pickle -> BZ2 codec.

Parameters

- **args** – the plotting args
- **kwargs** – the plotting keyword args

pylab

pyplot

getData(***kwargs*)
Macro API. Returns the data produced by the macro.

Raises Exception if no data has been set before on this macro

Returns the data produced by the macro

Return type object[236]

setData(***kwargs*)
Macro API. Sets the data for this macro

Parameters **data** (object[237]) – new data to be associated with this macro

data
macro data

print(***kwargs*)
Macro API. Prints a message. Accepted *args* and *kwargs* are the same as print(). Example:

```
self.print("this is a print for macro", self.getName())
```

Note: you will need python >= 3.0. If you have python 2.x then you must include at the top of your file the statement:

```
from __future__ import print_function
```

input(***kwargs*)
Macro API. If args is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that.

Depending on which type of application you are running, some of the keywords may have no effect (ex.: spock ignores decimals when a number is asked).

Recognized kwargs:

[235] https://docs.python.org/dev/library/stdtypes.html#str
[236] https://docs.python.org/dev/library/functions.html#object
[237] https://docs.python.org/dev/library/functions.html#object

- data_type : [default: Type.String] specific input type. Can also specify a sequence of strings with possible values (use allow_multiple=True to say multiple values can be selected)
- key : [default: no default] variable/label to assign to this input
- unit: [default: no default] units (useful for GUIs)
- timeout : [default: None, meaning wait forever for input]
- default_value : [default: None, meaning no default value] When given, it must be compatible with data_type
- allow_multiple : [default: False] in case data_type is a sequence of values, allow multiple selection
- minimum : [default: None] When given, must be compatible with data_type (useful for GUIs)
- maximum : [default: None] When given, must be compatible with data_type (useful for GUIs)
- step : [default: None] When given, must be compatible with data_type (useful for GUIs)
- decimals : [default: None] When given, must be compatible with data_type (useful for GUIs)

Examples:

```
device_name = self.input("Which device name (%s)?", "tab separated")

point_nb = self.input("How many points?", data_type=Type.Integer)

calc_mode = self.input("Which algorithm?", data_type=["Average", "Integral",
↪"Sum"],
                       default_value="Average", allow_multiple=False)
```

output (***kwargs*)

Macro API. Record a log message in this object's output. Accepted *args* and *kwargs* are the same as `logging.Logger.log()`[238]. Example:

```
self.output("this is a print for macro %s", self.getName())
```

Parameters

- **msg** (`str`[239]) – the message to be recorded
- **args** – list of arguments
- **kwargs** – list of keyword arguments

log (***kwargs*)

Macro API. Record a log message in this object's logger. Accepted *args* and *kwargs* are the same as `logging.Logger.log()`[240]. Example:

```
self.debug(logging.INFO, "this is a info log message for macro %s", self.
↪getName())
```

Parameters

[238] https://docs.python.org/dev/library/logging.html#logging.Logger.log
[239] https://docs.python.org/dev/library/stdtypes.html#str
[240] https://docs.python.org/dev/library/logging.html#logging.Logger.log

- **level** (int[241]) – the record level
- **msg** (str[242]) – the message to be recorded
- **args** – list of arguments
- **kwargs** – list of keyword arguments

debug(***kwargs*)

Macro API. Record a debug message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.debug()[243]. Example:

```
self.debug("this is a log message for macro %s", self.getName())
```

Parameters

- **msg** (str[244]) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

info(***kwargs*)

Macro API. Record an info message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.info()[245]. Example:

```
self.info("this is a log message for macro %s", self.getName())
```

Parameters

- **msg** (str[246]) – the message to be recorded
- **args** – list of arguments
- **kwargs** – list of keyword arguments

warning(***kwargs*)

Macro API. Record a warning message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.warning()[247]. Example:

```
self.warning("this is a log message for macro %s", self.getName())
```

Parameters

- **msg** (str[248]) – the message to be recorded
- **args** – list of arguments
- **kwargs** – list of keyword arguments

[241] https://docs.python.org/dev/library/functions.html#int
[242] https://docs.python.org/dev/library/stdtypes.html#str
[243] https://docs.python.org/dev/library/logging.html#logging.Logger.debug
[244] https://docs.python.org/dev/library/stdtypes.html#str
[245] https://docs.python.org/dev/library/logging.html#logging.Logger.info
[246] https://docs.python.org/dev/library/stdtypes.html#str
[247] https://docs.python.org/dev/library/logging.html#logging.Logger.warning
[248] https://docs.python.org/dev/library/stdtypes.html#str

error(***kwargs*)
: **Macro API**. Record an error message in this object's logger. Accepted *args* and *kwargs* are the same as `logging.Logger.error()`[249]. Example:

```
self.error("this is a log message for macro %s", self.getName())
```

Parameters

- **msg** (`str`[250]) – the message to be recorded
- **args** – list of arguments
- **kwargs** – list of keyword arguments

critical(***kwargs*)
: **Macro API**. Record a critical message in this object's logger. Accepted *args* and *kwargs* are the same as `logging.Logger.critical()`[251]. Example:

```
self.critical("this is a log message for macro %s", self.getName())
```

Parameters

- **msg** (`str`[252]) – the message to be recorded
- **args** – list of arguments
- **kwargs** – list of keyword arguments

trace(***kwargs*)
: **Macro API**. Record a trace message in this object's logger.

Parameters

- **msg** – (str) the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

traceback(***kwargs*)
: **Macro API**. Logs the traceback with level TRACE on the macro logger.

stack(***kwargs*)
: **Macro API**. Logs the stack with level TRACE on the macro logger.

report(***kwargs*)
: **Macro API**. Record a log message in the sardana report (if enabled) with default level **INFO**. The msg is the message format string, and the args are the arguments which are merged into msg using the string formatting operator. (Note that this means that you can use keywords in the format string, together with a single dictionary argument.)

kwargs are the same as `logging.Logger.debug()`[253] plus an optional level kwargs which has default value **INFO**

Example:

[249] https://docs.python.org/dev/library/logging.html#logging.Logger.error
[250] https://docs.python.org/dev/library/stdtypes.html#str
[251] https://docs.python.org/dev/library/logging.html#logging.Logger.critical
[252] https://docs.python.org/dev/library/stdtypes.html#str
[253] https://docs.python.org/dev/library/logging.html#logging.Logger.debug

```
self.report("this is an official report of macro %s", self.getName())
```

Parameters

- **msg** (str[254]) – the message to be recorded
- **args** – list of arguments
- **kwargs** – list of keyword arguments

flushOutput (***kwargs*)
Macro API. Flushes the output buffer.

getMacroThread (***kwargs*)
Macro API. Returns the python thread where this macro is running

Returns the python thread where this macro is running

Return type threading.Thread[255]

getMacroThreadID (***kwargs*)
Macro API. Returns the python thread id where this macro is running

Returns the python thread id where this macro is running

Return type int[256]

createExecMacroHook (***kwargs*)
Macro API. Creates a hook that executes the macro given as a sequence of strings where the first string is macro name and the following strings the macro parameters

Parameters

- **par_str_sequence** – the macro parameters
- **parent_macro** – the parent macro object. If None is given (default) then the parent macro is this macro

Returns a ExecMacroHook object (which is a callable object)

createMacro (***kwargs*)
Macro API. Create a new macro and prepare it for execution Several different parameter formats are supported:

```
# several parameters:
self.execMacro('ascan', 'th', '0', '100', '10', '1.0')
self.execMacro('mv', [[motor.getName(), '0']])
self.execMacro('mv', motor.getName(), '0') # backwards compatibility - see␣
↪note
self.execMacro('ascan', 'th', 0, 100, 10, 1.0)
self.execMacro('mv', [[motor.getName(), 0]])
self.execMacro('mv', motor.getName(), 0) # backwards compatibility - see note
th = self.getObj('th')
self.execMacro('ascan', th, 0, 100, 10, 1.0)
self.execMacro('mv', [[th, 0]])
self.execMacro('mv', th, 0)  # backwards compatibility - see note

# a sequence of parameters:
```

(continues on next page)

[254] https://docs.python.org/dev/library/stdtypes.html#str
[255] https://docs.python.org/dev/library/threading.html#threading.Thread
[256] https://docs.python.org/dev/library/functions.html#int

(continued from previous page)

```
self.execMacro(['ascan', 'th', '0', '100', '10', '1.0')
self.execMacro(['mv', [[motor.getName(), '0']]])
self.execMacro(['mv', motor.getName(), '0']) # backwards compatibility - see␣
↪note
self.execMacro(('ascan', 'th', 0, 100, 10, 1.0))
self.execMacro(['mv', [[motor.getName(), 0]]])
self.execMacro(['mv', motor.getName(), 0]) # backwards compatibility - see␣
↪note
th = self.getObj('th')
self.execMacro(['ascan', th, 0, 100, 10, 1.0])
self.execMacro(['mv', [[th, 0]]])
self.execMacro(['mv', th, 0]) # backwards compatibility - see note


# a space separated string of parameters (this is not compatible
# with multiple or nested repeat parameters, furthermore the repeat
# parameter must be the last one):
self.execMacro('ascan th 0 100 10 1.0')
self.execMacro('mv %s 0' % motor.getName())
```

Note: From Sardana 2.0 the repeat parameter values must be passed as lists of items. An item of a repeat parameter containing more than one member is a list. In case when a macro defines only one repeat parameter and it is the last parameter, for the backwards compatibility reasons, the plain list of items' members is allowed.

Parameters **pars** – the command parameters as explained above

Returns a sequence of two elements: the macro object and the result of preparing the macro

Return type `tuple`[257]<*Macro*, seq<obj>>

prepareMacroObj(***kwargs*)

Macro API. Prepare a new macro for execution

Parameters

- **name** (*macro_name_or_klass*) – name of the macro to be prepared or the macro class itself
- **pars** – list of parameter objects
- **init_opts** – keyword parameters for the macro constructor
- **prepare_opts** – keyword parameters for the macro prepare

Returns a sequence of two elements: the macro object and the result of preparing the macro

prepareMacro(***kwargs*)

Macro API. Prepare a new macro for execution Several different parameter formats are supported:

[257] https://docs.python.org/dev/library/stdtypes.html#tuple

```
# several parameters:
self.execMacro('ascan', 'th', '0', '100', '10', '1.0')
self.execMacro('mv', [[motor.getName(), '0']])
self.execMacro('mv', motor.getName(), '0') # backwards compatibility - see␣
↪note
self.execMacro('ascan', 'th', 0, 100, 10, 1.0)
self.execMacro('mv', [[motor.getName(), 0]])
self.execMacro('mv', motor.getName(), 0) # backwards compatibility - see note
th = self.getObj('th')
self.execMacro('ascan', th, 0, 100, 10, 1.0)
self.execMacro('mv', [[th, 0]])
self.execMacro('mv', th, 0) # backwards compatibility - see note

# a sequence of parameters:
self.execMacro(['ascan', 'th', '0', '100', '10', '1.0')
self.execMacro(['mv', [[motor.getName(), '0']]])
self.execMacro(['mv', motor.getName(), '0']) # backwards compatibility - see␣
↪note
self.execMacro(('ascan', 'th', 0, 100, 10, 1.0))
self.execMacro(['mv', [[motor.getName(), 0]]])
self.execMacro(['mv', motor.getName(), 0]) # backwards compatibility - see␣
↪note
th = self.getObj('th')
self.execMacro(['ascan', th, 0, 100, 10, 1.0])
self.execMacro(['mv', [[th, 0]]])
self.execMacro(['mv', th, 0]) # backwards compatibility - see note

# a space separated string of parameters (this is not compatible
# with multiple or nested repeat parameters, furthermore the repeat
# parameter must be the last one):
self.execMacro('ascan th 0 100 10 1.0')
self.execMacro('mv %s 0' % motor.getName())
```

Note: From Sardana 2.0 the repeat parameter values must be passed as lists of items. An item of a repeat parameter containing more than one member is a list. In case when a macro defines only one repeat parameter and it is the last parameter, for the backwards compatibility reasons, the plain list of items' members is allowed.

Parameters

- **args** – the command parameters as explained above
- **kwargs** – keyword optional parameters for prepare

Returns a sequence of two elements: the macro object and the result of preparing the macro

runMacro(***kwargs*)

Macro API. Runs the macro. This the lower level version of `execMacro()`. The method only returns after the macro is completed or an exception is thrown. It should be used instead of execMacro when some operation needs to be done between the macro preparation and the macro execution. Example:

```
macro = self.prepareMacro("mymacro", "myparam")
self.do_my_stuff_with_macro(macro)
```

(continues on next page)

(continued from previous page)

```
self.runMacro(macro)
```

Parameters **macro_obj** – macro object

Returns macro result

execMacroObj (***kwargs*)

Macro API. Execute a macro in this macro. The method only returns after the macro is completed or an exception is thrown. This is a higher level version of runMacro method. It is the same as:

```
macro = self.prepareMacroObjs(name, *args, **kwargs)
self.runMacro(macro)
return macro
```

Parameters

- **name** (*str*[258]) – name of the macro to be prepared
- **args** – list of parameter objects
- **kwargs** – list of keyword parameters

Returns a macro object

execMacro (***kwargs*)

Macro API. Execute a macro in this macro. The method only returns after the macro is completed or an exception is thrown. Several different parameter formats are supported:

```
# several parameters:
self.execMacro('ascan', 'th', '0', '100', '10', '1.0')
self.execMacro('mv', [[motor.getName(), '0']])
self.execMacro('mv', motor.getName(), '0') # backwards compatibility - see␣
↪note
self.execMacro('ascan', 'th', 0, 100, 10, 1.0)
self.execMacro('mv', [[motor.getName(), 0]])
self.execMacro('mv', motor.getName(), 0) # backwards compatibility - see note
th = self.getObj('th')
self.execMacro('ascan', th, 0, 100, 10, 1.0)
self.execMacro('mv', [th, 0]])
self.execMacro('mv', th, 0)  # backwards compatibility - see note

# a sequence of parameters:
self.execMacro(['ascan', 'th', '0', '100', '10', '1.0')
self.execMacro(['mv', [[motor.getName(), '0']]])
self.execMacro(['mv', motor.getName(), '0']) # backwards compatibility - see␣
↪note
self.execMacro(('ascan', 'th', 0, 100, 10, 1.0))
self.execMacro(['mv', [[motor.getName(), 0]]])
self.execMacro(['mv', motor.getName(), 0]) # backwards compatibility - see␣
↪note
th = self.getObj('th')
self.execMacro(['ascan', th, 0, 100, 10, 1.0])
self.execMacro(['mv', [[th, 0]]])
self.execMacro(['mv', th, 0]) # backwards compatibility - see note
```

(continues on next page)

[258] https://docs.python.org/dev/library/stdtypes.html#str

(continued from previous page)

```
# a space separated string of parameters (this is not compatible
# with multiple or nested repeat parameters, furthermore the repeat
# parameter must be the last one):
self.execMacro('ascan th 0 100 10 1.0')
self.execMacro('mv %s 0' % motor.getName())
```

Note: From Sardana 2.0 the repeat parameter values must be passed as lists of items. An item of a repeat parameter containing more than one member is a list. In case when a macro defines only one repeat parameter and it is the last parameter, for the backwards compatibility reasons, the plain list of items' members is allowed.

Parameters **pars** – the command parameters as explained above

Returns a macro object

getTangoFactory(***kwargs*)

Macro API. Helper method that returns the tango factory.

Returns the tango factory singleton

Return type TangoFactory[259]

getDevice(***kwargs*)

Macro API. Helper method that returns the device for the given device name

Returns the taurus device for the given device name

Return type TaurusDevice[260]

setLogBlockStart(***kwargs*)

Macro API. Specifies the begining of a block of data. Basically it outputs the 'BLOCK' tag

setLogBlockFinish(***kwargs*)

Macro API. Specifies the end of a block of data. Basically it outputs the '/BLOCK' tag

outputBlock(***kwargs*)

Macro API. Sends an line tagged as a block to the output

Parameters **line** (*str*[261]) – line to be sent

getPools(***kwargs*)

Macro API. Returns the list of known device pools.

Returns the list of known device pools

Return type seq<Pool>

addObj(***kwargs*)

Macro API. Adds the given object to the list of controlled objects of this macro. In practice it means that if a stop is executed the stop method of the given object will be called.

Parameters

- **obj** (*object*[262]) – the object to be controlled

[259] http://taurus-scada.org/devel/api/taurus/core/tango/_TangoFactory.html#taurus.core.tango.TangoFactory
[260] http://taurus-scada.org/devel/api/taurus/core/_TaurusDevice.html#taurus.core.TaurusDevice
[261] https://docs.python.org/dev/library/stdtypes.html#str
[262] https://docs.python.org/dev/library/functions.html#object

- **priority** (int[263]) – wheater or not reserve with priority [default: 0 meaning no priority]

addObjs(***kwargs*)

Macro API. Adds the given objects to the list of controlled objects of this macro. In practice it means that if a stop is executed the stop method of the given object will be called.

Parameters **obj_list** (sequence) – list of objects to be controlled

returnObj(*obj*)

Removes the given objects to the list of controlled objects of this macro.

Parameters **obj** – object to be released from the control

Return type object[264]

getObj(***kwargs*)

Macro API. Gets the object of the given type belonging to the given pool with the given name. The object (if found) will automatically become controlled by the macro.

Raises MacroWrongParameterType if name is not a string

Raises AttributeError if more than one matching object is found

Parameters

- **name** (str[265]) – string representing the name of the object. Can be a regular expression
- **type_class** – the type of object [default: All]
- **subtype** – a string representing the subtype [default: All] Ex.: if type_class is Type.ExpChannel, subtype could be 'CTExpChannel'
- **pool** – the pool to which the object should belong [default: All]
- **reserve** – automatically reserve the object for this macro [default: True]

Returns the object or None if no compatible object is found

getObjs(***kwargs*)

Macro API. Gets the objects of the given type belonging to the given pool with the given names. The objects (if found) will automatically become controlled by the macro.

Parameters

- **names** – a string or a sequence of strings representing the names of the objects. Each string can be a regular expression
- **type_class** – the type of object (optional, default is All). Example: Type.Motor, Type.ExpChannel
- **subtype** – a string representing the subtype (optional, default is All) Ex.: if type_class is Type.ExpChannel, subtype could be 'CTExpChannel'
- **pool** – the pool to which the object should belong (optional, default is All)
- **reserve** – automatically reserve the object for this macro (optional, default is True)

Returns a list of objects or empty list if no compatible object is found

[263] https://docs.python.org/dev/library/functions.html#int
[264] https://docs.python.org/dev/library/functions.html#object
[265] https://docs.python.org/dev/library/stdtypes.html#str

findObjs(***kwargs*)

Macro API. Gets the objects of the given type belonging to the given pool with the given names. The objects (if found) will automatically become controlled by the macro.

Parameters

- **names** – a string or a sequence of strings representing the names of the objects. Each string can be a regular expression
- **type_class** – the type of object (optional, default is All)
- **subtype** – a string representing the subtype [default: All] Ex.: if type_class is Type.ExpChannel, subtype could be 'CTExpChannel'
- **pool** – the pool to which the object should belong [default: All]
- **reserve** – automatically reserve the object for this macro [default: True]

Returns a list of objects or empty list if no compatible object is found

getMacroNames(***kwargs*)

Macro API. Returns a list of strings containing the names of all known macros

return a sequence of macro names

rtype seq<str[266]>

getMacros(***kwargs*)

Macro API. Returns a sequence of *MacroClass* / *MacroFunction* objects for all known macros that obey the filter expression.

Parameters filter – a regular expression for the macro name (optional, default is None meaning match all macros)

Returns a sequence of *MacroClass* / *MacroFunction* objects

Return type seq<*MacroClass* / *MacroFunction*>

getMacroLibraries(***kwargs*)

Macro API. Returns a sequence of *MacroLibrary* objects for all known macros that obey the filter expression.

Parameters filter – a regular expression for the macro library [default: None meaning match all macro libraries)

Returns a sequence of *MacroLibrary* objects

Return type seq<*MacroLibrary*>

getMacroLibrary(***kwargs*)

Macro API. Returns a *MacroLibrary* object for the given library name.

Parameters lib_name (str[267]) – library name

Returns a macro library *MacroLibrary*

Return type *MacroLibrary*

getMacroLib(***kwargs*)

Macro API. Returns a *MacroLibrary* object for the given library name.

Parameters lib_name (str[268]) – library name

[266] https://docs.python.org/dev/library/stdtypes.html#str
[267] https://docs.python.org/dev/library/stdtypes.html#str
[268] https://docs.python.org/dev/library/stdtypes.html#str

Returns a macro library `MacroLibrary`

Return type `MacroLibrary`

getMacroLibs(***kwargs*)
Macro API. Returns a sequence of `MacroLibrary` objects for all known macros that obey the filter expression.

Parameters **filter** – a regular expression for the macro library [default: None meaning match all macro libraries)

Returns a sequence of `MacroLibrary` objects

Return type seq<`MacroLibrary`>

getMacroInfo(***kwargs*)
Macro API. Returns the corresponding `MacroClass` /`MacroFunction` object.

Parameters **macro_name** (`str`[269]) – a string with the desired macro name.

Returns a `MacroClass` /`MacroFunction` object or None if the macro with the given name was not found

Return type `MacroClass` /`MacroFunction`

getMotion(***kwargs*)
Macro API. Returns a new Motion object containing the given elements.

Raises Exception if no elements are defined or the elems is not recognized as valid, or an element is not found or an element appears more than once

Parameters

- **elems** – list of moveable object names
- **motion_source** – obj or list of objects containing moveable elements. Usually this is a Pool object or a list of Pool objects (optional, default is None, meaning all known pools will be searched for the given moveable items
- **read_only** – not used. Reserved for future use
- **cache** – not used. Reserved for future use

Returns a Motion object

getElementsWithInterface(***kwargs*)

getControllers(***kwargs*)

getMoveables(***kwargs*)

getMotors(***kwargs*)

getPseudoMotors(***kwargs*)

getIORegisters(***kwargs*)

getMeasurementGroups(***kwargs*)

getExpChannels(***kwargs*)

getCounterTimers(***kwargs*)

get0DExpChannels(***kwargs*)

get1DExpChannels(***kwargs*)

[269] https://docs.python.org/dev/library/stdtypes.html#str

get2DExpChannels(***kwargs*)

getPseudoCounters(***kwargs*)

getInstruments(***kwargs*)

getElementWithInterface(***kwargs*)

getController(***kwargs*)

getMoveable(***kwargs*)

getMotor(***kwargs*)

getPseudoMotor(***kwargs*)

getIORegister(***kwargs*)

getMeasurementGroup(***kwargs*)

getExpChannel(***kwargs*)

getCounterTimer(***kwargs*)

get0DExpChannel(***kwargs*)

get1DExpChannel(***kwargs*)

get2DExpChannel(***kwargs*)

getPseudoCounter(***kwargs*)

getInstrument(***kwargs*)

getEnv(***kwargs*)

Macro API. Gets the local environment matching the given parameters:

- door_name and macro_name define the context where to look for the environment. If both are None, the global environment is used. If door name is None but macro name not, the given macro environment is used and so on…
- If key is None it returns the complete environment, otherwise key must be a string containing the environment variable name.

Raises UnknownEnv

Parameters

- **key** (*str*[270]) – environment variable name [default: None, meaning all environment]
- **door_name** (*str*[271]) – local context for a given door [default: None, meaning no door context is used]
- **macro_name** (*str*[272]) – local context for a given macro [default: None, meaning no macro context is used]

Returns a `dict`[273] containing the environment

Return type `dict`[274]

[270] https://docs.python.org/dev/library/stdtypes.html#str
[271] https://docs.python.org/dev/library/stdtypes.html#str
[272] https://docs.python.org/dev/library/stdtypes.html#str
[273] https://docs.python.org/dev/library/stdtypes.html#dict
[274] https://docs.python.org/dev/library/stdtypes.html#dict

getGlobalEnv(***kwargs*)

Macro API. Returns the global environment.

Returns a `dict`[275] containing the global environment

Return type `dict`[276]

getAllEnv(***kwargs*)

Macro API. Returns the enviroment for the macro.

Returns a `dict`[277] containing the environment for the macro

Return type `dict`[278]

getAllDoorEnv(***kwargs*)

Macro API. Returns the enviroment for the door where the macro is running.

Returns a `dict`[279] containing the environment

Return type `dict`[280]

setEnv(***kwargs*)

Macro API. Sets the environment key to the new value and stores it persistently.

Returns a `tuple`[281] with the key and value objects stored

Return type `tuple`[282]<`str`[283], object>

unsetEnv(***kwargs*)

Macro API. Unsets the given environment variable.

Parameters **key** (`str`[284]) – the environment variable name

reloadLibrary(***kwargs*)

Macro API. Reloads the given library(=module) names

Raises ImportError in case the reload process is not successfull

Parameters **lib_name** (`str`[285]) – library(=module) name

Returns the reloaded python module object

reloadMacro(***kwargs*)

Macro API. Reloads the module corresponding to the given macro name

Raises MacroServerExceptionList in case the macro is unknown or the reload process is not successfull

Parameters **macro_name** (`str`[286]) – macro name

reloadMacros(***kwargs*)

Macro API. Reloads the modules corresponding to the given macro names.

275 https://docs.python.org/dev/library/stdtypes.html#dict
276 https://docs.python.org/dev/library/stdtypes.html#dict
277 https://docs.python.org/dev/library/stdtypes.html#dict
278 https://docs.python.org/dev/library/stdtypes.html#dict
279 https://docs.python.org/dev/library/stdtypes.html#dict
280 https://docs.python.org/dev/library/stdtypes.html#dict
281 https://docs.python.org/dev/library/stdtypes.html#tuple
282 https://docs.python.org/dev/library/stdtypes.html#tuple
283 https://docs.python.org/dev/library/stdtypes.html#str
284 https://docs.python.org/dev/library/stdtypes.html#str
285 https://docs.python.org/dev/library/stdtypes.html#str
286 https://docs.python.org/dev/library/stdtypes.html#str

Raises MacroServerExceptionList in case the macro(s) are unknown or the reload process is not successfull

Parameters **macro_names** (sequence<`str`[287]>) – a list of macro names

reloadMacroLibrary(***kwargs*)

Macro API. Reloads the given library(=module) names

Raises MacroServerExceptionList in case the reload process is not successfull

Parameters **lib_name** (`str`[288]) – library(=module) name

Returns the `MacroLibrary` for the reloaded library

Return type `MacroLibrary`

reloadMacroLibraries(***kwargs*)

Macro API. Reloads the given library(=module) names

Raises MacroServerExceptionList in case the reload process is not successfull for at least one lib

param lib_names: a list of library(=module) names :type lib_name: seq<`str`[289]>

Returns a sequence of `MacroLibrary` objects for the reloaded libraries

Return type seq<`MacroLibrary`>

reloadMacroLib(***kwargs*)

Macro API. Reloads the given library(=module) names

Raises MacroServerExceptionList in case the reload process is not successfull

Parameters **lib_name** (`str`[290]) – library(=module) name

Returns the `MacroLibrary` for the reloaded library

Return type `MacroLibrary`

reloadMacroLibs(***kwargs*)

Macro API. Reloads the given library(=module) names

Raises MacroServerExceptionList in case the reload process is not successfull for at least one lib

param lib_names: a list of library(=module) names :type lib_name: seq<`str`[291]>

Returns a sequence of `MacroLibrary` objects for the reloaded libraries

Return type seq<`MacroLibrary`>

getViewOption(***kwargs*)

getViewOptions(***kwargs*)

setViewOption(***kwargs*)

resetViewOption(***kwargs*)

executor

Unofficial Macro API. Alternative to *`getExecutor()`* that does not throw StopException in case of a Stop. This should be called only internally

[287] https://docs.python.org/dev/library/stdtypes.html#str
[288] https://docs.python.org/dev/library/stdtypes.html#str
[289] https://docs.python.org/dev/library/stdtypes.html#str
[290] https://docs.python.org/dev/library/stdtypes.html#str
[291] https://docs.python.org/dev/library/stdtypes.html#str

door
: **Unofficial Macro API**. Alternative to `getDoorObj()` that does not throw StopException in case of a Stop. This should be called only internally

parent_macro
: **Unofficial Macro API**. Alternative to getParentMacro that does not throw StopException in case of a Stop. This should be called only internally by the *Executor*

description
: **Unofficial Macro API**. Alternative to `getDescription()` that does not throw StopException in case of a Stop. This should be called only internally by the *Executor*

isAborted()
: **Unofficial Macro API**.

isStopped()
: **Unofficial Macro API**.

isPaused()
: **Unofficial Macro API**.

classmethod hasResult()
: **Unofficial Macro API**. Returns True if the macro should return a result or False otherwise

  Returns True if the macro should return a result or False otherwise

  Return type bool[292]

getResult()
: **Unofficial Macro API**. Returns the macro result object (if any)

  Returns the macro result object or None

setResult(*result*)
: **Unofficial Macro API**. Sets the result of this macro

  Parameters **result** – (object) the result for this macro

exec_()
: **Internal method**. Execute macro as an iterator

stop()
: **Internal method**. Activates the stop flag on this macro.

abort()
: **Internal method**. Aborts the macro abruptly.

setProcessingStop(*yesno*)
: **Internal method**. Activates the processing stop flag on this macro

isProcessingStop()
: **Internal method**. Checks if this macro is processing stop

pause(*cb=None*)
: **Internal method**. Pauses the macro execution. To be called by the Door running the macro to pause the current macro

resume(*cb=None*)
: **Internal method**. Resumes the macro execution. To be called by the Door running the macro to resume the current macro

[292] https://docs.python.org/dev/library/functions.html#bool

iMacro class

class iMacro(**args, **kwargs*)

interactive = True

macro decorator

class macro(*param_def=None, result_def=None, env=None, hints=None, interactive=None*)

Class designed to decorate a python function to transform it into a macro. Examples:

```
@macro()
def my_macro1(self):
    self.output("Executing %s", self.getName())

@macro([ ["moveable", Type.Moveable, None, "motor to watch"] ])
def where_moveable(self, moveable):
    self.output("Moveable %s is at %s", moveable.getName(), moveable.
↪getPosition())
```

imacro decorator

imacro

Controller API reference

- *Controller* - Base API for all controller types
- *MotorController* - Motor controller API
- *CounterTimerController* - Counter/Timer controller API
- *ZeroDController* - 0D controller API
- *PseudoMotorController* - PseudoMotor controller API
- *PseudoCounterController* - PseudoCounter controller API
- *IORegisterController* - IORegister controller API

Data Type definition

When writing a new controller you may need to specify extra attributes (per controller or/and per axis) as well as extra properties. This chapter describes how to describe the data type for each of this additional members. Controller data type definition has the following equivalences. This means you can use any of the given possibilities to describe a field data type. The possibilities are ordered by preference (example: usage of `int`[293] is preferred to "int" or "PyTango.DevLong"):

- **for 0D data types:**

[293] https://docs.python.org/dev/library/functions.html#int

  - **integer**: `int`[294] | *`DataType.Integer`* | "int" | "integer" | "long" | `long` | ["PyTango."] "DevLong"
  - **double**: `float`[295] | *`DataType.Double`* | "double" | "float" | ["PyTango."] "DevDouble"
  - **string**: `str`[296] | *`DataType.String`* | "str" | "string" | ["PyTango."] "DevString"
  - **boolean**: `bool`[297] | *`DataType.Boolean`* | "bool" | "boolean" | ["PyTango."] "DevBoolean"

- **for 1D data types:**
  - **integer**: (`int`[298],) | (*`DataType.Integer`*,) | ("int",) | ("integer",) | (`long`,) | ("long",) | ["PyTango."] "DevVarLongArray" | (["PyTango."] "DevLong",)
  - **double**: (`float`[299],) | (*`DataType.Double`*,) | ("double",) | ("float",) | ["PyTango."] "DevVarDoubleArray" | (["PyTango."] "DevDouble",)
  - **string**: (`str`[300],) | (*`DataType.String`*,) | ("str",) | ("string",) | ["PyTango."] "DevVarStringArray" | (["PyTango."] "DevString",)
  - **boolean**: (`bool`[301],) | (*`DataType.Boolean`*,) | ("bool",) | ("boolean",) | ["PyTango."] "DevVarBooleanArray" | (["PyTango."] "DevBoolean",)

Deprecated since version 1.0: ["PyTango."] "Dev"<concrete type string> types are considered deprecated.

Note: when string, types are case insensitive. This means "long" is the same as "LONG"

Here is an example on how to define extra attributes per axis:

1. EncoderSource: a scalar r/w string
2. ReflectionMatrix: a 2D readable float with customized getter method

```python
from sardana import State, DataAccess
from sardana.pool.controller import MotorController, \
    Type, Description, DefaultValue, Access, FGet, FSet

class MyMotorCtrl(MotorController):

    axis_attributes = \
    {
        'EncoderSource' : { Type : str,
                            Description : 'motor encoder source', },

        'ReflectionMatrix' : { Type : ( (float,), ),
                               Access : DataAccess.ReadOnly,
                               FGet : 'getReflectionMatrix', },
    }

    def getAxisExtraPar(self, axis, name):
        name = name.lower()
```

(continues on next page)

294 https://docs.python.org/dev/library/functions.html#int
295 https://docs.python.org/dev/library/functions.html#float
296 https://docs.python.org/dev/library/stdtypes.html#str
297 https://docs.python.org/dev/library/functions.html#bool
298 https://docs.python.org/dev/library/functions.html#int
299 https://docs.python.org/dev/library/functions.html#float
300 https://docs.python.org/dev/library/stdtypes.html#str
301 https://docs.python.org/dev/library/functions.html#bool

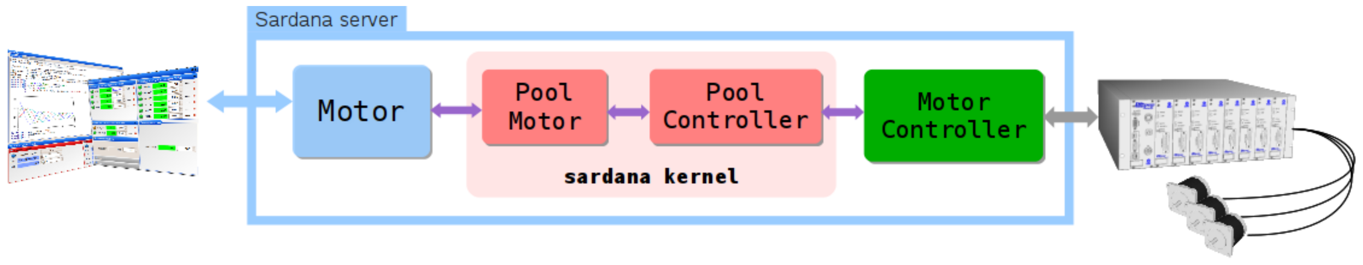
(continued from previous page)

```
        if name == 'encodersource':
            return self._encodersource[axis]

    def setAxisPar(self, axis, name, value):
        name = name.lower()
        if name == 'encodersource':
            self._encodersource[axis] = value

    def getReflectionMatrix(self, axis):
        return ( (1.0, 0.0), (0.0, 1.0) )
```

Motor API reference

The motor is one of the most used elements in sardana. A motor represents anything that can be *changed* (and can potentially take some time to do it).

This chapter explains the generic motor *API* in the context of sardana. In sardana there are, in fact, two Motor *API*s. To better explain why, let's consider the case were sardana server is running as a Sardana Tango device server:



Every motor in sardana is represented in the sardana kernel as a `PoolMotor`. The `PoolMotor` *API* is not directly accessible from outside the sardana server. This is a low level *API* that is only accessbile to someone writing a server extension to sardana. At the time of writing, the only available sardana server extension is Tango.

The second motor interface consists on the one provided by the server extension, which is in this case the one provided by the Tango motor device interface: `Motor`. The Tango motor interface tries to mimic the as closely as possible the `PoolMotor` *API*.

See also:

Motor overview the motor overview

`Motor` the motor tango device *API*

A motor will have, at least, a `state`, and a `position`. The state indicates at any time if the motor is stopped, in alarm or moving. The position, indicates the current *user position*. Unless a motor controller is specifically programmed not to, it's motors will also have:

limit switches the three limit switches (home, upper and lower). Each switch is represented by a boolean value: False means inactive while True means active.

low level `PoolMotor` API.

high level Tango Motor API: limit_switches tango attribute

acceleration motor acceleration (usually acceleration time in seconds, but it's up to the motor controller class to decide) `acceleration`

deceleration motor deceleration (usually deceleration time in seconds, but it's up to the motor controller class to decide)

`deceleration`

velocity top velocity

`velocity`

base rate initial velocity

`base_rate`

dial position the *dial position*

`dial_position`

offset the offset to be applied in the motor position computation [default: 0.0]

`offset`

sign the sign to be applied in the motor position computation [default: 1, possible values are (1, -1)]

`sign`

steps per unit This is the number of motor steps per *user position* [default: 1.0]

`step_per_unit`

backlash If this is defined to be something different than 0, the motor will always stop the motion coming from the same mechanical direction. This means that it could be possible to ask the motor to go a little bit after the desired position and then to return to the desired position. The value is the number of steps the motor will pass the desired position if it arrives from the "wrong" direction. This is a signed value. If the sign is positive, this means that the authorized direction to stop the motion is the increasing motor position direction. If the sign is negative, this means that the authorized direction to stop the motion is the decreasing motor position direction.

`backlash`

instability time This property defines the time in milliseconds that the software managing a motor movement will wait between it detects the end of the motion and the last motor position reading. It is typically used for motors that move mechanics which have an instability time after each motion.

`instability_time`

The available operations are:

start move absolute (*user position*) starts to move the motor to the given absolute user position

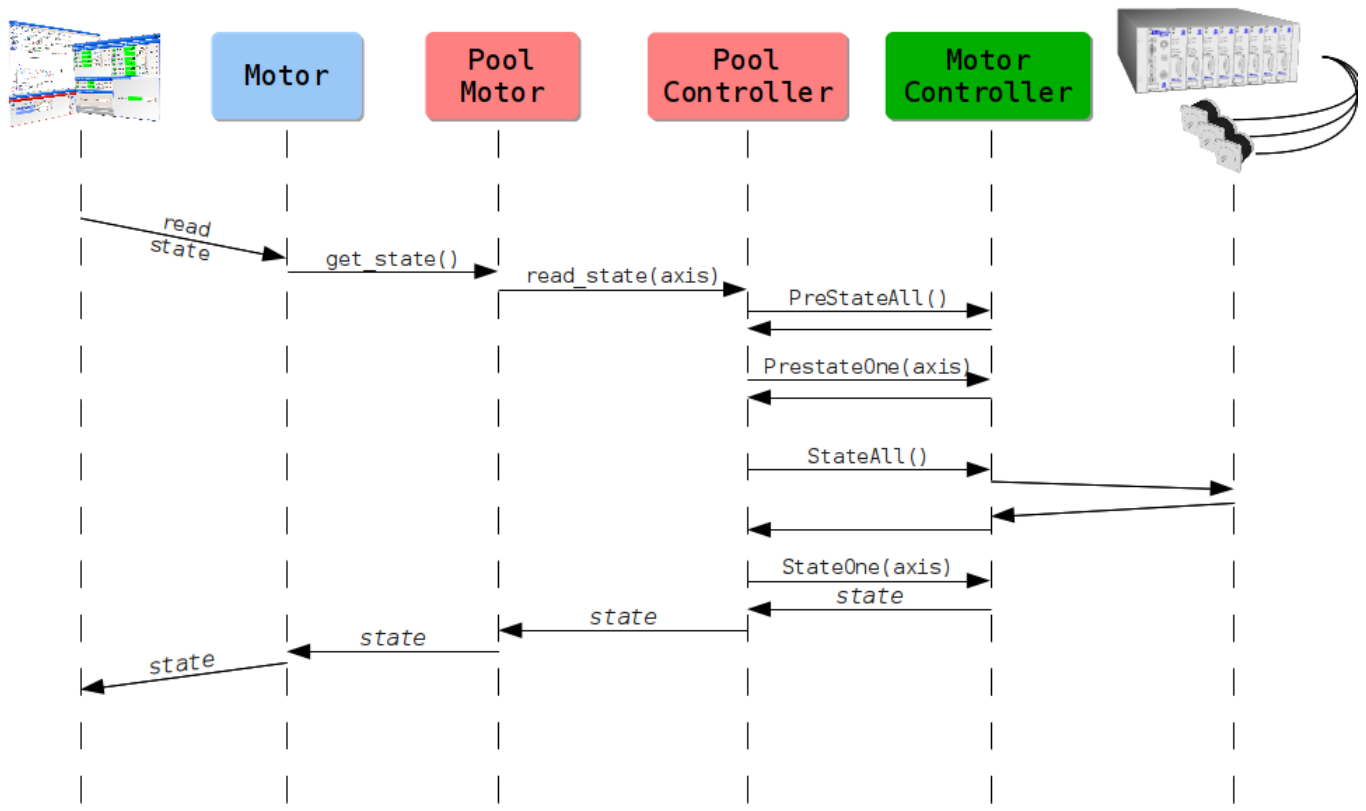
`start_move()`

stop stops the motor in an orderly fashion

abort stops the motor motion as fast as possible (possibly without deceleration time and loss of position)

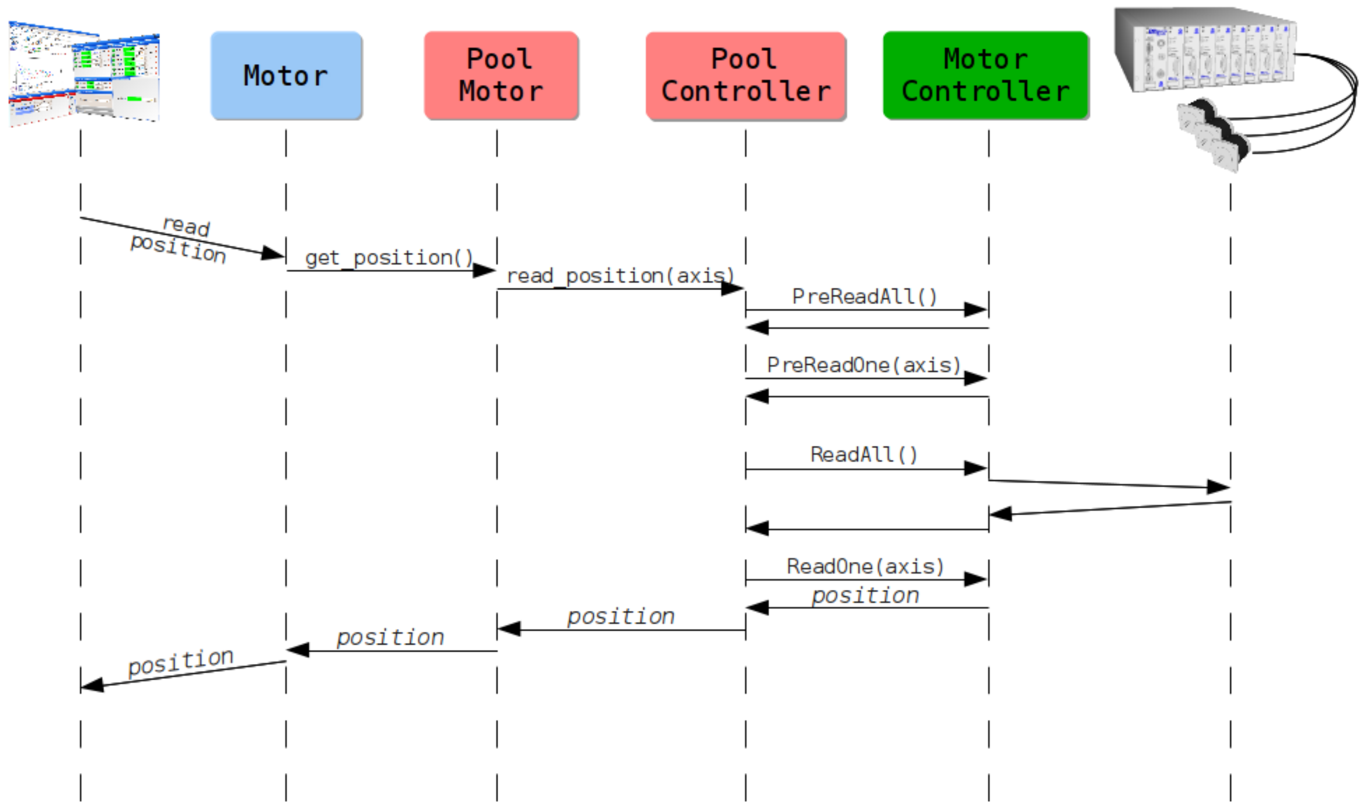
Motor state

On a sardana tango server, the motor state can be obtained by reading the state attribute or by executing the state command. The diagram shows the internal sequence of calls.



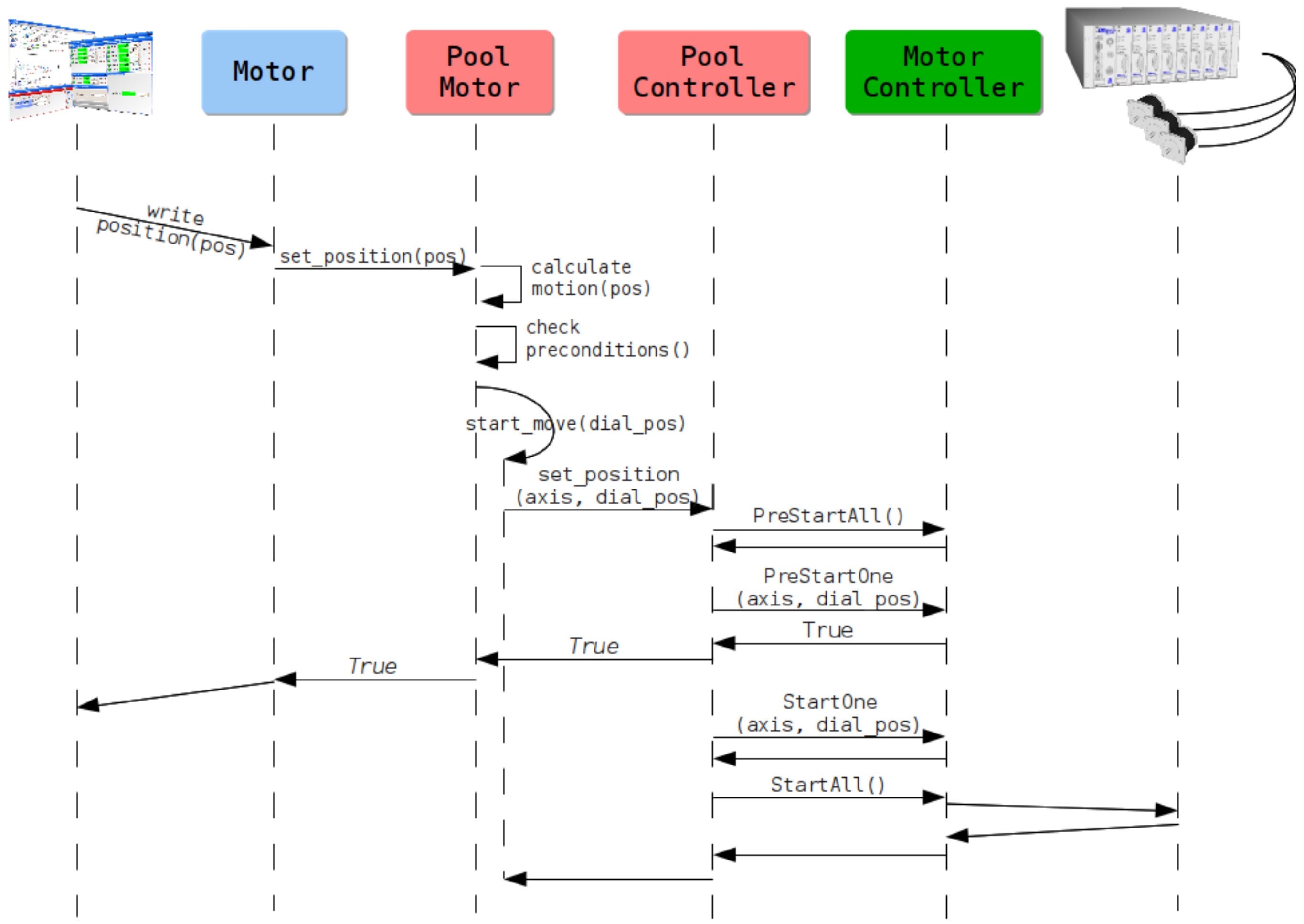
Motor position

The motor's current *user position* can be obtained by reading the position attribute. The diagram shows the internal sequence of calls.



Motion

The most useful thing to do with a motor is, of course, to move it. To move a motor to another absolute *user position* you have to write the value into the position attribute.



Before allowing a movement, some pre-conditions are automatically checked by tango (not represented in the diagram):

- motor is in a proper state;
- requested position is within the allowed motor boundaries (if defined)

Then, the *dial position* is calculated taking into account the *offset*, *signal* as well as a possible *backlash*.

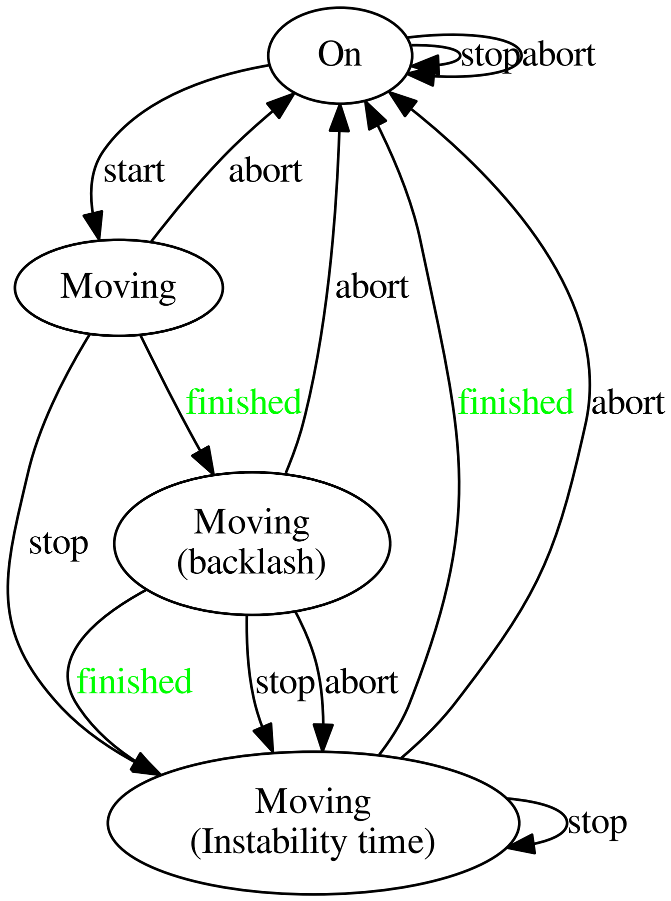
Afterward, and because the motor may be part of a pseudo motor system, other pre-conditions are checked:

- is the final *dial position* (including backlash) within the motor boundaries (if defined)
- will the resulting motion end in an allowed position for all the pseudo motors that depend on this motor

After all pre-conditions are checked, the motor will deploy a motion *job* into the sardana kernel engine which will trigger a series of calls to the underlying motor controller.

The motor awaits for the `PreStartOne()` to signal that the motion will be possible to return successfully from the move request.

The following diagram shows the motion state machine of a motor. The black state transitions are the ones which can be triggered by a *user*. For simplicity, only the most relevant states involved in a motor motion are shown. Error states are omitted.



I/O register API reference

The IOR is a generic element which allows to write/read from a given hardware register a value. This value type may be one of: `int`[302], `float`[303], `bool`[304].

An IOR has a `state`, and a `value` attributes. The state indicates at any time if the IOR is stopped, in alarm or moving. The value, indicates the current IOR value.

The available operations are:

write register(value) executes write operation on the IOR with the given value

`write_register()`

See also:

I/O register overview the I/O register overview

[302] https://docs.python.org/dev/library/functions.html#int
[303] https://docs.python.org/dev/library/functions.html#float
[304] https://docs.python.org/dev/library/functions.html#bool

`IORegister` the I/O register tango device *API*

Counter/Timer API reference

The counter/timer is one of the most used elements in sardana. A counter/timer represents an experimental channel which acquisition result is a scalar value.

A counter/timer has a `state`, and a `value` attributes. The state indicates at any time if the counter/timer is stopped, in alarm or moving. The value, indicates the current counter/timer value.

The available operations are:

start acquisition(integration time) starts to acquire the counter/timer with the given integration time

`start_acquisition()`

stop stops the counter/timer acquisition in an orderly fashion

abort stops the counter/timer acquisition as fast as possible

See also:

Counter/timer overview the counter/timer overview

`CTExpChannel` the counter/timer tango device *API*

0D channel API reference

The 0D experimental channel is used to access any kind of device which returns a scalar value and which are not counter/timer.

A 0D has a `state`, and a `value` attributes. The state indicates at any time if the 0D is stopped, in alarm or moving. The value behaves exactly the same as the accumulated value attribute.

The other attributes are:

accumulation Defines the computation type done on the values gathered during the acquisition. Three type of computation are supported:

- Sum - the accumulation value attribute is the sum of all the data read during the acquisition. This is the default type.
- Average - the accumulation value attribute is the average of all the data read during the acquisition.
- Integral - the accumulation value attribute is a type of the integral of all the data read during the acquisition.

current value This is the current a.k.a. instant value of the experimental channel. If the current value attribute is read while the acquisition is in progress, it returns the last updated by the acquisition operation value (cache). When there is no acquisition in progress the current value read executes the hardware readout and returns an updated value.

accumulated value This is the result of the data acquisition after the computation defined by the accumulation attribute has been applied. This value is 0 until an acquisition has been started. After an acquisition, the attribute value stays unchanged until the next acquisition is started.

accumulation buffer This buffer is filled with the instant values read by the acquisition operation.

time buffer This buffer is filled with the timestamps of the instant values present in the accumulation buffer and it is also filled during the acquisition operation.

The available operations are:

start acquisition(integration time) starts to acquire the 0D with the given integration time

    start_acquisition()

stop stops the 0D acquisition in an orderly fashion

abort stops the 0D acquisition as fast as possible

See also:

0D channel overview the 0D experiment channel overview

ZeroDExpChannel the 0D experiment channel tango device *API*

1D channel API reference

A 1D represents an experimental channel which acquisition result is a spectrum value.

A 1D has a `state`, and a `value` attributes. The state indicates at any time if the 1D is stopped, in alarm or moving. The value, indicates the current 1D value.

The other attributes are:

data source Unique identifier for the 1D data (value attribute)

The available operations are:

start acquisition(integration time) starts to acquire the 1D with the given integration time

    `start_acquisition()`

stop stops the 1D acquisition in an orderly fashion

abort stops the 1D acquisition as fast as possible

See also:

1D channel overview the 1D experiment channel overview

OneDExpChannel the 1D experiment channel tango device *API*

2D channel API reference

A 2D represents an experimental channel which acquisition result is a image value.

A 2D has a `state`, and a `value` attributes. The state indicates at any time if the 2D is stopped, in alarm or moving. The value, indicates the current 2D value.

The other attributes are:

data source Unique identifier for the 2D data (value attribute)

The available operations are:

start acquisition(integration time) starts to acquire the 2D with the given integration time

    `start_acquisition()`

stop stops the 2D acquisition in an orderly fashion

abort stops the 2D acquisition as fast as possible

See also:

2D channel overview the 2D experiment channel overview

TwoDExpChannel the 2D experiment channel tango device *API*

Trigger/Gate API reference

The trigger/gate element represents synchronization devices like for example the digital trigger and/or gate generators that are used to synchronize the experimental channels.

A trigger/gate has a `state`, and a `index` attributes. The state indicates at any time if the trigger/gate is stopped, in alarm or moving. The index, indicates the current trigger/gate index.

See also:

Trigger/gate overview the trigger/gate overview

TriggerGate the trigger/gate tango device *API*

Pseudo motor API reference

A pseudo motor has a `state`, and a `position` attributes. The state indicates at any time if the pseudo motor is stopped, in alarm or moving. The state is composed from the states of all the physical motors involved in the pseudo motor. So, if one of the motors is in moving or alarm state, the whole pseudo motor will be in that state. The position, indicates the current position.

The other pseudo motor's attributes are:

drift correction Flag to enable/disable drift correction while calculating physical motor(s) position(s). When enabled, the write sibling(s) position(s) will be used, when disabled, the read sibiling(s) position(s) will be used instead. By default drift correction is enabled.

`drift_correction`

siblings List of other psuedo motor objects that belongs to the same controller.

`siblings`

The available operations are:

start move absolute Starts to move the pseudo motor to the given absolute position.

`start_move()`

stop Stops the pseudo motor motion, by stopping all the physical motors, in an orderly fashion.

abort Stops the pseudo motor motion, by stopping all the physical motors, as fast as possible (possibly without deceleration time and loss of position).

See also:

Pseudo motor overview the pseudo-motor overview

PseudoMotor the pseudo-motor tango device *API*

Pseudo counter API reference

A pseudo counter has a `state`, and a `value` attributes. The state indicates at any time if the psuedo counter is stopped, in alarm or moving. The state is composed from the states of all the physical counters involved in the pseudo counter. So, if one of the counters is in moving or alarm state, the whole pseudo counter will be in that state. The value, indicates the current value.

The other pseudo counter's attributes are:

siblings List of other psuedo counter objects that belongs to the same controller.

siblings

The available operations are:

start acquisition(integration time) starts to acquire the pseudo counter with the given integration time

`start_acquisition()`

stop stops the pseudo counter acquisition in an orderly fashion

abort stops the pseudo counter acquisition as fast as possible

See also:

Pseudo counter overview the pseudo-counter overview

PseudoCounter the pseudo-counter tango device *API*

Measurement group API reference

The measurement group is a group element. It aggregates other elements like experimental channels (counter/timer, 0D, 1D and 2D or external attribute e.g. Tango[305]) and trigger/gates. The measurement group role is to execute acquisitions using the aggregated elements.

A measurement group has a `state` attribute. The state indicates at any time if the measurement group is stopped, in alarm or moving. The state is composed from the states of all the elements involved in the measurement group. So, if one of the involved element (experimental channel or trigger/gate) is in moving or alarm state, the whole measurement group will be in that state.

The other measurement group's attributes are:

timer The name of the channel used as a timer.

integration time Integration time to be used in the acquisition operation.

monitor count Monitor count to be used in the acquisition operation.

acquisition mode Acquisition mode to be used in the acquisition operation, either Timer or Monitor.

latency time Latency time between two consecutive acquisitions in the same acquisition operation.

synchronization Describes the acquisition operation synchronization. It is composed from the group(s) of equidistant acquisitions described by the following parameters:

- initial point
- initial delay
- total interval
- active interval
- number of repetitions

These parameters can be expressed in different synchronization domains if necessary (time and/or position).

[305] http://www.tango-controls.org

moveable Name of the master moveable.

> **Note:** This attribute has been included in Sardana on a provisional basis. Backwards incompatible changes (up to and including its removal) may occur if deemed necessary by the core developers.

The available operations are:

start acquisition() Starts to acquire the measurement group.

> *`start_acquisition()`*

See also:

Measurement group overview the measurement group overview

`MeasurementGroup` the measurement group tango device *API*

Device Pool Tango[306] API

> **Warning:** Device Pool chapter is out of date. Some parts of it are not valid and may create confusions e.g. "Specifying the motor controller features".

Todo: Update this chapter and distribute its contents logically around the documentation.

Introduction

This paper describes what could be the implementation of the Sardana device pool. This work is based on Jorg's paper called "Reordered SPEC[307]". It is **not at all** a final version of this device pool. It is rather a first approach to define this pool more precisely and to help defining its features and the way it could be implemented.

Overall pool design

The pool could be seen as a kind of intelligent Tango[308] device container to control the experiment hardware. In a first approach, it requires that the hardware to be controlled is connected to the control computer or to external crate(s) connected to the control computer using bus coupler. It has two basic features which are:

1. Hardware access using dynamically created/deleted Tango[309] devices according to the experiment needs
2. Management of some very common and well defined action regularly done on a beam line (scanning, motor position archiving. . . .)

To achieve these two goals and to provide the user with a way to control its behavior, it is implemented as a Tango[310] class with commands and attributes like any other Tango[311] class.

[306] http://www.tango-controls.org/
[307] http://www.certif.com/
[308] http://www.tango-controls.org/
[309] http://www.tango-controls.org/
[310] http://www.tango-controls.org/
[311] http://www.tango-controls.org/

Hardware access

Core hardware access

Most of the times, it is possible to define a list of very common devices found in most of the experiments, a list of communication link used between the experiment hardware and the control computer(s) and some of the most commonly used protocol used on these communication links. Devices commonly used to drive an experiment are:

- Motor
- Group of motor
- Pseudo motor
- Counter/Timer
- Multi Channel Analyzer
- CCD cameras
- And some other that I don't know

Communication link used to drive experiment devices are:

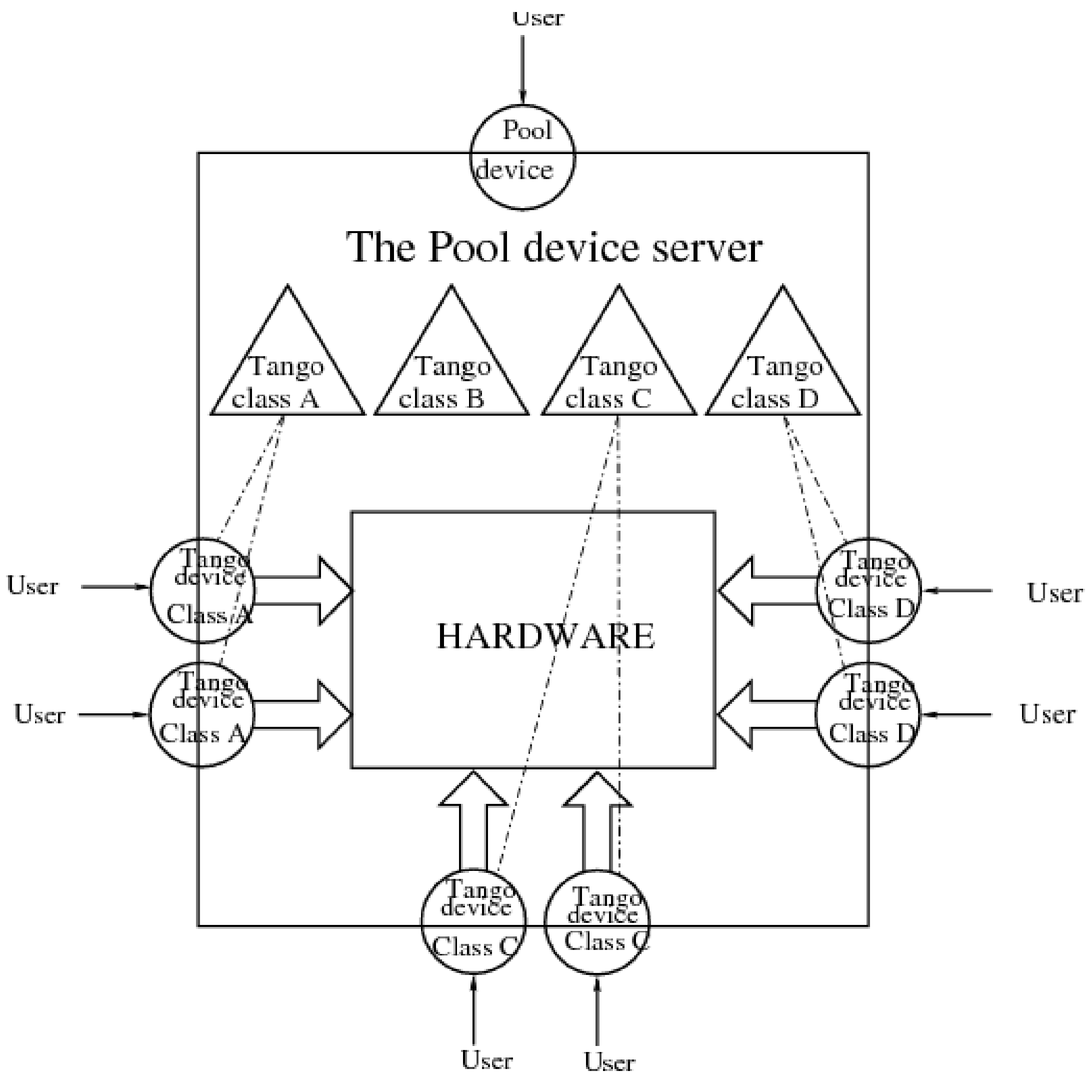
- Serial line
- GPIB
- Socket
- And some other that I don't know (USB????)

Protocol used on the communication links are:

- Modbus
- Ans some other that I don't know

Each of the controlled hardware (one motor, one pseudo-motor, one serial line device,. . .) will be driven by independent Tango[312] classes. The pool device server will embed all these Tango[313] classes together (statically linked). The pool Tango[314] device is the "container interface" and allows the user to create/delete classical Tango[315] devices which are instances of these embedded classes. This is summarized in the following drawing.

[312] http://www.tango-controls.org/
[313] http://www.tango-controls.org/
[314] http://www.tango-controls.org/
[315] http://www.tango-controls.org/



Therefore, the three main actions to control a new equipment using the pool will be (assuming the equipment is connected to the control computer via a serial line):

1. Create the serial line Tango[316] device with one of the Pool device command assigning it a name like "MyNewEquipment".
2. Connect to this newly created Tango[317] device using its assigned name
3. Send order or write/read data to/from the new equipment using for instance the WriteRead command of the serial line Tango[318] device

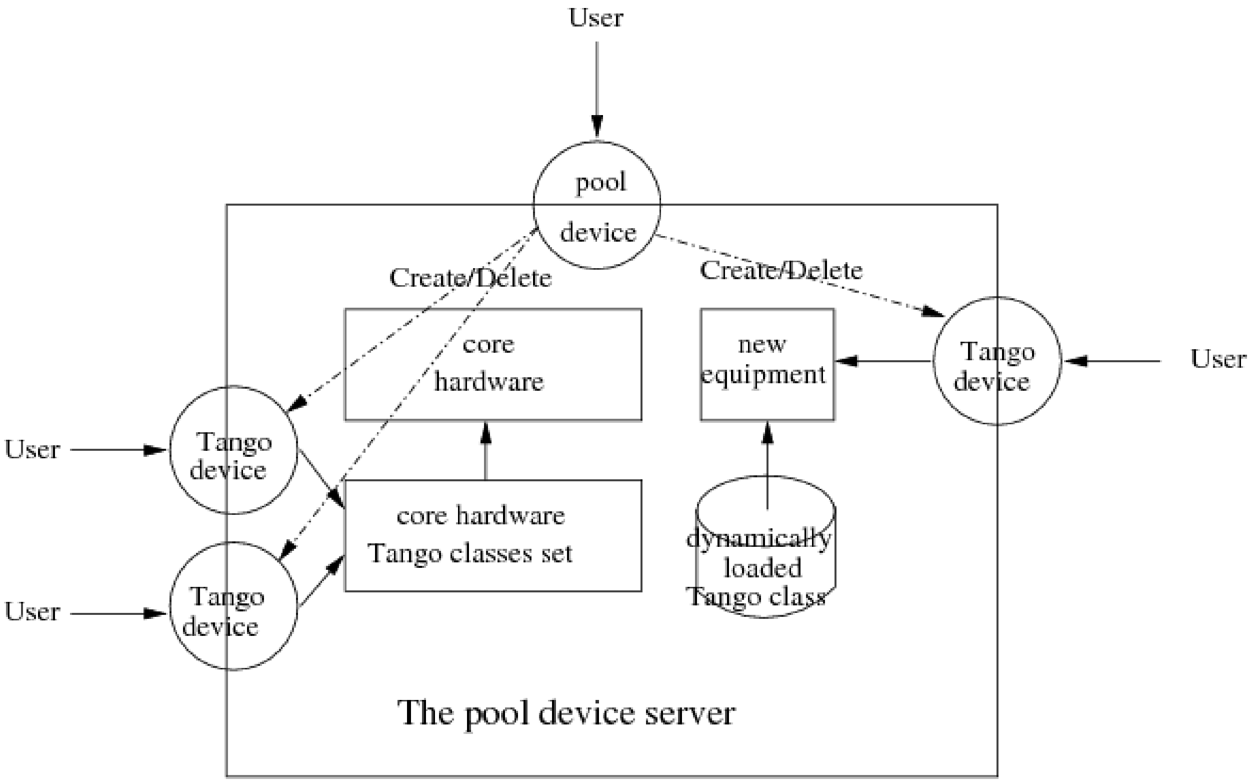
When the experiment does not need this new equipment any more, the user can delete the serial line Tango[319] device with another pool device command. Note that most of the time, creating Tango[320] device means defining some device configuration parameters (Property in Tango[321] language). The Tango[322] wizard will be used to retrieve which properties have to be defined and will allow the user to set them on the

316 http://www.tango-controls.org/
317 http://www.tango-controls.org/
318 http://www.tango-controls.org/
319 http://www.tango-controls.org/
320 http://www.tango-controls.org/
321 http://www.tango-controls.org/
322 http://www.tango-controls.org/

fly. This means that all the Tango[323] classes embedded within the Pool must have their wizard initialized.

Extending pool features

From time to time, it could be useful to extend the list of Tango[324] classes known by the device pool in case a new kind of equipment (not using the core hardware access) is added to the experiment. Starting with Tango[325] 5.5 (and the associated Pogo), each Tango[326] class has a method which allow the class to be dynamically loaded into a running process. This feature will be used to extend the pool feature. It has to be checked that it is possible for Tango[327] Python class.



To achieve this feature, the pool Tango[328] device will have commands to

- Load a Tango[329] class. This command will dynamically add two other commands and one attribute to the pool device Tango[330] interface. These commands and the attribute are:
  - Command: Create a device of the newly loaded class
  - Command: Delete a device of the newly loaded class
  - Attribute: Get the list of Tango[331] devices instances of the newly created class
- Unload a Tango[332] class

[323] http://www.tango-controls.org/
[324] http://www.tango-controls.org/
[325] http://www.tango-controls.org/
[326] http://www.tango-controls.org/
[327] http://www.tango-controls.org/
[328] http://www.tango-controls.org/
[329] http://www.tango-controls.org/
[330] http://www.tango-controls.org/
[331] http://www.tango-controls.org/
[332] http://www.tango-controls.org/

- Reload a Tango[333] class

Global actions

The following common actions regularly done on a beam line experiment will be done by the pool device server:

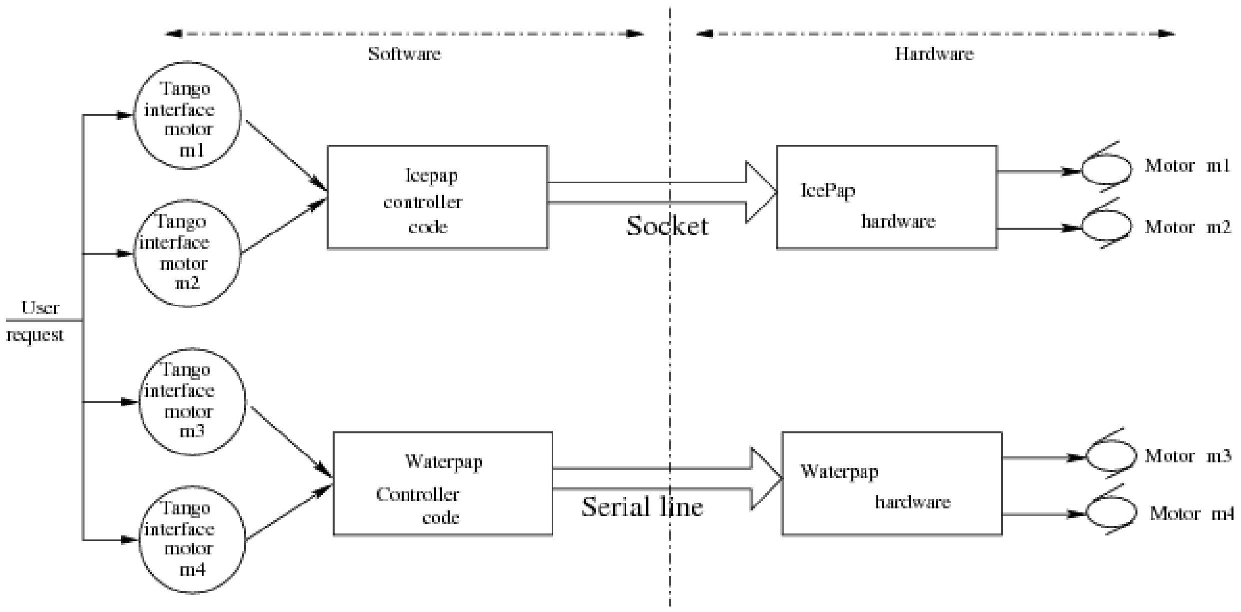
- Evaluating user constraint(s) before moving motor(s)
- Scanning
- Saving experiment data
- Experiment management
- Archiving motor positions

Sardana core hardware access

The Sardana Motor management

The user motor interface

The motor interface is a first approach of what could be a complete motor interface. It is statically linked with the Pool device server and supports several attributes and commands. It is implemented in C++ and used a set of the so-called "controller" methods. The motor interface is always the same whatever the hardware is. This is the rule of the "controller" to access the hardware using the communication link supported by the motor controller hardware (network link, serial line. . .).



The controller code has a well-defined interface and can be written using Python or C++. In both cases, it will be dynamically loaded into the pool device server process.

333 http://www.tango-controls.org/

The states

The motor interface knows five states which are ON, MOVING, ALARM, FAULT and UNKNOWN. A motor device is in MOVING state when it is moving! It is in ALARM state when it has reached one of the limit switches and is in FAULT if its controller software is not available (impossible to load it) or if a fault is reported from the hardware controller. The motor is in the UNKNOWN state if an exception occurs during the communication between the pool and the hardware controller. When the motor is in ALARM state, its status will indicate which limit switches is active.

The commands

The motor interface supports 3 commands on top of the Tango[334] classical Init, State and Status commands. These commands are summarized in the following table:

Command name	Input data type	Output data type
Abort	void	void
SetPosition	Tango::DevDouble	void
SaveConfig	void	void

- **Abort** : It aborts a running motion. This command does not have input or output argument.
- **SetPosition** : Loads a position into controller. It has one input argument which is the new position value (a double). It is allowed only in the ON or ALARM states. The unit used for the command input value is the physical unit: millimeters or milli-radians. It is always an absolute position.
- **SaveConfig** : Write some of the motor parameters in database. Today, it writes the motor acceleration, deceleration, base_rate and velocity into database as motor device properties. It is allowed only in the ON or ALARM states

The classical Tango[335] Init command destroys the motor and re-create it.

The attributes

The motor interface supports several attributes which are summarized in the following table:

Name	Data type	Data format	Writable	Memorized	Ope/Expert
Position	Tango::DevDouble	Scalar	R/W	No *	Ope
DialPosition	Tango::DevDouble	Scalar	R	No	Exp
Offset	Tango::DevDouble	Scalar	R/W	Yes	Exp
Acceleration	Tango::DevDouble	Scalar	R/W	No	Exp
Base_rate	Tango::DevDouble	Scalar	R/W	No	Exp
Deceleration	Tango::DevDouble	Scalar	R/W	No	Exp
Velocity	Tango::DevDouble	Scalar	R/W	No	Exp
Limit_Switches	Tango::DevBoolean	Spectrum	R	No	Exp
SimulationMode	Tango::DevBoolean	Scalar	R	No	Exp
Step_per_unit	Tango::DevDouble	Scalar	R/W	Yes	Exp
Backlash	Tango::DevLong	Scalar	R/W	Yes	Exp

- **Position** : This is read-write scalar double attribute. With the classical Tango min and max_value attribute properties, it is easy to define authorized limit for this attribute. See the definition of the

[334] http://www.tango-controls.org/
[335] http://www.tango-controls.org/

DialPosition and Offset attributes to get a precise definition of the meaning of this attribute. It is not allowed to read or write this attribute when the motor is in FAULT or UNKNOWN state. It is also not possible to write this attribute when the motor is already MOVING. **The unit used for this attribute is the physical unit: millimeters or milli-radian. It is always an absolute position.** The value of this attribute is memorized in the Tango[336] database but not by the default Tango[337] system memorization. See chapter XXX: Unknown inset LatexCommand ref{sub:Archiving-motor-position}: for details about motor position archiving.

- **DialPosition** : This attribute is the motor dial position. The following formula links together the Position, DialPosition, Sign and Offset attributes:

  ```
  Position = Sign * DialPosition + Offset
  ```

  This allows to have the motor position centered around any position defined by the Offset attribute (classically the X ray beam position). It is a read only attribute. To set the motor position, the user has to use the Position attribute. It is not allowed to read this attribute when the motor is in FAULT or UNKNOWN mode. The unit used for this attribute is the physical unit: millimeters or milli-radian. It is also always an **absolute** position.

- **Offset** : The offset to be applied in the motor position computation. By default set to 0. It is a memorized attribute. It is not allowed to read or write this attribute when the motor is in FAULT, MOVING or UNKNOWN mode.

- **Acceleration** : This is an expert read-write scalar double attribute. This parameter value is written in database when the SaveConfig command is executed. It is not allowed to read or write this attribute when the motor is in FAULT or UNKNOWN state.

- **Deceleration** : This is an expert read-write scalar double attribute. This parameter value is written in database when the SaveConfig command is executed. It is not allowed to read or write this attribute when the motor is in FAULT or UNKNOWN state.

- **Base_rate** : This is an expert read-write scalar double attribute. This parameter value is written in database when the SaveConfig command is executed. It is not allowed to read or write this attribute when the motor is in FAULT or UNKNOWN state.

- **Velocity** : This is an expert read-write scalar double attribute. This parameter value is written in database when the SaveConfig command is executed. It is not allowed to read or write this attribute when the motor is in FAULT or UNKNOWN state.

- **Limit_Switches** : Three limit switches are managed by this attribute. Each of the switch are represented by a boolean value: False means inactive while True means active. It is a read only attribute. It is not possible to read this attribute when the motor is in UNKNOWN mode. It is a spectrum attribute with 3 values which are:
  - Data[0] : The Home switch value
  - Data[1] : The Upper switch value
  - Data[2] : The Lower switch value

- **SimulationMode** : This is a read only scalar boolean attribute. When set, all motion requests are not forwarded to the software controller and then to the hardware. When set, the motor position is simulated and is immediately set to the value written by the user. To set this attribute, the user has to used the pool device Tango[338] interface. The value of the position, acceleration, deceleration, base_rate, velocity and offset attributes are memorized at the moment this attribute is set. When this mode is turned off, if the value of any of the previously memorized attributes has changed, it is

[336] http://www.tango-controls.org/
[337] http://www.tango-controls.org/
[338] http://www.tango-controls.org/

reapplied to the memorized value. It is not allowed to read this attribute when the motor is in FAULT or UNKNOWN states.

- **Step_per_unit** : This is the number of motor step per millimeter or per degree. It is a memorized attribute. It is not allowed to read or write this attribute when the motor is in FAULT or UNKNOWN mode. It is also not allowed to write this attribute when the motor is MOVING. The default value is 1.
- **Backlash** : If this attribute is defined to something different than 0, the motor will always stop the motion coming from the same mechanical direction. This means that it could be possible to ask the motor to go a little bit after the desired position and then to return to the desired position. The attribute value is the number of steps the motor will pass the desired position if it arrives from the "wrong" direction. This is a signed value. If the sign is positive, this means that the authorized direction to stop the motion is the increasing motor position direction. If the sign is negative, this means that the authorized direction to stop the motion is the decreasing motor position direction. It is a memorized attribute. It is not allowed to read or write this attribute when the motor is in FAULT or UNKNOWN mode. It is also not allowed to write this attribute when the motor is MOVING. Some hardware motor controllers are able to manage this backlash feature. If it is not the case, the motor interface will implement this behavior.

All the motor devices will have the already described attributes but some hardware motor controller supports other features which are not covered by this list of pre-defined attributes. Using Tango[339] dynamic attribute creation, a motor device may have extra attributes used to get/set the motor hardware controller specific features. The main characteristics of these extra attributes are :

- Name defined by the motor controller software (See next chapter)
- Data type is BOOLEAN, LONG, DOUBLE or STRING defined by the motor controller software (See next chapter)
- The data format is always Scalar
- The write type is READ or READ_WRITE defined by the motor controller software (See next chapter). If the write type is READ_WRITE, the attribute is memorized by the Tango[340] layer

The motor properties

Each motor device has a set of properties. Five of these properties are automatically managed by the pool software and must not be changed by the user. These properties are named Motor_id, _Acceleration, _Velocity, _Base_rate and _Deceleration. The user properties are:

Property name	Default value
Sleep_bef_last_read	0

This property defines the time in milli-second that the software managing a motor movement will wait between it detects the end of the motion and the last motor position reading.

Getting motor state and limit switches using event

The simplest way to know if a motor is moving is to survey its state. If the motor is moving, its state will be MOVING. When the motion is over, its state will be back to ON (or ALARM if a limit switch has been reached). The pool motor interface allows client interested by motor state or motor limit switches value to

[339] http://www.tango-controls.org/
[340] http://www.tango-controls.org/

use the Tango[341] event system subscribing to motor state change event. As soon as a motor starts a motion, its state is changed to MOVING and an event is sent. As soon as the motion is over, the motor state is updated ans another event is sent. In the same way, as soon as a change in the limit switches value is detected, a change event is sent to client(s) which have subscribed to change event on the Limit_Switches attribute.

Reading the motor position attribute

For each motor, the key attribute is its position. Special care has been taken on this attribute management. When the motor is not moving, reading the Position attribute will generate calls to the controller and therefore hardware access. When the motor is moving, its position is automatically read every 100 milli-seconds and stored in the Tango polling buffer. This means that a client reading motor Position attribute while the motor is moving will get the position from the Tango[342] polling buffer and will not generate extra controller calls. It is also possible to get a motor position using the Tango[343] event system. When the motor is moving, an event is sent to the registered clients when the change event criterion is true. By default, this change event criterion is set to be a difference in position of 5. It is tunable on a motor basis using the classical motor Position attribute abs_change property or at the pool device basis using its DefaultMotPos_AbsChange property. Anyway, not more than 10 events could be sent by second. Once the motion is over, the motor position is made unavailable from the Tango[344] polling buffer and is read a last time after a tunable waiting time (Sleep_bef_last_read property). A forced change event with this value is sent to clients using events.

The Motor Controller

XXX: Unknown inset LatexCommand label{sub:The-Motor-Controller}:

Each controller code is built as a shared library or as a Python module which is dynamically loaded by the pool device the first time one controller using the shared library (or the module) is created. Each controller is uniquely defined by its name following the syntax:

```
<controller_file_name>.<controller_class_name>/<instance_name>
```

At controller creation time, the pool checks the controller unicity on its control system (defined by the TANGO_HOST). It is possible to write controller using either C++ or Python language. Even if a Tango device server is a multi-threaded process, every access to the same controller will be serialized by a monitor managed by the Motor interface. This monitor is attached to the controller class and not to the controller instance to handle cases where several instances of the same controller class is used. For Python controller, this monitor will also take care of taking/releasing the Python Global Interpreter Lock (GIL) before any call to the Python controller is executed.

The basic

For motor controller, a pre-defined set of methods has to be implemented in the class implementing the controller interface. These methods can be splitted in 6 different types which are:

1. Methods to create/remove motor
2. Methods to move motor(s)
3. Methods to read motor(s) position

[341] http://www.tango-controls.org/
[342] http://www.tango-controls.org/
[343] http://www.tango-controls.org/
[344] http://www.tango-controls.org/

4. Methods to get motor(s) state
5. Methods to configure a motor
6. Remaining methods.

These methods, their rules and their execution sequencing is detailed in the following sub-chapters. The motor controller software layer is also used to inform the upper level of the features supported by the underlying hardware. This is called the controller **features** . It is detailed in a following sub-chapter. Some controller may need some configuration data. This will be supported using Tango properties. This is detailed in a dedicated sub-chapter.

Specifying the motor controller features

A controller feature is something that motor hardware controller is able to do or require on top of what has been qualified as the basic rules. Even if these features are common, not all the controllers implement them. Each of these common features are referenced by a pre- defined string. The controller code writer defined (from a pre-defined list) which of these features his hardware controller implements/requires. This list (a Python list or an array of C strings) has a well-defined name used by the upper layer software to retrieve it. The possible strings in this list are (case independent):

- **CanDoBacklash** : The hardware controller manages the motor backlash if the user defines one
- **WantRounding** : The hardware controller wants an integer number of step
- **encoder** : The hardware knows how to deal with encoder
- **home** : The hardware is able to manage home switch
- **home_acceleration** : It is possible to set the acceleration for motor homing
- **home_method _ xxx** : The hardware knows the home method called xxx
- **home_method_yyy** : The hardware knows the home method called yyy

The name of this list is simply: **ctrl_features** . If this list is not defined, this means that the hardware does not support/require any of the additional features. The Tango[345] motor class will retrieve this list from the controller before the first motor belonging to this controller is created. As an example, we suppose that we have a pool with two classes of motor controller called Ctrl_A and Ctrl_B. The controllers features list are (in Python)

```
Controller A : ctrl_features = ['CanDoBacklash','encoder']
ControllerB : ctrl_features = ['WantRounding','home','home_method_xxx']
```

All motors devices belonging to the controller A will have the Encoder and Backlash features. For these motors, the backlash will be done by the motor controller hardware. All the motors belonging to the controller B will have the rounding, home and home_method features. For these motors, the backlash will be done by the motor interface code.

Specifying the motor controller extra attributes

XXX: Unknown inset LatexCommand label{par:Specifying-the-motor}:

Some of the hardware motor controller will have features not defined in the features list or not accessible with a pre-defined feature. To provide an interface to these specific hardware features, the controller code can define extra attributes. Another list called : **ctrl_extra_attributes** is used to define them. This list (Python dictionary or an array of classical C strings) is used to define the name, data and read-write type

[345] http://www.tango-controls.org/

of the Tango[346] attribute which will be created to deal with these extra features. The attribute created for these controller extra features are all:

- Boolean, Long, Double or String
- Scalar
- Read or Read/Write (and memorized if Read/Write).

For Python classes (Python controller class), it is possible to define these extra attributes informations using a Python dictionary called **ctrl_extra _ attributes** . The extra attribute name is the dictionary element key. The dictionary element value is another dictionary with two members which are the extra attribute data type and the extra attribute read/write type. For instance, for our IcePap controller, this dictionary to defined one extra attribute called "SuperExtra" of data type Double which is also R/W will be:

```
ctrl_extra_attributes = { "SuperExtra" : { "Type" : "DevDouble", "R/W Type", "READ_
↪WRITE" } }
```

For C++ controller class, the extra attributes are defined within an array of **Controller::ExtraAttrInfo** structures. The name of this array has to be <Ctrl_class_name>_ctrl_extra_attributes. Each Controller::ExtraAttrInfo structure has three elements which are all pointers to classical C string (const char *). These elements are:

1. The extra attribute name
2. The extra attribute data type
3. The extra attribute R/W type

A NULL pointer defined the last extra attribute. The following is an example of extra attribute definition for a controller class called "DummyController":

```
Controller::ExtraAttrInfo DummyController_ctrl_extra_attributes[] =
    { { "SuperExtra", "DevDouble", "Read_Write" }, NULL };
```

The string describing the extra attribute data type may have the following value (case independent):

- DevBoolean, DevLong, DevDouble or DevString (in Python, a preceding "PyTango." is allowed)

The string describing the extra attribute R/W type may have the following value (case independent)

- Read or Read_Write (in Python, a preceding "PyTango." is allowed)

Methods to create/remove motor from controller

Two methods are called when creating or removing motor from a controller. These methods are called **AddDevice** and **DeleteDevice** . The AddDevice method is called when a new motor belonging to the controller is created within the pool. The DeleteDevice method is called when a motor belonging to the controller is removed from the pool.

Methods to move motor(s)

Four methods are used when a request to move motor(s) is executed. These methods are called **PreStartAll** , **PreStartOne** , **StartOne** and **StartAll** . The algorithm used to move one or several motors is the following:

[346] http://www.tango-controls.org/

```
/FOR/ Each controller(s) implied in the motion
     - Call PreStartAll()
/END FOR/

/FOR/ Each motor(s) implied in the motion
     - ret = PreStartOne(motor to move, new position)
     - /IF/ ret is true
          - Call StartOne(motor to move, new position)
     - /END IF/
/END FOR/

/FOR/ Each controller(s) implied in the motion
     - Call StartAll()
/END FOR/
```

The following array summarizes the rule of each of these methods:

Default action	Does nothing	Return true	Does nothing	Does nothing
Externally called by	Writing the Position attribute	Writing the Position attribute	Writing the Position attribute	Writing the Position attribute
Internally called	Once for each implied controller	For each implied motor	For each implied motor	Once for each implied controller
Typical rule	Init internal data for motion	Check if motor motion is possible	Set new motor position in internal data	Send order to physical controller

This algorithm covers the sophisticated case where a physical controller is able to move several motors at the same time. For some simpler controller, it is possible to implement only the StartOne() method. The default implementation of the three remaining methods is defined in a way that the algorithm works even in such a case.

Methods to read motor(s) position

Four methods are used when a request to read motor(s) position is received. These methods are called PreReadAll, PreReadOne, ReadAll and ReadOne. The algorithm used to read position of one or several motors is the following:

```
/FOR/ Each controller(s) implied in the reading
     - Call PreReadAll()
/END FOR/

/FOR/ Each motor(s) implied in the reading
     - PreReadOne(motor to read)
/END FOR/

/FOR/ Each controller(s) implied in the reading
     - Call ReadAll()
/END FOR/

/FOR/ Each motor(s) implied in the reading
     - Call ReadOne(motor to read)
/END FOR/
```

The following array summarizes the rule of each of these methods:

Default action	Does nothing	Does nothing	Does nothing	Print message on the screen and returns NaN. Mandatory for Python
Externally called by	Reading the Position attribute	Reading the Position attribute	Reading the Position attribute	Reading the Position attribute
Internally called	Once for each implied controller	For each implied motor	For each implied controller	Once for each implied motor
Typical rule	Init internal data for reading	Memorize which motor has to be read	Send order to physical controller	Return motor position from internal data

This algorithm covers the sophisticated case where a physical controller is able to read several motors positions at the same time. For some simpler controller, it is possible to implement only the ReadOne() method. The default implementation of the three remaining methods is defined in a way that the algorithm works even in such a case.

Methods to get motor(s) state

XXX: Unknown inset LatexCommand label{par:Methods-to-get-state}:

Four methods are used when a request to get motor(s) state is received. These methods are called PreStateAll, PreStateOne, StateAll and StateOne. The algorithm used to get state of one or several motors is the following :

```
/FOR/ Each controller(s) implied in the state getting
     - Call PreStateAll()
/END FOR/

/FOR/ Each motor(s) implied in the state getting
     - PreStateOne(motor to get state)
/END FOR/

/FOR/ Each controller(s) implied in the state getting
     - Call StateAll()
/END FOR/

/FOR/ Each motor(s) implied in the getting state
     - Call StateOne(motor to get state)
/END FOR/
```

The following array summarizes the rule of each of these methods:

Default action	Does nothing	Does nothing	Does nothing	Mandatory for Python
Externally called by	Reading the motor state	Reading the motor state	Reading the motor state	Reading the motor state
Internally called	Once for each implied controller	For each implied motor	For each implied controller	Once for each implied motor
Typical rule	Init internal data for reading	Memorize which motor has to be read	Send order to physical controller	Return motor state from internal data

This algorithm covers the sophisticated case where a physical controller is able to read several motors state at the same time. For some simpler controller, it is possible to implement only the StateOne() method. The default implementation of the three remaining methods is defined in a way that the algorithm works even in such a case.

Methods to configure a motor

The rule of these methods is to

- Get or Set motor parameter(s) with methods called GetPar() or SetPar()
- Get or Set motor extra feature(s) parameter with methods called GetExtraAttributePar() or SetExtraAttributePar()

The following table summarizes the usage of these methods:

Called by	Reading the Velocity, Acceleration, Base_rate, Deceleration and eventually Backlash attributes	Writing the Velocity, Acceleration, Base_rate, Deceleration, Step_per_unit and eventually Backlash attribute	Reading any of the extra attributes	Writing any of the extra attributes
Rule	Get parameter from physical controller	Set parameter in physical controller	Get extra attribute value from the physical layer	Set additional attribute value in physical controller

Please, note that the default implementation of the GetPar() prints a message on the screen and returns a NaN double value. The GetExtraAttributePar() default implementation also prints a message on the screen and returns a string set to “Pool_met_not_implemented”.

The remaining methods

The rule of the remaining methods are to

- Load a new motor position in a controller with a method called DefinePosition()
- Abort a running motion with a method called AbortOne()
- Send a raw string to the controller with a method called SendToCtrl()

The following table summarizes the usage of these methods:

Called by	The motor SetPosition command	The motor Abort command	The Pool SendToController command
Rule	Load a new motor position in controller	Abort a running motion	Send the input string to the controller and returns the controller answer

Controller properties

XXX: Unknown inset LatexCommand label{par:Controller-properties}:

Each controller may have a set of **properties** to configure itself. Properties are defined at the controller class level but can be re-defined at the instance level. It is also possible to define a property default value. These default values are stored within the controller class code. If a default value is not adapted to specific object instance, it is possible to define a new property value which will be stored in the Tango[347] database. Tango[348] database allows storing data which are not Tango[349] device property. This storage could be seen simply as a couple name/value. Naming convention for this kind of storage could be defined as:

controller_class->prop: value or controller_class/instance->prop: value

The calls necessary to retrieve/insert/update these values from/to the database already exist in the Tango[350] core. The algorithm used to retrieve a property value is the following:

```
- Property value = Not defined

/IF/ Property has a default value
    - Property value = default value
/ENDIF/

/IF/ Property has a value defined in db at class level
    - Property value = class db value
/ENDIF/

/IF/ Property has a value defined in db at instance level
    - Property value = instance db value
/ENDIF/

/IF/ Property still not defined
    - Error
/ENDIF/
```

As an example, the following array summarizes the result of this algorithm. The example is for an IcePap controller and the property is the port number (called port_number):

default value	5000	5000	5000	5000	
class in DB			5150	5150	
inst. in DB		5200		5250	
Property value	5000	5200	5150	5250	Error

- Case 1: The IcePap controller class defines one property called port_number and assigns it a default value of 5000

[347] http://www.tango-controls.org/
[348] http://www.tango-controls.org/
[349] http://www.tango-controls.org/
[350] http://www.tango-controls.org/

- Case 2 : An IcePap controller is created with an instance name "My_IcePap". The property IcePap/My_IcePap->port_number has been set to 5200 in db
- Case 3: The hard coded value of 5000 for port number does not fulfill the need. A property called IcePap->port_number set to 5150 is defined in db.
- Case 4: We have one instance of IcePap called "My_IcePap" for which we have defined a property "IcePap/My_IcePap" set to 5250.
- Case 5: The IcePap controller has not defined a default value for the property.

In order to provide the user with a friendly interface, all the properties defined for a controller class have to have informations hard-coded into the controller class code. We need at least three informations and sometimes four for each property. They are:

1. The property name (Mandatory)
2. The property description (Mandatory)
3. The property data type (Mandatory)
4. The property default value (Optional)

With these informations, a graphical user interface is able to build at controller creation time a panel with the list of all the needed properties, their descriptions and eventually their default value. The user then have the possibility to re-define property value if the default one is not valid for his usage. This is the rule of the graphical panel to store the new value into the Tango[351] database. The supported data type for controller property are:

Property data type	String to use in property definition
Boolean	DevBoolean
Long	DevLong
Double	DevDouble
String	DevString
Boolean array	DevVarBooleanArray
Long array	DevVarLongArray
Double array	DevVarDoubleArray
String array	DevVarStringArray

For Python classes (Python controller class), it is possible to define these properties informations using a Python dictionary called **class_prop** . The property name is the dictionary element key. The dictionary element value is another dictionary with two or three members which are the property data type, the property description and an optional default value. If the data type is an array, the default value has to be defined in a Python list or tuple. For instance, for our IcePap port number property, this dictionary will be

```
class_prop = { "port_number" : { "Type" : "DevLong", "Description",
    "Port on which the IcePap software server is listening", "DefaultValue" : 5000 } }
```

For C++ controller class, the properties are defined within an array of **Controller::PropInfo** structures. The name of this array has to be <Ctrl_class_name>_class_prop. Each Controller::PropInfo structure has four elements which are all pointers to classical C string (const char *). These elements are:

1. The property name
2. The property description
3. The property data type

[351] http://www.tango-controls.org/

4. The property default value (NULL if not used)

A NULL pointer defined the last property. The following is an example of property definition for a controller class called "DummyController":

```
Controller::PropInfo DummyController_class_prop[] =
{{"The prop","The first CPP property","DevLong","12"},
 {"Another_Prop","The second CPP property","DevString",NULL},
 {"Third_Prop","The third CPP property","DevVarLongArray","11,22,33"},
 NULL};
```

The value of these properties is passed to the controller at controller instance creation time using a constructor parameter. In Python, this parameter is a dictionnary and the base class of the controller class will create one object attribute for each property. In our Python example, the controller will have an attribute called "port_number" with its value set to 5000. In C++, the controller contructor receives a vector of **Controller::Properties** structure. Each Controller::Properties structure has two elements which are:

1. The property name as a C++ string
2. **The property value in a PropData structure. This PropData structure has four elements which are**

    (a) A C++ vector of C++ bool type

    (b) A C++ vector of C++ long type

    (c) A C++ vector of C++ double type

    (d) A C++ vector of C++ string.

Only the vector corresponding to the property data type has a size different than 0. If the property is an array, the vector has as many elements as the property has.

The MaxDevice property

Each controller has to have a property defining the maximum number of device it supports. This is a mandatory requirement. Therefore, in Python this property is simply defined by setting the value of a controller data member called **MaxDevice** which will be taken as the default value for the controller. In C++, you have to define a global variable called <Ctrl_class_name>_MaxDevice. The management of the number of devices created using a controller (limited by this property) will be completely done by the pool software. The information related to this property is automatically added as first element in the information passed to the controller at creation time. The following is an example of the definition of this MaxDevice property in C++ for a controller class called "DummyController"

```
long DummyController_MaxDevice = 16;
```

C++ controller

For C++, the controller code is implemented as a set of classes: A base class called **Controller** and a class called **MotorController** which inherits from Controller. Finally, the user has to write its controller class which inherits from MotorController.

XXX: Unknown layout Subparagraph: The Controller class XXX: XXX: Unknown inset LatexCommand label{sub:The-Cpp-Controller-class}: This class defined two pure virtual methods, seven virtual methods and some data types. The methods defined in this class are:

1. void **Controller::AddDevice** (long axe_number) Pure virtual

2. void **Controller::DeleteDevice** (long axe_number) Pure virtual
3. void **Controller::PreStateAll** () The default implementation does nothing
4. void **Controller::PreStateOne** (long idx_number) The default implementation does nothing. The parameter is the device index in the controller
5. void **Controller::StateAll** () The default implementation does nothing
6. void **Controller::StateOne** (long idx_number,CtrlState *ptr) Read a device state. The CtrlState data type is a structure with two elements which are:
   - A long dedicated to return device state (format ??)
   - A string used in case the motor is in FAULT and the controller is able to return a string describing the fault.
7. string **Controller::SendToCtrl** (string in_string) Send the input string to the controller without interpreting it and returns the controller answer
8. Controller::CtrlData **Controller::GetExtraAttributePar** (long idx_number,string &extra_attribute_name) Get device extra attribute value. The name of the extra attribute is passed as the second argument of the method. The default definition of this method prints a message on the screen and returns a string set to "Pool_meth_not_implemented". The CtrlData data type is a structure with the following elements
   (a) A data type enumeration called data_type describing which of the following element is valid (BOOLEAN, LONG, DOUBLE or STRING)
   (b) A boolean data called bo_data for boolean transfer
   (c) A long data called lo_data for long transfer
   (d) A double data called db_data for double transfer
   (e) A C++ string data called str_data for string transfer
9. void **Controller::SetExtraAttributePar** (long idx_number, string &extra_attribute_name, Controller::CtrlData &extra_attribute_value) Set device extra attribute value.

It also has one data member which is the controller instance name with one method to return it

1. string & **Controller::get_name** (): Returns the controller instance name

XXX: Unknown layout Subparagraph: The MotorController class This class defined twelve virtual methods with default implementation. The virtual methods declared in this class are:

1. void **MotorController::PreStartAll** () The default implementation does nothing.
2. bool **MotorController::PreStartOne** (long axe_number, double wanted_position) The default implementation returns True.
3. void **MotorController::StartOne** (long axe_number, double wanted_position) The default implementation does nothing.
4. void **MotorController::StartAll** () Start the motion. The default implementation does nothing.
5. void **MotorController::PreReadAll** () The default implementation does nothing.
6. void **MotorController::PreReadOne** (long axe_number) The default implementation does nothing.
7. void **MotorController::ReadAll** () The default implementation does nothing.
8. double **MotorController::ReadOne** (long axe_number) Read a position. The default implementation does nothing.

9. void **MotorController::AbortOne** (long axe_number) Abort a motion. The default implementation does nothing.
10. void **MotorController::DefinePosition** (long axe_number, double new_position) Load a new position. The default implementation does nothing.
11. Controller::CtrlData **MotorController::GetPar** (long axe_number, string &par_name) Get motor parameter value. The CtrlData data type is a structure with the following elements

    (a) A data type enumeration called data_type describing which of the following element is valid (BOOLEAN, LONG, DOUBLE or STRING)

    (b) A boolean data called bo_data for boolean transfer

    (c) A long data called lo_data for long transfer

    (d) A double data called db_data for double transfer

    (e) A C++ string data called str_data for string transfer

    A motor controller has to handle four or five different possible values for the "par_name" parameter which are:

    - Acceleration
    - Deceleration
    - Velocity
    - Base_rate
    - Backlash which has to be handled only for controller which has the backlash feature

    The default definition of this method prints a message on the screen and returns a NaN double value.
12. void **MotorController::SetPar** (long axe_number, string &par_name, Controller::CtrlData &par_value) Set motor parameter value. The default implementation does nothing. A motor controller has to handle five or six different value for the "par_name" parameter which are:

    - Acceleration
    - Deceleration
    - Velocity
    - Base_rate
    - Step_per_unit
    - Backlash which has to be handled only for controller which has the backlash feature

    The description of the CtrlData type is given in the documentation of the GetPar() method. The default definition of this method does nothing

This class has only one constructor which is

1. **MotorController::MotorController** (const char *) Constructor of the MotorController class with the controller name as instance name

Please, note that this class defines a structure called MotorState which inherits from the Controller::CtrlState and which has a data member:

1. A long describing the motor limit switches state (bit 0 for the Home switch, bit 1 for Upper Limit switch and bit 2 for the Lower Limit switch)

This structure is used in the StateOne() method.

XXX: Unknown layout Subparagraph: The user controller class XXX: XXX: Unknown inset LatexCommand label{par:The-user-controller}: The user has to implement the remaining pure virtual methods (AddDevice and DeleteDevice) and has to re-define virtual methods if the default implementation does not cover his needs. The controller code has to define two global variables which are:

1. **Motor_Ctrl_class_name** (for Motor controller). This is an array of classical C strings terminated by a NULL pointer. Each array element is the name of a Motor controller class defined in this file.
2. **<CtrlClassName>_MaxDevice** . This variable is a long defining the maximum number of device that the controller hardware can support.

On top of that, a controller code has to define a C function (defined as "extern C") which is called by the pool to create instance(s) of the controller class. This function has the following definition:

```
Controller * **_create_<Controller class name>** (const char \*ctrl_instance_name,
↪vector<Controller::Properties> &props)
```

For instance, for a controller class called DummyController, the name of this function has to be: _create_DummyController(). The parameters passed to this function are:

1. The forth parameter given to the pool during the CreateController command (the instance name).
2. A reference to a C++ vector with controller properties as defined in XXX: Unknown inset LatexCommand ref{par:Controller-properties}:

The rule of this C function is to create one instance of the user controller class passing it the arguments it has received. The following is an example of these definitions

```
//
// Methods of the DummyController controller
//
....

const char *Motor_Ctrl_class_name[] = {"DummyController",NULL};

long DummyController_MaxDevice = 16;

extern "C" {
Controller *_create_DummyController(const char *inst,vector<Controller::Properties> &
↪prop)
{
   return new DummyController(inst,prop);
}
}
```

On top of these mandatory definitions, you can define a controller documentation string, controller properties, controller features and controller extra features. The documentation string is the first element of the array returned by the Pool device GetControllerInfo command as detailed in XXX: Unknown inset LatexCommand ref{ite:GetControllerInfo:}: . It has to be defined as a classical C string (const char *) with a name like <Ctrl_class_name>_doc. The following is an example of a controller C++ code defining all these elements.

```
//
// Methods of the DummyController controller
//
....
```

(continues on next page)

(continued from previous page)

```
const char *Motor_Ctrl_class_name[] = {"DummyController",NULL};
const char *DummyController_doc = "This is the C++ controller for the DummyController␣
↪class";

long DummyController_MaxDevice = 16;

char *DummyController_ctrl_extra_features_list[] = {{"Extra_1","DevLong","Read_Write"}
↪,
                                                    {"Super_2","DevString","Read"},
                                                    NULL};
char *DummyController_ctrl_features[] = {"WantRounding","CanDoBacklash",NULL};

Controller::PropInfo DummyController_class_prop[] =
{{"The prop","The first CPP property","DevLong","12"},
 {"Another_Prop","The second CPP property","DevString",NULL},
 {"Third_Prop","The third CPP property","DevVarLongArray","11,22,33"},
 NULL};

extern "C" {
Controller *_create_DummyController(const char *inst,vector<Controller::Properties> &
↪prop)
{
   return new DummyController(inst,prop);
}
}
```

Python controller

The principle is exactly the same than the one used for C++ controller but we don't have pure virtual methods with a compiler checking if they are defined at compile time. Therefore, it is the pool software which checks that the following methods are defined within the controller class when the controller module is loaded (imported):

- AddDevice
- DeleteDevice
- StartOne or StartAll method
- ReadOne method
- StateOne method

With Python controller, there is no need for function to create controller class instance. With the help of the Python C API, the pool device is able to create the needed instances. Note that the StateOne() method does not have the same signature for Python controller.

1. tuple **Stat** e **One** (self,axe_number) Get a motor state. The method has to return a tuple with two or three elements which are:
   (a) The motor state (as defined by Tango)
   (b) The limit switch state (integer with bit 0 for Home switch, bit 1 for Upper switch and bit 2 for Lower switch)
   (c) A string describing the motor fault if the controller has this feature.

A Python controller class has to inherit from a class called **MotorController** . This does not add any feature but allow the pool software to realize that this class is a motor controller.

Python controller examples

XXX: Unknown layout Subparagraph: A minimum controller code The following is an example of the minimum code structure needed to write a Python controller :

```python
import socket
import PyTango
import MotorController

class MinController(MotorController.MotorController):

#
# Some controller definitions
#

    MaxDevice = 1

#
# Controller methods
#

    def __init__(self,inst,props):
        MotorController.MotorController.__init__(self,inst,props)
        self.inst_name = inst
        self.socket_connected = False
        self.host = "the_host"
        self.port = 1111

#
# Connect to the icepap
#

        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.connect(self.host, self.port)
        self.socket_connected = True

        print "PYTHON -> Connected to", self.host, " on port", self.port


    def AddDevice(self,axis):
        print "PYTHON -> MinController/",self.inst_name,": In AddDevice method for␣
↪axis",axis

    def DeleteDevice(self,axis):
        print "PYTHON -> MinController/",self.inst_name,": In DeleteDevice method␣
↪for axis",axis

    def StateOne(self,axis):
        print "PYTHON -> MinController/",self.inst_name,": In StateOne method for␣
↪axis",axis
        tup = (PyTango.DevState.ON,0)
        return tup

    def ReadOne(self,axis):
        print "PYTHON -> MinController/",self.inst_name,": In ReadOne method for axis
↪",axis
        self.sock.send("Read motor position")
```

(continues on next page)

(continued from previous page)

```
49          pos = self.sock.recv(1024)
50          return pos
51
52      def StartOne(self,axis,pos):
53          print "PYTHON -> MinController/",self.inst_name,": In StartOne method for axis",axis," with pos",pos
54          self.sock.send("Send motor to position pos")
```

Line 11: Definition of the mandatory MaxDevice property set to 1 in this minimum code Line 17-32: The IcePapController constructor code Line 35-36: The AddDevice method Line 38-39: The DeleteDevice method Line 41-44: The StateOne method Line 46-50: The ReadOne method reading motor position from the hardware controller Line 52-54: The StartOne method writing motor position at position pos

XXX: Unknown layout Subparagraph: A full features controller code The following is an example of the code structure needed to write a full features Python controller :

```
import socket
import PyTango
import MotorController

class IcePapController(MotorController.MotorController)
    "This is an example of a Python motor controller class"
#
# Some controller definitions
#

    MaxDevice = 128
    ctrl_features = ['CanDoBacklash']
    ctrl_extra_attributes = {'IceAttribute':{'Type':'DevLong','R/W Type':'READ_WRITE'}}
    class_prop = {'host':{'Type':'DevString','Description':"The IcePap controller
                          host name",'DefaultValue':"IcePapHost"},
                  'port':{'Type':'DevLong','Description':"The port on which the
                          IcePap software is listenning",'DefaultValue':5000}}

#
# Controller methods
#

    def __init__(self,inst,props):
        MotorController.MotorController.__init__(self,inst,props)
        self.inst_name = inst
        self.socket_connected = False

#
# Connect to the icepap
#

        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.connect(self.host, self.port)
        self.socket_connected = True

        print "PYTHON -> Connected to", self.host, " on port", self.port


    def AddDevice(self,axis):
```

(continues on next page)

(continued from previous page)

```
       print "PYTHON -> IcePapController/",self.inst_name,": In AddDevice method␣
↪for axis",axis

    def DeleteDevice(self,axis):
       print "PYTHON -> IcePapController/",self.inst_name,": In DeleteDevice method␣
↪for axis",axis

    def PreReadAll(self):
       print "PYTHON -> IcePapController/",self.inst_name,": In PreReadAll method"
       self.read_pos = []
       self.motor_to_read = []

    def PreReadOne(self,axis):
       print "PYTHON -> IcePapController/",self.inst_name,": In PreReadOne method␣
↪for axis",axis
       self.motor_to_read.append(axis)

    def ReadAll(self):
       print "PYTHON -> IcePapController/",self.inst_name,": In ReadAll method"
       self.sock.send("Read motors in the motor_to_read list")
       self.read_pos = self.sock.recv(1024)

    def ReadOne(self,axis):
       print "PYTHON -> IcePapController/",self.inst_name,": In ReadOne method for␣
↪axis",axis
       return read_pos[axis]

    def PreStartAll(self):
       print "PYTHON -> IcePapController/",self.inst_name,": In PreStartAll method"
       self.write_pos = []
       self.motor_to_write = []

    def PreStartOne(self,axis,pos):
       print "PYTHON -> IcePapController/",self.inst_name,": In PreStartOne method␣
↪for axis",axis," with pos",pos
       return True

    def StartOne(self,axis,pos):
       print "PYTHON -> IcePapController/",self.inst_name,": In StartOne method for␣
↪axis",axis," with pos",pos
       self.write_pos.append(pos)
       self.motor_to_write(axis)

    def StartAll(self):
       print "PYTHON -> IcePapController/",self.inst_name,": In StartAll method"
       self.sock.send("Write motors in the motor_to_write list at position in the␣
↪write_pos list"

    def PreStateAll(self):
       print "PYTHON -> IcePapController/",self.inst_name,": In PreStateAll method"
       self.read_state = []
       self.motor_to_get_state = []

    def PreStateOne(self,axis):
       print "PYTHON -> IcePapController/",self.inst_name,": In PreStateOne method␣
↪for axis",axis
```

(continues on next page)

(continued from previous page)

```
      self.motor_to_get_state.append(axis)

   def StateAll(self):
      print "PYTHON -> IcePapController/",self.inst_name,": In StateAll method"
      self.sock.send("Read motors state for motor(s) in the motor_to_get_state list")
      self.read_state = self.sock.recv(1024)

   def StateOne(self,axis):
      print "PYTHON -> IcePapController/",self.inst_name,": In StateOne method for axis",axis
      one_state = [read_state[axis]]
      return one_state

   def SetPar(self,axis,name,value):
      if name == 'Acceleration'
         print "Setting acceleration to",value
      elif name == 'Deceleration'
         print "Setting deceleartion to",value
      elif name == 'Velocity'
         print "Setting velocity to",value
      elif name == 'Base_rate'
         print "Setting base_rate to",value
      elif name == 'Step_per_unit'
         print "Setting step_per_unit to",value
      elif name == 'Backlash'
         print "Setting backlash to",value

    def GetPar(self,axis,name):
      ret_val = 0.0
      if name == 'Acceleration'
         print "Getting acceleration"
         ret_val = 12.34
       elif name == 'Deceleration'
         print "Getting deceleration"
         ret_val = 13.34
       elif name == 'Velocity'
         print "Getting velocity"
         ret_val = 14.34
       elif name == 'Base_rate'
         print "Getting base_rate"
         ret_val = 15.34
       elif name == 'Backlash'
         print "Getting backlash"
         ret_val = 123
      return ret_val

   def SetExtraAttributePar(self,axis,name,value):
      if name == 'IceAttribute'
         print "Setting IceAttribute to",value

   def GetExtraAttributePar(self,axis,name):
      ret_val = 0.0
      if name == 'IceAttribute'
         print "Getting IceAttribute"
         ret_val = 12.34
```

(continues on next page)

(continued from previous page)

```
142         return ret_val
143
144     def AbortOne(self,axis):
145         print "PYTHON -> IcePapController/",self.inst_name,": Aborting motion for axis:",axis
146
147     def DefinePosition(self,axis,value):
148         print "PYTHON -> IcePapController/",self.inst_name,": Defining position for axis:",axis
149
150     def __del__(self):
151         print "PYTHON -> IcePapController/",self.inst_name,": Aarrrrrg, I am dying"
152
153     def SendToCtrl(self,in_str)
154         print "Python -> MinController/",self.inst_name,": In SendToCtrl method"
155         self.sock.send("The input string")
156         out_str = self.sock.recv(1024)
157         return out_str
```

Line 6 : Definition of the Python DocString which will also be used for the first returned value of the Pool device GetControllerInfo command. See chapter XXX: Unknown inset LatexCommand ref{ite:GetControllerInfo:}: to get all details about this command. Line 11: Definition of the mandatory MaxDevice property set to 128 Line 12: Definition of the pre-defined feature supported by this controller. In this example, only the backlash Line 13: Definition of one controller extra feature called IceFeature Line 14-17: Definition of 2 properties called host and port Line 23-36: The IcePapController constructor code. Note that the object attribute host and port automatically created by the property management are used on line 32 Line 39-40: The AddDevice method Line 42-43: The DeleteDevice method Line 45-48: The PreReadAll method which clears the 2 list read_pos and motor_to_read Line 50-52: The PreReadOne method. It stores which method has to be read in the motor_to_read list Line 54-57: The ReadAll method. It send the request to read motor positions to the controller and stores the result in the internal read_pos list Line 59-61: The ReadOne method returning motor position from the internal read_pos list Line 63-66: The PreStartAll method which clears 2 internal list called write_pos and motor_to_write Line 68-70: The PreStartOne method Line 72-75: The StartOne method which appends in the write_pos and motor_to_write list the new motor position and the motor number which has to be moved Line 77-79: The StartAll method sending the request to the controller Line 81-84: The PreStateAll method which clears 2 internal list called read_state and motor_to_get_state Line 86-88: The PreStateOne method Line 90-93: The StateAll method sending the request to the controller Line 95-98: The StateOne method returning motor state from the internal read_state list Line 100-112: The SetPar method managing the acceleration, deceleration, velocity, base_rate and backlash attributes (because defined in line 11) Line 114-131: The GetPar method managing the same 5 parameters plus the step_per_unit Line 133-135: The SetExtraAttributePar method for the controller extra feature defined at line 12 Line 137-142: The GetExtraAttributePar method for controller extra feature Line 144-145: The AbortOne method Line 147-148: The DefinePosition method Line 153-157: The SendToCtrl method

Defining available controller features

Four data types and two read_write modes are available for the attribute associated with controller features. The possible data type are:

- BOOLEAN
- LONG
- DOUBLE
- STRING

The read_write modes are:

- READ
- READ_WRITE

All the attributes created to deal with controller features and defined as READ_WRITE will be memorized attributes. This means that the attribute will be written with the memorized value just after the device creation by the Tango[352] layer. The definition of a controller features means defining three elements which are the feature name, the feature data type and the feature read_write mode. It uses a C++ structure called MotorFeature with three elements which are a C string (const char *) for the feature name and two enumeration for the feature data type and feature read_write mode. All the available features are defined as an array of these structures in a file called **MotorFeatures.h**

Controller access when creating a motor

When you create a motor (a new one or at Pool startup time), the calls executed on the controller depend if a command "SaveConfig" has already been executed for this motor. If the motor is new and the command SaveConfig has never been executed for this motor, the following controller methods are called:

1. The AddDevice() method
2. The SetPar() method for the Step_per_unit parameter
3. The GetPar() method for the Velocity parameter
4. The GetPar() method for the Acceleration parameter
5. The GetPar() method for the Deceleration parameter
6. The GetPar() method for the Base_rate parameter

If the motor is not new and if a SaveConfig command has been executed on this motor, during Pool startup sequence, the motor will be created and the following controller methods will be called:

1. The AddDevice() method
2. The SetPar() method for the Step_per_unit parameter
3. The SetPar() method for the Velocity parameter
4. The SetPar() method for the Acceleration parameter
5. The SetPar() method for the Deceleration parameter
6. The SetPar() method for the Base_rate parameter
7. The SetExtraAttributePar() method for each of the memorized motor extra attributes

The pool motor group interface

The motor group interface allows the user to move several motor(s) at the same time. It supports several attributes and commands. It is implemented in C++ and is mainly a set of controller methods call or individual motor call. The motor group interface is statically linked with the Pool device server. When creating a group, the user can define as group member three kinds of elements which are :

1. A simple motor
2. Another already created group

[352] http://www.tango-controls.org/

3. A pseudo-motor

Nevertheless, it is not possible to have several times the same physical motor within a group. Therefore, each group has a logical structure (the one defined by the user when the group is created) and a physical structure (the list of physical motors really used in the group).

The states

The motor group interface knows four states which are ON, MOVING, ALARM and FAULT. A motor group device is in MOVING state when one of the group element is in MOVING state. It is in ALARM state when one of the motor is in ALARM state (The underlying motor has reached one of the limit switches). A motor group device is in FAULT state as long as any one of the underlying motor is in FAULT state.

The commands

The motor interface supports 1 command on top of the Tango[353] Init, State and Status command. This command is summarized in the following table:

Command name	Input data type	Output data type
Abort	void	void

- **Abort** : It aborts a running motion. This command does not have input or output argument. It aborts the motion of the motor(s) member of the group which are still moving while the command is received.

The attributes

The motor group supports the following attributes:

Name	Data type	Data format	Writable
Position	Tango::DevVarDoubleStringArray	Spectrum	R/W

- P **osition** : This is a read/write spectrum of double attribute. Each spectrum element is the position of one motor. The order of this array is the order used when the motor group has been created. The size of this spectrum has to be the size corresponding to the motor number when the group is created. For instance, for a group created with 2 motors, another group of 3 motors and one pseudo-motor, the size of this spectrum when written has to be 6 (2 + 3 + 1)

The properties

Each motor group has 6 properties. Five of them are automatically managed by the pool software and must not be changed by the user. These properties are called Motor_group_id, Pool_device, Motor_list, User_group_elt and Pos_spectrum_dim_x. The last property called Sleep_bef_last_read is a user property.This user property is:

Property name	Default value
Sleep_bef_last_read	0

[353] http://www.tango-controls.org/

It defines the time in milli-second that the software managing a motor group motion will wait between it detects the end of the motion of the last group element and the last group motors position reading.

Getting motor group state using event

The simplest way to know if a motor group is moving is to survey its state. If the group is moving, its state will be MOVING. When the motion is over, its state will be back to ON. The pool motor interface allows client interested by group state to use the Tango[354] event system subscribing to motor group state change event. As soon as a group starts a motion, its state is changed to MOVING and an event is sent. As soon as the motion is over, the group state is updated ans another event is sent. Events will also be sent to each motor element of the group when they start moving and when they stop. These events could be sent before before the group state change event is sent in case of group motion with different motor motion for each group member.

Reading the group position attribute

For each motor group, the key attribute is its position. Special care has been taken on this attribute management. When the motor group is not moving (None of the motor are moving), reading the Position attribute will generate calls to the controller(s) and therefore hardware access. When the motor group is moving (At least one of its motor is moving), its position is automatically read every 100 milli- seconds and stored in the Tango[355] polling buffer. This means that a client reading motor group Position attribute while the group is moving will get the position from the Tango[356] polling buffer and will not generate extra controller calls. It is also possible to get a group position using the Tango[357] event system. When the group is moving, an event is sent to the registered clients when the change event criterion is true. By default, this change event criterion is set to be a difference in position of 5. It is tunable on a group basis using the classical group Position attribute "abs_change" property or at the pool device basis using its DefaultMotGrpPos_AbsChange property. Anyway, not more than 10 events could be sent by second. Once the motion is over (None of the motors within the group are moving), the group position is made unavailable from the Tango[358] polling buffer and is read a last time after a tunable waiting time (Sleep_bef_last_read property). A forced change event with this value is sent to clients using events.

The ghost motor group

In order to allow pool client software to be entirely event based, some kind of polling has to be done on each motor to inform them on state change which are not related to motor motion. To achieve this goal, one internally managed motor group is created. Each pool motor is a member of this group. The Tango[359] polling thread polls the state command of this group (Polling period tunable with the pool Ghostgroup_PollingPeriod property). The code of this group state command detects change in every motor state and send a state change event on the corresponding motor. This motor group is not available to client and is even not defined in the Tango[360] database. This is why it is called the ghost group.

[354] http://www.tango-controls.org/
[355] http://www.tango-controls.org/
[356] http://www.tango-controls.org/
[357] http://www.tango-controls.org/
[358] http://www.tango-controls.org/
[359] http://www.tango-controls.org/
[360] http://www.tango-controls.org/

The pool pseudo motor interface

The pseudo motor interface acts like an abstraction layer for a motor or a set of motors allowing the user to control the experiment by means of an interface which is more meaningful to him(her).

Each pseudo motor is represented by a C++ written tango device whose interface allows for the control of a single position (scalar value).

In order to translate the motor positions into pseudo positions and vice versa, calculations have to be performed. The device pool provides a python API class that can be overwritten to provide new calculations.

The states

The pseudo motor interface knows four states which are ON, MOVING, ALARM and FAULT. A pseudo motor device is in MOVING state when at least one motor is in MOVING state. It is in ALARM state when one of the motor is in ALARM state (The underlying motor has reached one of the limit switches. A pseudo motor device is in FAULT state as long as any one of the underlying motor is in FAULT state).

The commands

The pseudo motor interface supports 1 command on top of the Tango Init, State and Status commands. This command is summarized in the following table:

Command name	Input data type	Output data type
Abort	void	void

- **Abort** : It aborts a running movement. This command does not have input or output argument. It aborts the movement of the motor(s) member of the pseudo motor which are still moving while the command is received.

The attributes

The pseudo motor supports the following attributes:

Name	Data type	Data format	Writable
Position	Tango::DevDouble	Scalar	R/W

- **Position** : This is read-write scalar double attribute. With the classical Tango min and max_value, it is easy to define authorized limit for this attribute. It is not allowed to read or write this attribute when the pseudo motor is in FAULT or UNKNOWN state. It is also not possible to write this attribute when the motor is already MOVING.

The PseudoMotor system class

This chapter describes how to write a valid python pseudo motor system class.

Prerequisites

Before writing the first python pseudo motor class for your device pool two checks must be performed:

1. The device pool **PoolPath** property must exist and must point to the directory which will contain your python pseudo motor module. The syntax of this PseudoPath property is the same used in the PATH or PYTHONPATH environment variables. Please see XXX: Unknown inset LatexCommand ref{sub:PoolPath}: for more information on setting this property
2. A PseudoMotor.py file is part of the device pool distribution and is located in <device pool home dir>/py_pseudo. This directory must be in the PYTHONPATH environment variable or it must be part of the **PoolPath** device pool property metioned above

Rules

A correct pseudo motor system class must obey the following rules:

1. the python class PseudoMotor of the PseudoMotor module must be imported into the current namespace by using one of the python import statements:

   ```
   from PseudoMotor import *
   import PseudoMotor or
   from PseudoMotor import PseudoMotor or
   ```

2. the pseudo motor system class being written must be a subclass of the PseudoMotor class (see example below)
3. the class variable **motor_roles** must be set to be a tuple of text descriptions containing each motor role description. It is crucial that all necessary motors contain a textual description even if it is an empty one. This is because the number of elements in this tuple will determine the number of required motors for this pseudo motor class. The order in which the roles are defined is also important as it will determine the index of the motors in the pseudo motor system.
4. the class variable **pseudo_motor_roles** must be set if the pseudo motor class being written represents more than one pseudo motor. The order in which the roles are defined will determine the index of the pseudo motors in the pseudo motor system. If the pseudo motor class represents only one pseudo motor then this operation is optional. If omitted the value will of pseudo_motor_roles will be set to:
5. if the pseudo motor class needs some special parameters then the class variable parameters must be set to be a dictionary of <parameter name> : { <property> : <value> } values where:

   <parameter name> - is a string representing the name of the parameter

   <property> - is one of the following mandatory properties: 'Description', 'Type'. The 'Default Value' property is optional.

   <value> - is the corresponding value of the property. The 'Description' can contain any text value. The 'Type' must be one of available Tango[361] property data types and 'Default Value' must be a string containning a valid value for the corresponding 'Type' value.

6. the pseudo motor class must implement a **calc_pseudo** method with the following signature:

   ```
   number = calc_pseudo(index, physical_pos, params = None)
   ```

   The method will receive as argument the index of the pseudo motor for which the pseudo position calculation is requested. This number refers to the index in the pseudo_motor_roles class variable.

   The physical_pos is a tuple containing the motor positions.

   The params argument is optional and will contain a dictionary of <parameter name> : <value>.

[361] http://www.tango-controls.org/

The method body should contain a code to translate the given motor positions into pseudo motor positions.

The method will return a number representing the calculated pseudo motor position.

7. the pseudo motor class must implement a **calc_physical** method with the following signature:

```
number = calc_physical(index, pseudo_pos, params = None)
```

The method will receive as argument the index of the motor for which the physical position calculation is requested. This number refers to the index in the motor_roles class variable.

The pseudo_pos is a tuple containing the pseudo motor positions.

The params argument is optional and will contain a dictionary of <parameter name> : <value>.

The method body should contain a code to translate the given pseudo motor positions into motor positions.

The method will return a number representing the calculated motor position.

8. Optional implementation of **calc_all_pseudo** method with the following signature:

```
()/[]/number = calc_all_pseudo(physical_pos,params = None)
```

The method will receive as argument a physical_pos which is a tuple of motor positions.

The params argument is optional and will contain a dictionary of <parameter name> : <value>.

The method will return a tuple or a list of calculated pseudo motor positions. If the pseudo motor class represents a single pseudo motor then the return value could be a single number.

9. Optional implementation of **calc_all_physical** method with the following signature:

```
()/[]/number = calc_all_physical(pseudo_pos, params = None)
```

The method will receive as argument a pseudo_pos which is a tuple of pseudo motor positions.

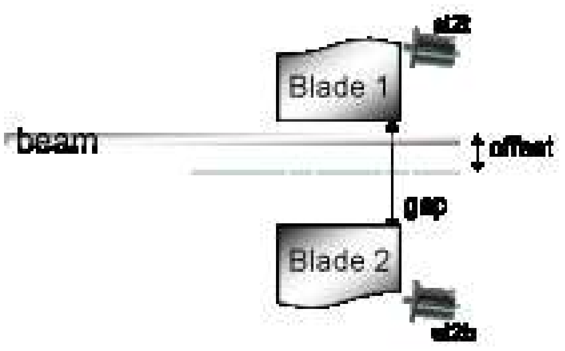
The params argument is optional and will contain a dictionary of <parameter name> : <value>.

The method will return a tuple or a list of calculated motor positions. If the pseudo motor class requires a single motor then the return value could be a single number.

Note: The default implementation **calc_all_physical** and **calc_all_pseudo** methods will call calc_physical and calc_pseudo for each motor and physical motor respectively. Overwriting the default implementation should only be done if a gain in performance can be obtained.

Example

One of the most basic examples is the control of a slit. The slit has two blades with one motor each. Usually the user doesn't want to control the experiment by directly handling these two motor positions since their have little meaning from the experiments perspective.



Instead, it would be more useful for the user to control the experiment by means of changing the gap and offset values. Pseudo motors gap and offset will provide the necessary interface for controlling the experiments gap and offset values respectively.

The calculations that need to be performed are:

$$\begin{cases} gap = sl2t + sl2b \\ offset = \frac{sl2t - sl2b}{2} \end{cases}$$

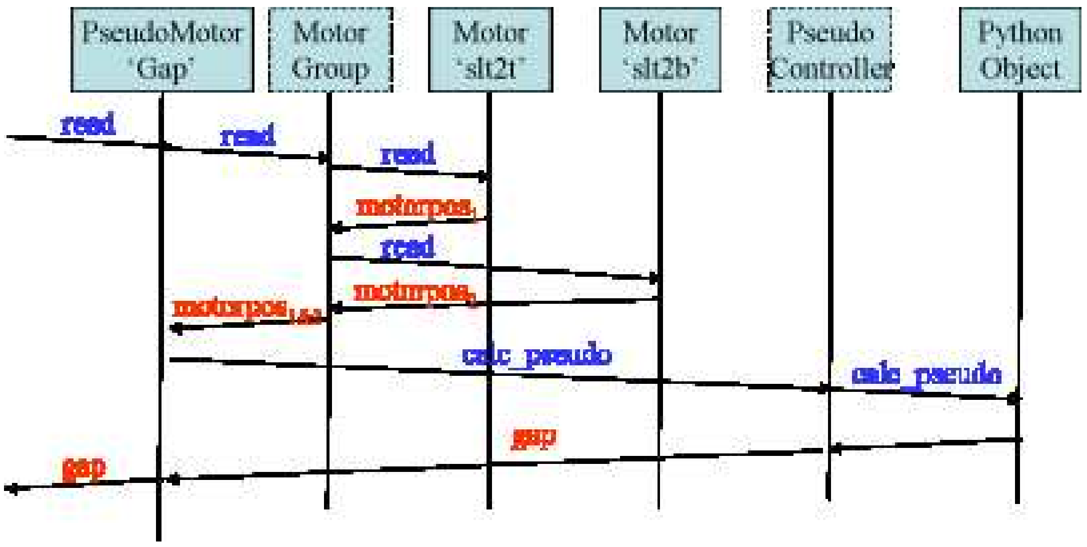
$$\begin{cases} sl2t = -offset + \frac{gap}{2} \\ sl2b = offset + \frac{gap}{2} \end{cases}$$

The corresponding python code would be:

```
class Slit(PseudoMotor):
    """A Slit system for controlling gap and offset pseudo motors."""

    pseudo_motor_roles = ("Gap", "Offset")
    motor_roles = ("Motor on blade 1", "Motor on blade 2")

def calc_physical(self,index,pseudo_pos,params = None):
    half_gap = pseudo_pos[0]/2.0
    if index == 0:
        return -pseudo_pos[1] + half_gap
    else
        return pseudo_pos[1] + half_gap

def calc_pseudo(self,index,physical_pos,params = None):
    if index == 0:
        return physical_pos[1] + physical_pos[0]
    else:
        return (physical_pos[1] - physical_pos[0])/2.0
```

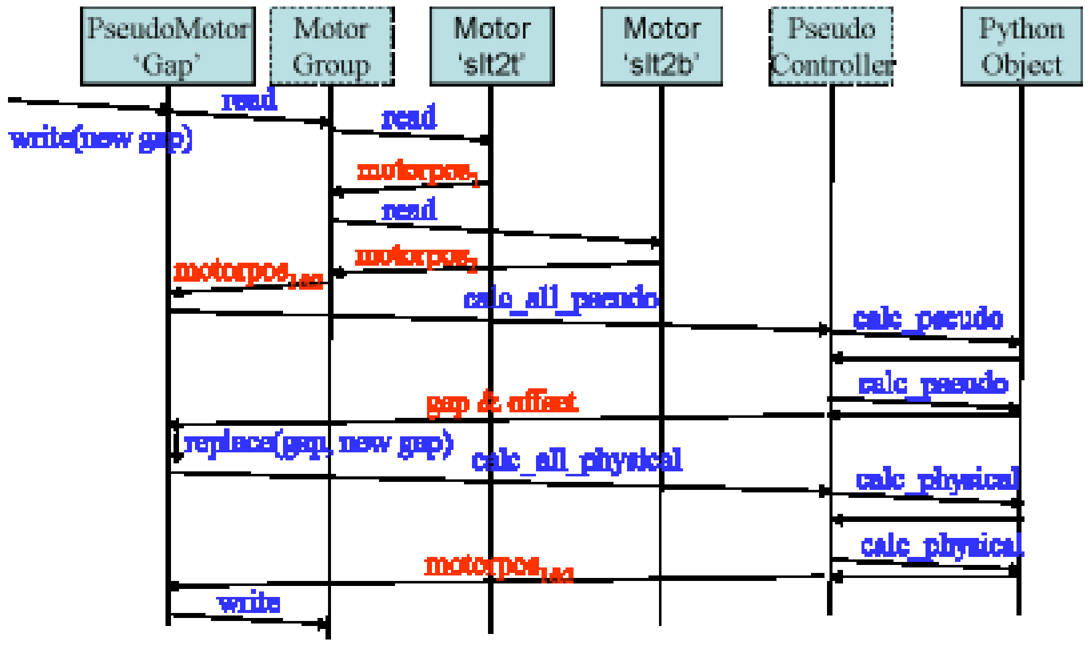
read gap position diagram

The following diagram shows the sequence of operations performed when the position is requested from the gap pseudo motor:



write gap position diagram

The following diagram shows the sequence of operations performed when a new position is written to the gap pseudo motor:



The Counter/Timer interface

The Counter/Timer user interface

The Counter/Timer interface is statically linked with the Pool device server and supports several attributes and commands. It is implemented in C++ and used a set of the so-called "controller" methods. The Counter/Timer interface is always the same whatever the hardware is. This is the rule of the "controller" to access the hardware using the communication link supported by the hardware (network link, Serial line. . .).

The controller code has a well-defined interface and can be written using Python or C++. In both cases, it will be dynamically loaded into the pool device server process.

The states

The Counter/Timer interface knows four states which are *ON*, *MOVING*, **FAULT** and UNKNOWN. A Counter/Timer device is in MOVING state when it is counting! It is in FAULT if its controller software is not available (impossible to load it), if a fault is reported from the hardware controller or if the controller software returns an unforeseen state. The device is in the UNKNOWN state if an exception occurs during the communication between the pool and the hardware controller.

The commands

The Counter/Timer interface supports 2 commands on top of the Tango classical Init, State and Status commands. These commands are summarized in the following table:

Command name	Input data type	Output data type
Start	void	void
Stop	void	void

- **Start** : When the device is used as a counter, this commands allows the counter to start counting. When it is used as a timer, this command starts the timer. This command changes the device state from ON to MOVING. It is not allowed to execute this command if the device is already in the MOVING state.
- **Stop** : When the device is used as a counter, this commands stops the counter. When it is used as a timer, this command stops the timer. This commands changes the device state from MOVING to ON. It is a no action command if this command is received and the device is not in the MOVING state.

The attributes

The Counter/Timer interface supports several attributes which are summarized in the following table:

Name	Data type	Data format	Writable	Memorized	Ope/Expert
Value	Tango::DevDouble	Scalar	R/W	No	Ope
SimulationMode	Tango::DevBoolean	Scalar	R	No	Ope

- **Value** : This is read-write scalar double attribute. Writing the value is used to clear (or to preset) a counter or to set a timer time. For counter, reading the value allows the user to get the count number. For timer, the read value is the elapsed time since the timer has been started. After the acquisition, the value stays unchanged until a new count/time is started. For timer, the unit of this attribute is the second.
- **SimulationMode** : This is a read only scalar boolean attribute. When set, all the counting/timing requests are not forwarded to the software controller and then to the hardware. When set, the device Value is always 0. To set this attribute, the user has to used the pool device Tango interface. It is not allowed to read this attribute when the device is in FAULT or UNKNOWN states.

The properties

Each Counter/Timer device has one property which is automatically managed by the pool software and must not be changed by the user. This property is named Channel_id.

The Counter/Timer controller

The CounterTimer controller follows the same principles already explained for the Motor controller in chapter XXX: Unknown inset LatexCommand ref{sub:The-Motor-Controller}:

The basic

For Counter/Timer, the pre-defined set of methods which has to be implemented can be splitted in 7 different types which are:

1. Methods to create/remove counter/timer experiment channel
2. Methods to get channel(s) state
3. Methods to read channel(s)
4. Methods to load channel(s)
5. Methods to start channel(s)
6. Methods to configure a channel
7. Remaining method

The CounterTimer controller features

Not defined yet

The CounterTimer controller extra attributes

The definition is the same than the one defined for Motor controller and explained in chapter XXX: Unknown inset LatexCommand ref{par:Specifying-the-motor}:

Methods to create/remove Counter Timer Channel

Two methods are called when creating or removing counter/timer channel from a controller. These methods are called **AddDevice** and **DeleteDevice** . The AddDevice method is called when a new channel belonging to the controller is created within the pool. The DeleteDevice method is called when a channel belonging to the controller is removed from the pool.

Method(s) to get Counter Timer Channel state.

These methods follow the same definition than the one defined for Motor controller which are detailed in chapter XXX: Unknown inset LatexCommand ref{par:Methods-to-get-state}: .

Method(s) to read Counter Timer Experiment Channel

Four methods are used when a request to read channel(s) value is received. These methods are called PreReadAll, PreReadOne, ReadAll and ReadOne. The algorithm used to read value of one or several channels is the following :

```
/FOR/ Each controller(s) implied in the reading
     - Call PreReadAll()
/END FOR/

/FOR/ Each channel(s) implied in the reading
     - PreReadOne(channel to read)
/END FOR/

/FOR/ Each controller(s) implied in the reading
     - Call ReadAll()
/END FOR/

/FOR/ Each channel(s) implied in the reading
     - Call ReadOne(channel to read)
/END FOR/
```

The following array summarizes the rule of each of these methods:

Default action	Does nothing	Does nothing	Does nothing	Print message on the screen and returns NaN. Mandatory for Python
Externally called by	Reading the Value attribute	Reading the Value attribute	Reading the Value attribute	Reading the Value attribute
Internally called	Once for each implied controller	For each implied channel	For each implied controller	Once for each implied channel
Typical rule	Init internal data for reading	Memorize which channel has to be read	Send order to physical controller	Return channel value from internal data

This algorithm covers the sophisticated case where a physical controller is able to read several channels positions at the same time. For some simpler controller, it is possible to implement only the ReadOne() method. The default implementation of the three remaining methods is defined in a way that the algorithm works even in such a case.

Method(s) to load Counter Timer Experiment Channel

Four methods are used when a request to load channel(s) value is received. These methods are called PreLoadAll, PreLoadOne, LoadAll and LoadOne. The algorithm used to load value in one or several channels is the following:

```
/FOR/ Each controller(s) implied in the loading
     - Call PreLoadAll()
/END FOR/

```

(continues on next page)

(continued from previous page)

```
/FOR/ Each channel(s) implied in the loading
     - ret = PreLoadOne(channel to load,new channel value)
     - /IF/ ret is true
          - Call LoadOne(channel to load, new channel value)
     - /END IF/
/END FOR/

/FOR/ Each controller(s) implied in the loading
     - Call LoadAll()
/END FOR/
```

The following array summarizes the rule of each of these methods:

Default action	Does nothing	Returns true	Does nothing	Does nothing
Externally called by	Writing the Value attribute	Writing the Value attribute	Writing the Value attribute	Writing the Value attribute
Internally called	Once for each implied controller	For each implied channel	For each implied channel	Once for each implied controller
Typical rule	Init internal data for loading	Check if counting is possible	Set new channel value in internal data	Send order to physical controller

This algorithm covers the sophisticated case where a physical controller is able to write several channels positions at the same time. For some simpler controller, it is possible to implement only the LoadOne() method. The default implementation of the three remaining methods is defined in a way that the algorithm works even in such a case.

Method(s) to start Counter Timer Experiment Channel

Four methods are used when a request to start channel(s) is received. These methods are called PreStartAllCT, PreStartOneCT, StartAllCT and StartOneCT. The algorithm used to start one or several channels is the following:

```
/FOR/ Each controller(s) implied in the starting
     - Call PreStartAllCT()
/END FOR/

/FOR/ Each channel(s) implied in the starting
     - ret = PreStartOneCT(channel to start)
     - /IF/ ret is true
          - Call StartOneCT(channel to start)
     - /END IF/
/END FOR/

/FOR/ Each controller(s) implied in the starting
     - Call StartAllCT()
/END FOR/
```

The following array summarizes the rule of each of these methods:

Default action	Does nothing	Returns true	Does nothing	Does nothing
Externally called by	The Start command	The Start command	The Start command	The Start command
Internally called	Once for each implied controller	For each implied channel	For each implied channel	Once for each implied controller
Typical rule	Init internal data for starting	Check if starting is possible	Set new channel value in internal data	Send order to physical controller

This algorithm covers the sophisticated case where a physical controller is able to write several channels positions at the same time. For some simpler controller, it is possible to implement only the StartOneCT() method. The default implementation of the three remaining methods is defined in a way that the algorithm works even in such a case.

Methods to configure Counter Timer Experiment Channel

The rule of these methods is to

- Get or Set channel extra attribute(s) parameter with methods called GetExtraAttributePar() or SetExtraAttributePar()

The following table summarizes the usage of these methods:

Called by	Reading any of the extra attributes	Writing any of the extra attributes
Rule	Get extra attribute value from the physical layer	Set additional attribute value in physical controller

The GetExtraAttributePar() default implementation returns a string set to "Pool_meth_not_implemented".

Remaining methods

The rule of the remaining methods is to

- Send a raw string to the controller with a method called SendToCtrl()
- Abort a counting counter/timer with a method called AbortOne()

The following table summarizes the usage of this method:

Called by	The Pool SendToController command	The Stop CounterTimer command
Rule	Send the input string to the controller and returns the controller answer	Abort a running count

The Counter Timer controller properties (including the MaxDevice

property)

The definition is the same than the one defined for Motor controller and explained in chapter XXX: Unknown inset LatexCommand ref{par:Controller-properties}:

C++ controller

For C++, the controller code is implemented as a set of classes: A base class called **Controller** and a class called **CoTiController** which inherits from Controller. Finally, the user has to write its controller class which inherits from CoTiController. The Controller class has already been detailed in XXX: Unknown inset LatexCommand ref{sub:The-Cpp-Controller-class}: .

XXX: Unknown layout Subparagraph: The CoTiController class The CoTiController class defines thirteen virtual methods which are:

1. void **CoTiController::PreReadAll** () The default implementation does nothing
2. void **CoTiController::PreReadOne** (long idx_to_read) The default implementation does nothing
3. void **CoTiController::ReadAll** () The default implementation does nothing
4. double **CoTiController::ReadOne** (long idx_to_read) The default implementation prints a message on the screen and return a NaN value
5. void **CoTiController::PreLoadAll** () The default implementation does nothing
6. bool **CoTiController::PreLoadOne** (long idx_to_load,double new_value) The default implementation returns true
7. void **CoTiController::LoadOne** (long idx_to_load,double new_value) The default implementation does nothing
8. void **CoTiController::LoadAll** () The default implementation does nothing
9. void **CoTiController::PreStartAllCT** () The default implementation does nothing
10. bool **CoTiController::PreStartOneCT** (long idx_to_start) The default implementation returns true
11. void **CoTiController::StartOneCT** (long idx_to_start) The default implementation does nothing
12. void **CoTiController::StartAllCT** () The default implementation does nothing
13. void **CoTiController::AbortOne** (long idx_to_abort) The default implementation does nothing

This class has one constructor which is

1. **CoTiController::CoTiController** (const char *) Constructor of the CoTiController class with the controller instance name as parameter

XXX: Unknown layout Subparagraph: The user controller class The user has to implement the remaining pure virtual methods (AddDevice and DeleteDevice) and has to re-define virtual methods if the default implementation does not cover his needs. The controller code has to define two global variables which are:

1. **CounterTimer_Ctrl_class_name** : This is an array of classical C strings terminated by a NULL pointer. Each array element is the name of a Counter Timer Channel controller defined in the file.
2. **<CtrlClassName>_MaxDevice** : Idem motor controller definition

On top of that, a controller code has to define a C function to create the controller object. This is similar to the Motor controller definition which is documented in XXX: Unknown inset LatexCommand ref{par:The-user-controller}:

Python controller

The principle is exactly the same than the one used for C++ controller but we don't have pure virtual methods with a compiler checking if they are defined at compile time. Therefore, it is the pool software which checks that the following methods are defined within the controller class when the controller module is loaded (imported):

- AddDevice
- DeleteDevice
- ReadOne method
- StateOne method
- StartOneCT or StartAllCT method
- LoadOne or LoadAll method

With Python controller, there is no need for function to create controller class instance. With the help of the Python C API, the pool device is able to create the needed instances. Note that the StateOne() method does not have the same signature for Python controller.

1. tuple **Stat** e **One** (self,idx_number) Get a channel state. The method has to return a tuple with one or two elements which are:
   (a) The channel state (as defined by Tango)
   (b) A string describing the motor fault if the controller has this feature.

A Python controller class has to inherit from a class called **CounterTimerController** . This does not add any feature but allows the pool software to realize that this class is a Counter Timer Channel controller.

The Unix Timer

A timer using the Unix getitimer() and setitimer() system calls is provided. It is a Counter/Timer C++ controller following the definition of the previous chapter. Therefore, the device created using this controller will have the Tango[362] interface as the one previously described.

The Unix Timer controller shared library is called **UxTimer.so** and the Controlller class is called **UnixTimer** . This controller is foresee to have only one device (MaxDevice = 1)

The ZeroDExpChannel interface

The ZeroDExpChannel is used to access any kind of device which returns a scalar value and which are not counter or timer. Very often (but not always), this is a commercial measurement equipment connected to a GPIB bus. In order to have a precise as possible measurement, an acquisition loop is implemented for these ZeroDExpChannel device. This acquisition loop will simply read the data from the hardware as fast as it can (only "sleeping" 20 mS between each reading) and a computation is done on the resulting data set to return only one value. Three types of computation are foreseen. The user selects which one he needs with an attribute. The time during which this acquisition loop will get data is also defined by an attribute

[362] http://www.tango-controls.org/

The ZeroDExpChannel user interface

The ZeroDExpChannel interface is statically linked with the Pool device server and supports several attributes and commands. It is implemented in C++ and used a set of the so-called "controller" methods. The ZeroDExpChannel interface is always the same whatever the hardware is. This is the rule of the "controller" to access the hardware using the communication link supported by the hardware (network link, GPIB…).

The controller code has a well-defined interface and can be written using Python or C++. In both cases, it will be dynamically loaded into the pool device server process.

The states

The ZeroDExpChannel interface knows five states which are ON, MOVING, ALARM, FAULT and UNKNOWN. A ZeroDExpChannel device is in MOVING state when it is acquiring data! It is in ALARM state when at least one error has occured during the last acquisition. It is in FAULT if its controller software is not available (impossible to load it), if a fault is reported from the hardware controller or if the controller software returns an unforeseen state. The device is in the UNKNOWN state if an exception occurs during the communication between the pool and the hardware controller.

The commands

The ZeroDExpChannel interface supports 2 commands on top of the Tango classical Init, State and Status commands. These commands are summarized in the following table:

Command name	Input data type	Output data type
Start	void	void
Stop	void	void

- **Start** : Start the acquisition for the time defined by the attribute CumulatedTime. If the CumulatedTime attribute value is 0, the acquisition will not automatically stop until a Stop command is received. This command changes the device state from ON to MOVING. It is not allowed to execute this command if the device is already in the MOVING state.
- **Stop** : Stop the acquisition. This commands changes the device state from MOVING to ON. It is a no action command if this command is received and the device is not in the MOVING state.

The attributes

The ZeroDExpChannel interface supports several attributes which are summarized in the following table:

Name	Data type	Data format	Writable	Memorized	Ope/Expert
Value	Tango::DevDouble	Scalar	R	No	Ope
CumulatedValue	Tango::DevDouble	Scalar	R	No	Ope
CumulationTime	Tango::DevDouble	Scalar	R/W	Yes	Ope
CumulationType	Tango::DevLong	Scalar	R/W	Yes	Ope
CumulatedPointsNumber	Tango::DevLong	Scalar	R	No	Ope
CumulatedPointsError	Tango::DevLong	Scalar	R	No	Ope
SimulationMode	Tango::DevBoolean	Scalar	R	No	Ope

- **Value** : This is read scalar double attribute. This is the live value reads from the hardware through the controller

- **CumulatedValue** : This is a read scalar double attribute. This is the result of the data acquisition after the computation defined by the CumulationType attribute has been applied. This value is 0 until an acquisition has been started. After an acquisition, the attribute value stays unchanged until the next acquisition is started. If during the acquisition some error(s) has been received while reading the data, the attribute quality factor will be set to ALARM
- **CumulationTime** : This is a read-write scalar double and memorized attribute. This is the acquisition time in seconds. The acquisition will automatically stops after this CumulationTime. Very often, reading the hardware device to get one data is time-consuming and it is not possible to read the hardware a integer number of times within this CumulationTime. A device property called StopIfNoTime (see XXX: Unknown inset LatexCommand ref{ite:StopIfNoTime:-A-boolean}:) allows the user to tune the acquisition loop.
- **CumulationType** : This a read-write scalar long and memorized attribute. Defines the computation type done of the values gathered during the acquisition. Three type of computation are supported:
  1. Sum: The CumulatedValue attribute is the sum of all the data read during the acquisition. This is the default type.
  2. Average: The CumulatedValue attribute is the average of all the data read during the acquisition
  3. Integral: The CumulatedValue attribute is a type of the integral of all the data read during the acquisition
- **CumulatedPointsNumber** : This is a read scalar long attribute. This is the number of data correctly read during the acquisition. The attribute value is 0 until an acquisition has been started and stay unchanged between the end of the acquisition and the start of the next one.
- **CumulatedPointsError** : This is a read scalar long attribute. This is the number of times it was not possible to read the data from the hardware due to error(s). The property ContinueOnError allows the user to define what to do in case of error. The attribute value is 0 until an acquisition has been started and stay unchanged between the end of the acquisition and the start of the next one.
- **SimulationMode** : This is a read only scalar boolean attribute. When set, all the acquisition requests are not forwarded to the software controller and then to the hardware. When set, the device Value, CumulatedValue, CumulatedPointsNumber and CumulatedPointsError are always 0. To set this attribute, the user has to used the pool device Tango[363] interface. The value of the CumulationTime and CumulationType attributes are memorized at the moment this attribute is set. When this mode is turned off, if the value of any of the previously memorized attributes has changed, it is reapplied to the memorized value. It is not allowed to read this attribute when the device is in FAULT or UNKNOWN states.

The properties

Each ZeroDExpChannel device has a set of properties. One of these properties is automatically managed by the pool software and must not be changed by the user. This property is named Channel_id. The user properties are:

Property name	Default value
StopIfNoTime	true
ContinueOnError	true

- XXX: Unknown inset LatexCommand label{ite:StopIfNoTime:-A-boolean}: **StopIfNoTime** : A boolean property. If this property is set to true, the acquisition loop will check before acquiring a new data that it has enough time to do this. To achieve this, the acquisition loop measures the time

[363] http://www.tango-controls.org/

needed by the previous data read and checks that the actual time plus the acquisition time is still less than the CumulationTime. If not, the acquisition stops. When this property is set to false, the acquisition stops when the acquisition time is greater or equal than the CumulationTime

- **ContinueOnError** : A boolean property. If this property is set to true (the default), the acquisition loop continues reading the data even after an error has been received when trying to read data. If it is false, the acquisition stops as soon as an error is detected when trying to read data from the hardware.

Getting ZeroDExpChannel state using event

The simplest way to know if a Zero D Experiment Channel is acquiring data is to survey its state. If the device is acquiring data, its state will be MOVING. When the acquisition is over, its state will be back to ON. The pool ZeroDExpChannel interface allows client interested by Experiment Channel state value to use the Tango[364] event system subscribing to channel state change event. As soon as a channel starts an acquisition, its state is changed to MOVING and an event is sent. As soon as the acquisition is over (for one reason or another), the channel state is updated and another event is sent.

XXX: Unknown inset LatexCommand label{par:Reading-the-ZeroDExpChannel}:

Reading the ZeroDExpChannel CumulatedValue attribute

During an acquisition, events with CumulatedValue attribute are sent from the device server to the interested clients. The acquisition loop will periodically read this event and fire an event. The first and the last events fired during the acquisition loop do not check the change event criteria. The other during the acquisition loop check the change event criteria

The ZeroDExpChannel Controller

The ZeroDExpChannel controller follows the same principles already explained for the Motor controller in chapter XXX: Unknown inset LatexCommand ref{sub:The-Motor-Controller}:

The basic

For Zero Dimension Experiment Channel, the pre-defined set of methods which has to be implemented can be splitted in 5 different types which are:

1. Methods to create/remove zero dimension experiment channel
2. Methods to get channel(s) state
3. Methods to read channel(s)
4. Methods to configure a channel
5. Remaining method

The ZeroDExpChannel controller features

Not defined yet

[364] http://www.tango-controls.org/

The ZeroDExpChannel controller extra attributes

The definition is the same than the one defined for Motor controller and explained in chapter XXX: Unknown inset LatexCommand ref{par:Specifying-the-motor}:

Methods to create/remove Zero D Experiment Channel

Two methods are called when creating or removing experiment channel from a controller. These methods are called **AddDevice** and **DeleteDevice** . The AddDevice method is called when a new channel belonging to the controller is created within the pool. The DeleteDevice method is called when a channel belonging to the controller is removed from the pool.

Method(s) to get Zero D Experiment Channel state.

These methods follow the same definition than the one defined for Motor controller which are detailed in chapter XXX: Unknown inset LatexCommand ref{par:Methods-to-get-state}: .

Method(s) to read Zero D Experiment Channel

Four methods are used when a request to read channel(s) value is received. These methods are called PreReadAll, PreReadOne, ReadAll and ReadOne. The algorithm used to read value of one or several channels is the following:

```
/FOR/ Each controller(s) implied in the reading
     - Call PreReadAll()
/END FOR/

/FOR/ Each channel(s) implied in the reading
     - PreReadOne(channel to read)
/END FOR/

/FOR/ Each controller(s) implied in the reading
     - Call ReadAll()
/END FOR/

/FOR/ Each channel(s) implied in the reading
     - Call ReadOne(channel to read)
/END FOR/
```

The following array summarizes the rule of each of these methods:

Default action	Does nothing	Does nothing	Does nothing	Print message on the screen and returns NaN. Mandatory for Python
Externally called by	Reading the Value attribute	Reading the Value attribute	Reading the Value attribute	Reading the Value attribute
Internally called	Once for each implied controller	For each implied channel	For each implied controller	Once for each implied channel
Typical rule	Init internal data for reading	Memorize which channel has to be read	Send order to physical controller	Return channel value from internal data

This algorithm covers the sophisticated case where a physical controller is able to read several channels positions at the same time. For some simpler controller, it is possible to implement only the ReadOne() method. The default implementation of the three remaining methods is defined in a way that the algorithm works even in such a case.

Methods to configure Zero D Experiment Channel

The rule of these methods is to

- Get or Set channel extra attribute(s) parameter with methods called GetExtraAttributePar() or SetExtraAttributePar()

The following table summarizes the usage of these methods:

Called by	Reading any of the extra attributes	Writing any of the extra attributes
Rule	Get extra attribute value from the physical layer	Set additional attribute value in physical controller

The GetExtraAttributePar() default implementation returns a string set to "Pool_meth_not_implemented".

Remaining method

The rule of the remaining method is to

- Send a raw string to the controller with a method called SendToCtrl()

The following table summarizes the usage of this method:

Called by	The Pool SendToController command
Rule	Send the input string to the controller and returns the controller answer

The ZeroDExpChannel controller properties (including the MaxDevice property)

The definition is the same than the one defined for Motor controller and explained in chapter XXX: Unknown inset LatexCommand ref{par:Controller-properties}:

C++ controller

For C++, the controller code is implemented as a set of classes: A base class called **Controller** and a class called **ZeroDController** which inherits from Controller. Finally, the user has to write its controller class which inherits from ZeroDController. The Controller class has already been detailed in XXX: Unknown inset LatexCommand ref{sub:The-Cpp-Controller-class}: .

XXX: Unknown layout Subparagraph: The ZeroDController class The ZeroDController class defines four virtual methods which are:

1. void **ZeroDController::PreReadAll** () The default implementation does nothing
2. void **ZeroDController::PreReadOne** (long idx_to_read) The default implementation does nothing
3. void **ZeroDController::ReadAll** () The default implementation does nothing
4. double **ZeroDController::ReadOne** (long idx_to_read) The default implementation prints a message on the screen and return a NaN value

This class has one constructor which is

1. **ZeroDController::ZeroDController** (const char *) Constructor of the ZeroDController class with the controller instance name as parameter

XXX: Unknown layout Subparagraph: The user controller class The user has to implement the remaining pure virtual methods (AddDevice and DeleteDevice) and has to re-define virtual methods if the default implementation does not cover his needs. The controller code has to define two global variables which are:

1. **ZeroDExpChannel_Ctrl_class_name** : This is an array of classical C strings terminated by a NULL pointer. Each array element is the name of a ZeroDExpChannel controller defined in the file.
2. **<CtrlClassName>_MaxDevice** : Idem motor controller definition

On top of that, a controller code has to define a C function to create the controller object. This is similar to the Motor controller definition which is documented in XXX: Unknown inset LatexCommand ref{par:The-user-controller}:

Python controller

The principle is exactly the same than the one used for C++ controller but we don't have pure virtual methods with a compiler checking if they are defined at compile time. Therefore, it is the pool software which checks that the following methods are defined within the controller class when the controller module is loaded (imported):

- AddDevice
- DeleteDevice
- ReadOne method
- StateOne method

With Python controller, there is no need for function to create controller class instance. With the help of the Python C API, the pool device is able to create the needed instances. Note that the StateOne() method does not have the same signature for Python controller.

1. tuple **Stat** e **One** (self,idx_number) Get a channel state. The method has to return a tuple with one or two elements which are:

   (a) The channel state (as defined by Tango)

   (b) A string describing the motor fault if the controller has this feature.

A Python controller class has to inherit from a class called **ZeroDController** . This does not add any feature but allows the pool software to realize that this class is a Zero D Experiment Channel controller.

The OneDExpChannel interface

To be filled in

The TwoDExpChannel interface

To be filled in

The Measurement Group interface

The measurement group interface allows the user to access several data acquisition channels at the same time. It is implemented as a C++ Tango[365] device that is statically linked with the Pool device server. It supports several attributes and commands.

The measurement group is the key interface to be used when getting data. The Pool can have several measurement groups but only one will be 'in use' at a time. When creating a measurement group, the user can define four kinds of channels which are:

1. Counter/Timer
2. ZeroDExpChannel
3. OneDExpChannel
4. TwoDExpChannel

In order to properly use the measurement group, one of the channels has to be defined as the timer or the monitor. It is not possible to have several times the same channel in a measurement group. It is also not possible to create two measurement groups with exactly the same channels.

The States

The measurement group interface knows five states which are ON, MOVING, ALARM, FAULT. A group is in MOVING state when it is acquiring data (which means that the timer/monitor channel is in MOVING state). A STANDBY state means that the group is not the current active group of the Pool it belongs to. An ON state means that the group is ready to be used. ALARM means that no timer or monitor are defined for the group. If at least one of the channels reported a FAULT by the controller(s) of that(those) channel(s), the group will be in FAULT state.

[365] http://www.tango-controls.org/

The commands

The measurement group interface supports three commands on top of the Tango[366] Init, State and Status commands. These commands are summarized in the following table:

Command name	Input data type	Output data type
Start	void	void
Abort	void	void
AddExpChannel	String	void
RemoveExpChannel	String	void

- **Start** : When the device is in timer mode (Integration_time attribute > 0), it will start counting on all channels at the same time until the timer channel reaches a value of the Integration_time attribute. When the device in in monitor mode (Integration_count attribute > 0), it will start counting on all channels at the same time until de monitor channel reaches the value of the Integration_count attribute. For more details on setting the acquisition mode see XXX: Unknown inset LatexCommand ref{Measurement Group: The attributes}: . This command will change the device state to MOVING. It will not be allowed to execute this command if the device is already in MOVING state. This command does not have any input or output arguments. The state will change from MOVING to ON only when the last channel reports that its acquisition has finished.
- **Abort** : It aborts the running data acquisition. It will stop each channel member of the measurement group. This command does not have any input or output arguments.
- **AddExpChannel** : adds a new experiment channel to the measurement group. The given string argument must be a valid experiment channel in the pool and must not be one of the channels of the measurement group. An event will be sent on the corresponding attribute representing the list of channels in the measurement group. For example, if the given channel is a Counter/Timer channel, then an event will be sent for the attribute "Counters "(See below for a list of attributes in the measurement group).
- **RemoveExpChannel** : removes the given channel from the measurement group. The given string argument must be a valid experiment channel in the measurement group. If the channel to be deleted is the current Timer/Monitor then the value for the corresponding attribute will be set to "Not Initialized "and an event will be sent. An event will be sent on the corresponding attribute representing the list of channels in the measurement group.

XXX: Unknown inset LatexCommand label{Measurement Group: The attributes}:

The attributes

A measurement group will support 8+n (n being the number of channels) attributes summarized in the following table: == ================ ===================== ======== ========= ========== Name Data type Data format Writable Memorized Ope/Expert == ================ ===================== ======== ========= ========== Integration_time Tango::DevDouble Scalar R/W Yes Ope Integration_count Tango::DevLong Scalar R/W Yes Ope Timer Tango::DevString Scalar R/W Yes Ope Monitor Tango::DevString Scalar R/W Yes Ope Counters Tango::DevString Spectrum R No Ope ZeroDExpChannels Tango::DevString Spectrum R No Ope OneDExpChannels Tango::DevString Spectrum R No Ope TwoDExpChannels Tango::DevString Spectrum R No Ope <channel_name ¡ >_Value Tango::DevDouble Scalar/Spectrum/Image R No Ope == ================ ===================== ======== ========= ==========

[366] http://www.tango-controls.org/

- **Integration_time** : The group timer integration time. Setting this value to >0 will set the measurement group acquisition mode to timer. It will force Integration_count attribute to 0 (zero). It will also exclude the current Timer channel from the list of Counters. Units are in seconds.
- **Integration_count** : The group monitor count value. Setting this value to >0 will set the measurement group acquisition mode change to monitor. It will force Integration_time attribute to 0 (zero).
- **Timer** : The name of the channel used as a Timer. A "Not Initialized "value means no timer is defined
- **Monitor** : The name of the channel used as a Monitor. A "Not Initialized "value means no timer is defined
- **Counter** : The list of counter names in the group
- **ZeroDExpChannels** : The list of 0D Experiment channel names in the group
- **OneDExpChannels** : The list of 1D Experiment channel names in the group
- **TwoDExpChannels** : The list of 2D Experiment channel names in the group
- **<channel_name $_i$ >_Value** : (with $0 \leq i < n$) attributes dynamically created (one for each channel) which will contain the corresponding channel Value(for Counter/Timer, 1D or 2DExpChannels), CumulatedValue(for 0DExpChannels). For Counter/Timers and 0DExpChannels the data format will be Scalar. For 1DExpChannels it will be Spectrum and for 2DExpChannels it will be Image.

The properties

Device properties

Each measurement group has five properties. All of them are managed automatically by the pool software and must not be changed by the user. These properties are called Measurement_group_id, Pool_device, CT_List, ZeroDExpChannel_List, OneDExpChannel_List, TwoDExpChannel_List.

XXX: Unknown inset LatexCommand label{measurement group:Checking-operation-modes}:

Checking operation mode

Currently, the measurement group supports two operation modes. The table below shows how to determine the current mode for a measurement group.

mode	Integration_time	Integration_count
Timer	>0.0	0
Monitor	0.0	>0
Undef	0.0	0

'Undef' means no valid values are defined in Integration_time and in Integration_count. You will not be able to execute the Start command in this mode.

Getting measurement group state using event

The simplest way to know if a measurement group is acquiring data is to survey its state. If a measurement group is acquiring data its state will be MOVING. When the data acquisition is over, its state will change back to ON. The data acquisition is over when the measurement group detects that all channels finished acquisition (their state changed from MOVING to ON).The pool group interface allows clients interested in group state to use the Tango[367] event system subscribing to measurement group state change event. As soon as a group starts acquiring data, its state is changed to MOVING and an event is sent. A new event will be sent when the data acquisition ends. Events will also be sent to each channel of the group when they start acquiring data and when they stop.

Reading the measurement group channel values

For each measurement group there is a set of key dynamic attributes representing the value of each channel in the group. They are named <channel_name $_i$ >_Value. Special care has been taken on the management of these attributes with distinct behavior depending on the type of channel the attribute represents (Counter/Timer, 0D, 1D or 2D channel).

Counter/Timer channel values

A Counter/Timer Value is represented by a scalar read-only double attribute. When the measurement group is not taking data, reading the counter/timer value will generate calls to the controller and therefore hardware access. When the group is taking data (master channel is moving), the value of a counter/timer is read every 100 miliseconds and stored in the Tango[368] polling buffer. This means that a client reading the value of the channel while the group is moving will get the value from the Tango[369] polling buffer and will not generate exra controller calls. It is also possible to get the value using the Tango event system. When the group is moving, an event is sent to the registered clients when the change event criteria is true. This is applicable for each Counter/Timer channel in the group. By default, this change event criterion is set to be an absolute difference in the value of 5.0. It is tunable by attribute using the classical "abs_change "property or the pool device basis using its defaultCtGrpVal_AbsChange property. Anyway, not more than 10 events could be sent by second. Once the data acquisition is over, the value is made unavailable from the Tango[370] polling buffer and is read a last time. A forced change event is sent to clients using events.

Zero D channel values

A ZeroDExpChannel CumulatedValue is represented by a scalar read-only double attribute. Usually a ZeroDChannel represents the value of a single device (ex.: multimeter). Therefore, has hardware access cannot be optimized for a group of devices, reading the value on the measurement group device attribute has exactly the same behavior as reading it directly on the CumulatedValue attribute of the ZeroDChannel device (see XXX: Unknown inset LatexCommand ref{par:Reading-the-ZeroDExpChannel}:).

One D channel values

To be filled in

Two D channel values

To be filled in

[367] http://www.tango-controls.org/
[368] http://www.tango-controls.org/
[369] http://www.tango-controls.org/
[370] http://www.tango-controls.org/

Performance

Measurement group devices can often contain many channels. Client applications often request channel values for the set (or subset) of channels in a group. Read requests for these channel values through the <channel_name $_i$ >_Value attributes of a measurement group should be done by clients in groups as often as possible. This can be achieved by using the client Tango[371] API call read_attributes on a DeviceProxy object. This will ensure maximum performance by minimizing hardware access since the measurement group can order channel value requests per controller thus avoiding unecessary calls to the hardware.

Measurement group configuration

Timer/Monitor

Measurement group operation mode can be checked/set through the Integration_time and Integration_count (see XXX: Unknown inset LatexCommand ref{measurement group:Checking-operation-modes}:). Setting the Integration_time to >0.0 will make the data acquisition (initiated by the invoking the Start command) finish when the channel defined in the Timer attribute reaches the value of Integration_time. Setting the Integration_count to >0 will make the data acquisition (initiated by the invoking the Start command) finish when the channel defined in the Monitor attribute reaches the value of Integration_count.

In either case, the measurement group will NOT assume that the master channel(timer/monitor) is able to stop all the other channels in the group, so it will force a Stop on these channels as soon as it detects that the master has finished. This is the case of the UnixTimer channel which itself has no knowledge of the channels involved and therefore is not able to stop them directly.

Integration_time, Integration_count, timer and monitor are memorized attributes. This means that the configuration values of these attributes are stored in the database. The next time the Pool starts the values are restored. This is done in order to reduce Pool configuration at startup to the minimum.

The ghost measurement group

In order to allow pool client software to be entirely event based, some kind of polling has to be done on each channel to inform them on state change which are not related to data acquisition. To achieve this goal, one internally managed measurement group is created. Each pool channel (counter/timer, 0D, 1D or 2D experiment channel) is a member of this group. The Tango[372] polling thread polls the state command of this group (Polling period tunable with the pool Ghostgroup_PollingPeriod property). The code of this group state command detects change in every channel state and send a state change event on the corresponding channel. This measurment group is not available to client and is even not defined in the Tango[373] database. This is why it is called the ghost measurement group.

The pool serial line, GPIB, socket interfaces

To be filled in

The pool Modbus interface

To be filled in

[371] http://www.tango-controls.org/
[372] http://www.tango-controls.org/
[373] http://www.tango-controls.org/

Extending pool features

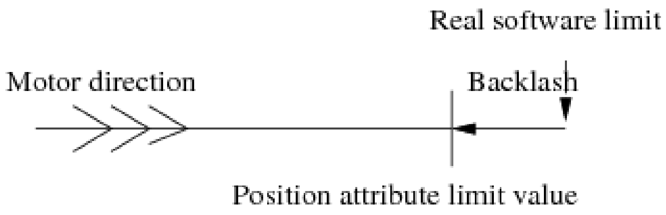
To be filled in

Common task handled by the pool

Constraint

Two types of constraint are identified.

1. Simple constraint: This type of constraint is valid only for motor motion. It limits motor motion. This in not the limit switches which are a hardware protection. It's a software limit. This type of constraint is managed by the min_value and max_value property of the motor Position Tango[374] attribute. Tango[375] core will refused to write the attribute (Position) if outside the limits set by these min_value and max_value attribute properties. These values are set on motor Position attribute in physical unit. **Warning** : The backlash has to be taken into account in the management of this limit. In order to finish the motion always coming from the same direction, sometimes the motor has to go a little bit after the wanted position and then returns to the desired position. The limit value has to take the backlash value into account. If the motor backlash attribute is modified, it will also change the Position limit value.



2. User constraint: This kind of constraint is given to the user to allow him to write constraint macros which will be executed to allow or disallow an action to be done on one object. In the pool case, the object is a writable attribute and the action is writing the attribute. Therefore, the following algorithm is used when writing an attribute with constraint:

```
/IF/ Simple constraint set
   /IF/ New value outside limits
       - Throw an exception
   /ENDIF/
/ENDIF/

/IF/ Some user constraint associated to this attribute
   /FOR/ All the user constraint
      - Evaluate the constraint
      /IF/ The constraint evaluates to False
          - Throw an exception
      /ENDIF/
   /ENDFOR/
/ENDIF/

- Write the attribute
```

The first part of this algorithm is part of the Tango[376] core. The second part will be coded in the Pool

[374] http://www.tango-controls.org/
[375] http://www.tango-controls.org/
[376] http://www.tango-controls.org/

Tango[377] classes and in a first phase will be available only for the Position attribute of the Motor class.

User constraint implementation

When the user creates a constraint, he has to provide to the pool the following information:

1. The name of the object to which the constraint belongs. It is the name of the writable Tango[378] attribute (actually only a motor position attribute.

A user constraint will be written using the Python language. It has to be a Python class with a constructor and a "Evaluate" method. This class has to inherit from a class called PoolConstraint. This will allow the pool software to dynamically discover that this class is a pool constraint. The class may define the depending attributes/devices. A depending attribute/device is an object used to evaluate if the constraint is true or false. The depending attributes have to be defined in a list called **depending_attr_list** . Each element in this list is a dictionnary with up to 2 elements which are the description of the depending attribute and eventually a default value. The depending devices have to be defined in a list called **depending_dev_list** which follow the same syntax than the depending_attr_list. A constraint may also have properties as defined in XXX: Unknown inset LatexCommand ref{par:Controller-properties}: . The constructor will receive three input arguments which are:

1. A list with the depending attribute name
2. A list with the depending device name
3. A dictionnary (name:value) with the properties definition

One rule of the constructor is to build the connection with these Tango[379] objects and to keep them in the instance. The Evaluate method will evaluate the constraint and will return true or false. It receives as input argument a list with the result of a read_attribute call executed on all the depending attributes.

Five pool device commands and two attribute allow the management of these constraints. The commands are **CreateConstraint** , **DeleteConstraint** , **EvaluateContraint, GetConstraintClassInfo** and **GetConstraint** . The attributes are called **ConstraintList** and **ConstraintClassList** . They are all detailed in chapters XXX: Unknown inset LatexCommand ref{sub:Device-pool-commands}: and XXX: Unknown inset LatexCommand ref{sub:Device-pool-attributes}: . The following is an example of a user constraint

```python
import PyTango

class MyConstraint(PoolConstraint):

   depending_attr_list = [{'DefaultValue':"first_mot/position",
                           'Description':"X position"},
                          {'DefaultValue':"second_mot/position",
                           'Description':"Z position"},
                          {'DefaultValue':"first_mot/velocity",
                           'Description':"X position speed"}]

   depending_dev_list = [{'DefaultValue':"first_dev",
                          'Description':"Air pressure device"}]

   inst_prop = {'MyProp':{'Type':PyTango.DevLong,'Description':'The psi constant',
                          'DefaultValue',1234}}

    def __init__(self,attr_list,dev_list,prop_dict)
```

(continues on next page)

[377] http://www.tango-controls.org/
[378] http://www.tango-controls.org/
[379] http://www.tango-controls.org/

(continued from previous page)

```
18        self.air_device = PyTango.DeviceProxy(dev_list[0])
19        self.const = prop_dict["MyProp"]
20
21    def Evaluate(self,att_value):
22       if att_value[0].value > (xxx * self.const)
23          return False
24       elif att_value[1].value > yyy
25          return False
26       elif att_value[2].value > zzz
27          return False
28       elif self.air_device.state() == PyTango.FAULT
29          return False
30       return True
```

Line 3 : The class inherits from the PoolConstraint class Line 5-10: Definition of the depending attributes Line 11-12: Definition of the depending devices Line 14-15: Definition of a constraint property Line 17-19: The constructor Line 21-30: The Evaluate method

Archiving motor position

XXX: Unknown inset LatexCommand label{sub:Archiving-motor-position}:

It is not possible to archive motor position using the Tango[380] memorized attribute feature because Tango[381] writes the attribute value into the database just after it has been set by the user. In case of motors which need some time to go to the desired value and which from time to time do not go exactly to the desired value (for always possible to have position which is a integer number of motor steps), it is more suited to store the motor position at the end of the motion. To achieve this, the pool has a command (called **ArchieveMotorPosition**) which will store new motor positions into the database. This command will be polled by the classical Tango[382] polling thread in order to execute it regularly. The algorithm used by this command is the following:

```
- Read motors position for all motors which are not actually moving

- /FOR/ all motors
    - /IF/ The new position just read is different than the old one
       - Mark the motor as storable
    - /ENDIF/
- /ENDFOR/

- Store in DB position of all storable motors
- Memorize motors position
```

In order to minimize the number of calls done on the Tango[383] database, we need to add to the Tango[384] database software the ability to store x properties of one attribute of y devices into the database in one call (or may be simply the same property of one attribute of several device).

Scanning

To be filled in

[380] http://www.tango-controls.org/
[381] http://www.tango-controls.org/
[382] http://www.tango-controls.org/
[383] http://www.tango-controls.org/
[384] http://www.tango-controls.org/

Experiment management

To be filled in

The pool device Tango[385] interface

The pool is implemented as a C++ Tango[386] device server and therefore supports a set of commands/attributes. It has several attributes to get object (motor, pseudo-motor, controller) list. These lists are managed as attributes in order to have events on them when a new object (motor, controller. . .) is created/deleted.

Device pool commands

XXX: Unknown inset LatexCommand label{sub:Device-pool-commands}:

On top of the three classical Tango[387] commands (State, Status and Init), the pool device supports the commands summarized in the following table:

[385] http://www.tango-controls.org/
[386] http://www.tango-controls.org/
[387] http://www.tango-controls.org/

Device type	Name	Input data type	Output data type
related	InitController	Tango::DevString	void
com- mands	ReloadControllerCode SendToController	Tango::DevString Tango::DevVarStringArray	void Tango::DevString
Motor	CreateMotor	Tango::DevVarLongStringArray	void
related com- mands	DeleteMotor	Tango::DevString	void
Motor group	CreateMotorGroup	Tango::DevVarStringArray	void
related com- mands	DeleteMotorGroup GetPseudoMotorInfo	Tango::DevString Tango::DevVarStringArray	void Tango::DevVarStringArray
Pseudo motor	CreatePseudoMotor	Tango::DevVarStringArray	void
related com- mands	DeletePseudoMotor ReloadPseudoMotorCode GetConstraintClassInfo CreateConstraint	Tango::DevString Tango::DevString Tango::DevString Tango::DevVarStringArray	void void Tango::DevVarStringArray void
User Con- straint	DeleteConstraint	Tango::DevString	void
related	EvaluateConstraint	Tango::DevString	Tango::DevBoolean
com- mands	GetConstraint ReloadConstraintCode	Tango::DevString Tango::DevString	Tango::DevVarLongArray void
Exper- iment Channel	CreateExpChannel	Tango::DevVarStringArray	void
related com- mands	DeleteExpChannel	Tango::DevString	void
Mea- sure- ment group	CreateMeasurementGroup	Tango::DevVarStringArray	void
related com- mands	DeleteMeasurementGroup	Tango::DevString	void
Dyn loaded Tango	LoadTangoClass		
class re- lated	UnloadTangoClass		
com- mands	ReloadTangoClass		
Dyn. created	CreateXXX		
com- mands	DeleteXXX		
Miscel- laneous	ArchiveMotorPosition	void	void

- **CreateController** : This command creates a controller object. It has four arguments (all strings) which are:
  1. The controller device type: Actually three types are supported as device type. They are:
     - "Motor" (case independent) for motor device
     - "CounterTimer" (case independent) for counter timer device
     - "ZeroDExpChannel" (case independent) for zero dimension experiment channel device
  2. Controller code file name: For C++ controller, this is the name of the controller shared library file. For Python controller, this is the name of the controller module. This parameter is only a file name, not a path. The path is automatically taken from the pool device **PooPath** property. It is not necessary to change your LD_LIBRARY_PATH or PYTHONPATH environment variable. Everything is taken from the PoolPath property.
  3. Controller class name: This is the name of the class implementing the controller. This class has to be implemented within the controller shared library or Python module passed as previous argument
  4. Instance name: It is a string which allows the device pool to deal with several instance of the same controller class. The pool checks that this name is uniq within a control system.

The list of created controllers is kept in one of the pool device property and at next startup time, all controllers will be automatically re-created. If you have several pool device within a control system (the same TANGO_HOST), it is not possible to have two times the same controller defines on different pool device. Even if the full controller name is <Controller file name>.<Controller class name>/<Instance name>, each created controller has an associated name which is:

> <Instance name>

which has to be used when the controller name is requested. This name is case independent.

- **DeleteController** : This command has only one input argument which is the controller name (as defined previously). It is not possible to delete a controller with attached device(s). You first have to delete controller's device(s).
- **InitController** : This command is used to (re)-initialize a controller if the controller initialization done at pool startup time has failed. At startup time, the device pool creates controller devices even if the controller initialization has failed. All controller devices are set to the FAULT state. This command will try to re-create the controller object and if successful, send an "Init" command to every controller devices. Its input argument is the controller name.
- **GetControllerInfo** : This command has three or four input parameters which are: XXX: Unknown inset LatexCommand label{ite:GetControllerInfo:}:
  1. The controller device type
  2. The controller code file name: For C++ controller, this is the name of the controller shared library file. For Python controller, this is the name of the controller module. This parameter is only a file name, not a path. The path is automatically taken from the pool device **PooPath** property.
  3. The controller class name: This is the name of the class implementing the controller. This class has to be implemented within the controller shared library or Python module passed as previous argument
  4. The controller instance name: This parameter is optional. If you do not specify it, the command will return information concerning controller properties as defined at the class level. If you defined it, the command will return information concerning controller properties for this specific controller instance.

It returns to the caller all the informations related to controller properties as defined in the controller code and/or in the Tango database. The following format is used to return these informations:

1. The string describing the controller (or an empty string if not defined)
2. Number of controller properties
3. For each property:
   (a) The property name
   (b) The property data type
   (c) The property description
   (d) The property default value (Empty string if not defined)

- **ReloadControllerCode** : The controller code is contains in a shared library dynamically loaded or in a Python module. The aim of this command is to unlink the pool to the shared library and to reload it (or Reload the Python module). The command argument is a string which is the controller file name as defined for the CreateController command. For motor controller, it is not possible to do this command if one of the motor attached to controller(s) using the code within the file is actually moving. All motor(s) attached to every controller(s) using this file is switched to FAULT state during this command execution. Once the code is reloaded, an "Init" command is sent to every controller devices.
- **SendToController** : Send data to a controller. The first element of the input argument array is the controller name. The second one is the string to be sent to the controller. This command returns the controller answer or an empty string is the controller does not have answer.
- **CreateMotor** : This command creates a new motor. It has three arguments which are:
  1. The motor name (a string). This is a Tango[388] device alias. It is not allowed to have '/' character within this name. It is a case independent name.
  2. The motor controller name (a string)
  3. The axe number within the controller

The motor is created as a Tango[389] device and automatically registered in the database. At next startup time, all motors will be automatically re-created. A Tango[390] name is assigned to every motor. This name is a Tango[391] device name (3 fields) and follow the syntax:

motor/controller_instance_name/axe_number

in lower case letters.

- **DeleteMotor** : This command has only one argument which is the motor name as given in the first argument of the CreateMotor command. The device is automatically unregistered from the Tango[392] database and is not accessible any more even for client already connected to it.
- **CreateMotorGroup** : This command creates a new motor group. It has N arguments which are:
  1. The motor group name (a string). This is a Tango[393] device alias. It is not allowed to have '/' character within this name. It is a case independent name.
  2. The list of motor element of the group (motor name or another group name or pseudo-motor name)

[388] http://www.tango-controls.org/
[389] http://www.tango-controls.org/
[390] http://www.tango-controls.org/
[391] http://www.tango-controls.org/
[392] http://www.tango-controls.org/
[393] http://www.tango-controls.org/

The motor group is created as a Tango[394] device and automatically registered in the database. At next startup time, all motor groups will be automatically re-created. A Tango[395] name is assigned to every motor group. This name is a Tango[396] device name (3 fields) and follow the syntax:

mg/ds_instance_name/motor_group_name

in lower case letters.

- **DeleteMotorGroup** : This command has only one argument which is the motor group name as given in the first argument of the CreateMotorGroup command. The device is automatically unregistered from the Tango[397] database and is not accessible any more even for client already connected to it. This command is not allowed if another motor group is using the motor group to be deleted.
- **GetPseudoMotorInfo** : XXX: Unknown inset LatexCommand label{sub:GetPseudoMotorClassInfo}: : This command has one input argument (a string):

  <module_name>.<class_name>

  The command returns a list of strings representing the pseudo motor system information with the following meaning:

  pseudo_info[0] - textual description of the pseudo motor class.

  pseudo_info[1] - (=M) the number of motors required by this pseudo motor class.

  pseudo_info[2] - (=N) the number of pseudo motors that the pseudo motor system aggregates.

  pseudo_info[3] - the number of parameters required by the pseudo motor system.

  pseudo_info[4..N+4] - the textual description of the roles of the N motors.

  pseudo_info[N+5..N+M+5] - the textual description of the roles of the M pseudo motors.

  pseudo_info[N+M+6..N+M+P+6] - the textual description of the P parameters.

example :

```
GetPseudoMotorInfo('PseudoLib.Slit')

could have as a return:
```

```
["A Slit system for controlling gap and offset pseudo motors.",
"2",
"2",
"0",
"Motor on blade 1",
"Motor on blade 2",
"Gap",
"Offset"]
```

- **CreatePseudoMotor** :This command has a variable number of input arguments (all strings):

  1. the python file which contains the pseudo motor python code.
  2. the class name representing the pseudo motor system.
  3. the N pseudo motor names. These will be the pseudo motor alias for the corresponding pseudo motor tango devices.

[394] http://www.tango-controls.org/
[395] http://www.tango-controls.org/
[396] http://www.tango-controls.org/
[397] http://www.tango-controls.org/

4. the M motor names. These names are the existing tango motor alias.

N and M must conform to the class name information. See XXX: Unknown inset LatexCommand ref{sub:GetPseudoMotorClassInfo}: to find how to get class information.

For each given pseudo motor name a Tango[398] pseudo motor device is created and automatically registered in the database. At next startup time, all pseudo motors will be automatically re- created. A Tango[399] name is assigned to every pseudo motor. This name is a Tango[400] device name (3 fields) and follow the syntax:

pm/python_module_name.class_name/pseudo_motor_name

For each Tango[401] pseudo motor device the device pool will also create a corresponding alias named pseudo_motor_name.

If a motor group Tango[402] device with the given motor names doesn't exist then the device pool will also create a motor group with the following name:

mg/tango_device_server_instance_name/_pm_<internal motor group number>

This motor group is built for internal Pool usage. It is not intended that the pseudo motor is accessed directly through this motor group. However, if needed elsewhere, it can be accessed as the usual motor group without any special restrictions.

example:

CreatePseudoMotor('PseudoLib.py','Slit','gap01','offset01','blade01',' blade02')

- **DeletePseudoMotor** : This command has only one argument which is the pseudo motor identifier. The device is automatically unregistered from the Tango database and is not accessible any more even for client already connected to it. This command is not allowed if a motor group is using the pseudo motor to be deleted.
- **ReloadPseudoMotorCode** :The calculation code is contains in a dynamically loaded Python module. The aim of this command is to reload the Python module. The command argument is a string which is the python module as defined for the CreatePseudoMotor and GetPseudoMotorInfo commands. It is not possible to do this command if one of the motor attached to pseudo motor system(s) using code within the file is actually moving. All pseudo motor(s) using this file are switched to FAULT state during this command execution.
- **CreateExpChannel** : This command creates a new experiment channel. It has three arguments which are:
  1. The experiment channel name (a string). This is a Tango[403] device alias. It is not allowed to have '/' character within this name. It is a case independent name.
  2. The experiment channel controller name (a string)
  3. The index number within the controller

The experiment channel is created as a Tango[404] device and automatically registered in the database. At next startup time, all created experiment channels will be automatically re-created. A Tango[405] name is assigned to every experiment channel. This name is a Tango[406] device name (3 fields) and follow the syntax:

[398] http://www.tango-controls.org/
[399] http://www.tango-controls.org/
[400] http://www.tango-controls.org/
[401] http://www.tango-controls.org/
[402] http://www.tango-controls.org/
[403] http://www.tango-controls.org/
[404] http://www.tango-controls.org/
[405] http://www.tango-controls.org/
[406] http://www.tango-controls.org/

expchan/controller_instance_name/index_number

in lower case letters. The precise type of the experiment channel (Counter/Timer, ZeroD, OneD…) is retrieved by the pool device from the controller given as command second parameter.

- **DeleteExpChannel** : This command has only one argument which is the experiment channel name as given in the first argument of the CreateExpChannel command. The device is automatically unregistered from the Tango[407] database and is not accessible any more even for client already connected to it.
- **GetConstraintClassInfo** : This command has one input parameter (a string) which is the constraint class name. It returns to the caller all the information related to constraint dependencies and to constraint properties as defined in the constraint code. The following format is used to return properties:
  - Depending attributes number
    - Depending attribute name
    - Depending attribute description
  - Depending devices number
    - Depending device name
    - Depending device description
  - Class property number
    - Class property name
    - Class property description
    - Class property default value (Set to "NotDef" if not defined)
  - Instance property number
    - Instance property name
    - Instance property description
    - Instance property default value (Set to "NotDef" if not defined)
- **CreateMeasurementGroup** : This command creates a new measurement group. It has N arguments which are:
  1. The measurement group name (a string). This is a Tango[408] device alias. It is not allowed to have '/' character within this name. It is a case independent name.
  2. The list of channel elements of the group (Counter/Timer, 0D, 1D or 2D experiment channel)

The measurement group is created as a Tango[409] device and automatically registered in the database. At next startup time, all measurement groups will be automatically re-created. A Tango[410] name is assigned to every measurement group. This name is a Tango[411] device name (3 fields) and follow the syntax:

mntgrp/ds_instance_name/measurement_group_name

in lower case letters.

[407] http://www.tango-controls.org/
[408] http://www.tango-controls.org/
[409] http://www.tango-controls.org/
[410] http://www.tango-controls.org/
[411] http://www.tango-controls.org/

- **DeleteMeasurementGroup** : This command has only one argument which is the measurement group name as given in the first argument of the CreateMeasurementGroup command. The device is automatically unregistered from the Tango database and is not accessible any more even for client already connected to it.
- **AddConstraint** : This command creates a user constraint object. It has several arguments (all strings) which are:
  1. Constraint code file name: The name of the constraint module. This parameter is only a file name, not a path. The path is automatically taken from the pool PooPath property.
  2. Constraint class name: This is the name of the class implementing the controller. This class has to be implemented within the controller shared library or Python module passed as previous argument
  3. Instance name: It is a string which allows the device pool to deal with several instance of the same controller class.
  4. The object to which the constraint belongs. It has to be a writable attribute name (actually only a motor position)
  5. The list of depending objects. (Variable length list which may be empty)

The list of created constraints is kept in one of the pool device property and at next startup time, all constraints will be automatically re-created. It is possible to create several constraint on the same object. They will be executed in the order of their creation. Each created constraint has a associated name which is:

<Constraint class name>/<Instance name>

- **DeleteConstraint** : This command has only one argument which is the constraint name as define previously.
- **EvaluateConstraint** : This command has only one argument which is the constraint name. It runs the "evaluate" method of the constraint and sends the return value to the caller
- **GetConstraint** : The input parameter of this command is the name of a Tango[412] object. Actually, it has to be the name of one of the motor Position attribute. The command returns the list of Constraint ID attached to this object.
- **ReloadConstraintCode** : The constraint code is contains in a Python module. The aim of this command is to reload the Python module. The command argument is a string which is the constraint file name as defined for theAddConstraint command. All object(s) using this constraint are switched to FAULT state during this command execution.
- **LoadTangoClass** :
- **UnloadTangoClass** :
- **ReloadTangoClass** :
- **CreateXXX** :
- **DeleteXXX:**
- **ArchiveMotorPosition** : Send new motor(s) position to the database. This command will be polled with a default polling period of 10 seconds.

The classical Tango[413] **Init** command destroys all constructed controller(s) and re-create them reloading their code. Then, it sends an "Init" command to every controlled objects (motor, pseudo-motor and motor group) belonging to the pool device. Motor(s) are switched to FAULT state when controller are destroyed.

[412] http://www.tango-controls.org/
[413] http://www.tango-controls.org/

The pool device knows only two states which are ON and ALARM. The pool device is in ALARM state if one of its controller failed during its initialization phase. It is in ON state when all controllers are correctly constructed. In case the pool device in in ALARM state, its status indicates which controller is faulty.

Device pool attributes

XXX: Unknown inset LatexCommand label{sub:Device-pool-attributes}:

The device pool supports the following attributes:

Name	Data type	Data format	Writable
ControllerList	Tango::DevString	Spectrum	R
ControllerClassList	Tango::DevString	Spectrum	R
MotorList	Tango::DevString	Spectrum	R
MotorGroupList	Tango::DevString	Spectrum	R
PseudoMotorList	Tango::DevString	Spectrum	R
PseudoMotorClassList	Tango::DevString	Spectrum	R
ExpChannelList	Tango::DevString	Spectrum	R
MeasurementGroupList	Tango::DevString	Spectrum	R
ConstraintList	Tango::DevString	Spectrum	R
ConstraintClassList	Tango::DevString	Spectrum	R
SimulationMode	Tango::DevBoolean	Scalar	R/W
XXXList	Tango::DevString	Spectrum	R

- **ControllerList** : This is a read only spectrum string attribute. Each spectrum element is the name of one controller following the syntax:

  <instance_name> - <Ctrl file>.<controller_class_name/instance_name> - <Device type> <Controller language> Ctrl (<Ctrl file>)

- **ControllerClassList** : This is a read only spectrum string attribute. Each spectrum element is the name of one of the available controller class that the user can create. To build this list, the pool device server is using a property called **PoolPath** which defines the path where all files containing controller code should be (Python and C++ controllers). The syntax used for this PoolPath property is similar to the syntax used for Unix PATH environment variable (list of absolute path separated by the ":" character). Each returned string has the following syntax:

  Type: <Ctrl dev type> - Class: <Ctrl class name> - File: <Abs ctrl file path>

- **MotorList** : This is a read only spectrum string attribute. Each spectrum element is the name of one motor known by this pool. The syntax is:

  <Motor name> (<Motor tango name>)

- **MotorGroupList** : This is a read only spectrum string attribute. Each spectrum element is the name of one motor group known by this pool. The syntax is:

  <Motor group name> (<Motor group tango name>) Motor list: <List of group members> (<List of physical motors in the group>)

  The last information is displayed only if the physical group structure differs from the logical one (pseudo-motor or other group used as group member)

- **PseudoMotorList** :This is a read only spectrum string attribute. Each spectrum element is the name of one motor known by this pool. The syntax is:

  <pseudo motor name> (<pseudo motor tango name>) Motor List: <motor name>1,…,<motor name>M

- **ExpChannelList** : This is a read only spectrum string attribute. Each spectrum element is the name of one experiment channel known by this pool. The syntax is:

  <Exp Channel name> (<Channel tango name>) <Experiment channel type>

  The string describing the experiment channel type may be:

  - Counter/Timer Experiment Channel
  - Zero D Experiment Channel

- **MeasurementGroupList** : This is a read only spectrum string attribute. Each spectrum element is the name of one measurement group known by the pool. The syntax is:

  > <Measurement group name> (<Measurement group tango name>) Experiment Channel list: <List of group members>

- **PseudoMotorClassList** :This is a read only spectrum string attribute. Each spectrum element is the name of a valid Pseudo python system class. The syntax is:

  > <python module name>.<python class name>

  . The python files to be found depend on the current value of the pool path. See XXX: Unknown inset LatexCommand ref{sub:PoolPath}:

- **ConstraintClassList** : This is a read only spectrum string attribute. Each spectrum element is the name of one of the available constraint class that the user can create. To build this list, the pool device server is using a property called **PoolPath** which defines the path where all files containing constraint code should be. The syntax used for this property is similar to the syntax used for Unix PATH environment variable (list of absolute path separated by the ":" character). To find constraint classes, the pool will look into all Python files (those with a .py suffix) for classes definition which inherit from a base class called **PoolConstraint** .

- **ConstraintList** : This is a read only spectrum string attribute. each spectrum element is one of the constraint actually registered in the pool. The syntax of each string is:

  > <Constraint class name/instance name> - <associated to> - <depending on attribute(s) - <depending on device(s)>

- **SimulationMode** : This is a read-write scalar boolean attribute. If set to true, all the pool device(s) are switched to Simulation mode. This means that all commands received by pool device(s) will not be forwarded to the associated controllers.

- **XXXList** :

Device pool property

The pool device supports the following property:

Property name	Property data type	Default value
PoolPath	String	
DefaultMotPos_AbsChange	Double	5
DefaultMotGrpPos_AbsChange	Double	5
DefaultCtVal_AbsChange	Double	5
DefaultZeroDVal_AbsChange	Double	5
DefaultCtGrpVal_AbsChange	Double	5
DefaultZeroDGrpVal_AbsChange	Double	5
GhostGroup_PollingPeriod	String	5000
MotThreadLoop_SleepTime	Long	10
NbStatePerRead	Long	10
ZeroDNbReadPerEvent	Long	5

- **PoolPath** : XXX: Unknown inset LatexCommand label{sub:PoolPath}: The path (same syntax than the Unix PATH environment variable) where the pool software is able to locate Controller software, Pseudo-motor software or Constraint software for both Python or C++ languages
- **DefaultMotPos_AbsChange** : The default value used to trigger change event when the position attribute is changing (the associated motor is moving). This property has a hard-coded default value set to 5
- **DefaultMotGrpPos_AbsChange** : The default value used to trigger change event when the group device position attribute is changing. This property has a hard-coded default value set to 5
- **DefaultCtVal_AbsChange** : The default value used to trigger change event when the counter/timer attribute is changing (the counter is counting or the timer is timing). This property has a hard-coded default value set to 5
- **DefaultZeroDVal_AbsChange** : The default value used to trigger change event when the Zero Dimension Experiment Channel is acquiring data. This property has a hard-coded default value set to 5
- **DefaultCtGrpVal_AbsChange** : The default value used to trigger change event when the counter/timer attribute(s) of a measurement group is(are) changing (the counter is counting or the timer is timing). This property has a hard-coded default value set to 5
- **DefaultZeroDGrpVal_AbsChange** : The default value used to trigger change event when the Zero Dimension Experiment Channel(s) of a measurement group is(are) acquiring data. This property has a hard-coded default value set to 5
- **GhostGroup_PollingPeriod** : The ghost motor/measurement group polling period in mS. This property has a default value of 5000 (5 sec)
- **MotThreadLoop_SleepTime** : The time (in mS) during which the motion thread will sleep between two consecutive motor state request. The default value is 10
- **NbStatePerRead** : The number of motor state request between each position attribute reading done by the motion thread. The default value is 10. This means that during a motion, the motor position is read by the thread every 100 mS (10 * 10)
- **ZeroDNbReadPerEvent** : The number of times the Zero D Experiment Channel value is read by the acquisition thread between firing a change event. The event will be effectively fired to the interested clients according to the CumulatedValue attribute "Absolute Change" property value.
- **Controller** : An internally managed property which allow the pool device to remember which controller has been created.

Creating device

This chapter gives details on what has to be done to create device using the device pool in order to check the work to be done by a Sardana configuration tool.

Creating motor

The following is the action list which has to be done when you want to create a new motor:

1. Display the list of all the controller the pool already has.
2. Select one of this controller
3. If the user selects a new controller
   (a) Read the attribute ControllerClassList to get the list of Controller installed in your system
   (b) Select one of the controller class
   (c) With the GetControllerInfo command, get the list of controller properties
   (d) Give a controller instance name
   (e) Display and eventually change the controller properties (if any)
   (f) Create the controller object using the CreateController pool command
4. Give a motor name and a motor axis number in the selected controller
5. Create the motor with the CreateMotor pool command
6. Read the attribute list of the newly created motor
7. Display and eventually change the motor attributes related to motor features and eventually to extra-features

Creating motor group

The following is the action list which has to be done when creating a motor group

1. Give a name to the motor group
2. Display the list of all registered motors (attribute MotorList), all registered motor groups (attribute MotorGroupList), all registered pseudo motors (attribute PseudoMotorList) and select those which have to be member of the group.
3. Create the group (command CreateMotorGroup)

Creating a pseudo motor system

The following is the action list which has to be done when you want to create a new pseudo motor:

1. Display the list of all available pseudo motor system classes and select one of them
   (a) if there is no proper pseudo system class write one in Python
   (b) update the PoolPath Pool property if necessary
2. Get the selected pseudo motor system class information
3. Give names to the pseudo motors involved in the selected pseudo motor system

4. Create the motor(s) which are involved (if they have are not created yet: See XXX: Unknown inset LatexCommand ref{sub:Creating-motor}:) and assign the coresponding roles
5. Create the pseudo motor system (command CreatePseudoMotor)

Creating a user constraint

The following is the action list which has to be done when you want to create a new user constraint:

1. Display the list of all the constraint the pool already has.
2. Select one of this constraint
3. If the user selects a new constraint
   (a) Read the attribute ConstraintClassList to get the list of Constraint installed in your system
   (b) Select one of the constraint class
   (c) With the GetConstraintClassInfo command, get the list of constraint dependencies and properties
   (d) Give a constraint instance name
   (e) If it is the first constraint of this class
      i. Display and eventually change the constraint class properties (if any)
4. Display and eventually change the constraint depending attribute (if any)
5. Display and eventually change the constraint depending device (if any)
6. Display and eventually change the constraint instance properties (if any)
7. Create the constraint object using the CreateConstraint pool command

Some words on internal implementation

This chapter gives some details on some part of the pool implementation in order to clarify reader ideas

Moving motor

Moving a motor means writing its Position attribute. In Tango, it is already splitted in two actions which are:

1. Call a Motor class method called "is_allowed"
2. Call a Motor class method called "write_Position"

The second method will be executed only if the first one returns true. The move order is sent to the motor (via the controller) in the code of the second method.

The is_allowed method

The code implemented in this method follow the algorithm:

```
- /IF/ There are any Pseudo Motor using the motor
   - /FOR/ All these Pseudo Motors
      - /IF/ They have some limits defined
         - Compute new Pseudo Motor position if motor moved to the desired value
         - /IF/ The computed value is outside the authorized window
             - Return False
         - /ENDIF/
      - /ENDIF/
   - /ENDFOR/
- /ENDIF/

- /IF/ There are some user constraint attached to the motor
  - /FOR/ Each user constraint
      - /IF/ The constraint has some depending attribute(s)
         - Read these attributes
      - /ENDIF/
      - /IF/ If the execution of the contraint "Evaluate" method returns False
         - Return False
      - /ENDIF/
  - /ENDFOR/
- /ENDIF/

- Return True
```

The write_Position method

The code implemented in this method follows the algorithm:

```
- Compute the dial position from the user position
- /IF/ A backlash is defined for this motor and the controller does not manage it
    - Update motor desired position according to motion direction and backlash value
- /ENDIF/
- Start a thread sending it which motor has to move to which position
- Wait for thread acknowledge
- Return to caller
```

The motion thread will execute the following algorithm:

```
- /FOR/ Each controller(s) implied in the motion
     - Lock the controller object
     - Call PreStartAll()
- /ENDFOR/

- /FOR/ Each motor(s) implied in the motion
     - ret = PreStartOne(motor to move, new position)
     - /IF/ ret is true
          - Call StartOne(motor to move, new position)
     - /ELSE/
          - Inform write_Position that an error occurs
          - Send acknowledge to write_Position method
     - /ENDIF/
- /ENDFOR/

- /FOR/ Each motor(s) implied in the motion
     - Set motor state to MOVING and send a Tango_ event to the requesting client
```

(continues on next page)

(continued from previous page)

```
- /ENDFOR/

- /FOR/ Each controller(s) implied in the motion
     - Call StartAll()
     - Unlock the controller object
- /ENDFOR/

- Send acknowledge to the write_Position method

- /WHILE/ One of the motor state is MOVING (From controller)
     - Sleep for 10 mS

     - /IF/ One of the motor implied in the motion is not moving any more
        - /IF/ This motor has backlash and the motion is in the "wrong" direction
            - Ask for a backlash motion in the other direction
              (Easy to write, not as easy to do...)
        - /ENDIF/
        - Send a Tango_ event on the state attribute to the requesting client
        - Leave the loop
     - /ENDIF/

     - /IF/ it is time to read the motor position
        - Read the motor position
        - Send a change event on the Position attribute to the requested client if
          the change event criterion is true
     - /ENDIF/
- /ENDWHILE/

- Sleep for the time defined by the motor (group) Sleep_bef_last_read property
- Read the motor position
- Send a forced change event on the Position attribute to the requesting client
  with the value set to the one just read
```

Data acquisition

Data aquisition is triggered by invoking a Start command on the measurement group. The code implemented implements the following algorithm.

```
/IF/ in timer mode
    - Write CumulationTime on all 0D channels with Integration_time value
/ELIF/ in monitor mode
    - Write CumulationTime on all 0D channels with 0(zero) value
/ENDIF/

/FOR/ Each 0D channel implied in the data aquisition
    - Load configuration
/END FOR/

- Start a CounterTimer thread with channels involved, master channel and the proper␣
↪value to be set on it
- Wait for CounterTimer thread acknowledge

/FOR/ Each 0D channel implied in the data aquisition
    - Send Start command
/END FOR/
```

(continues on next page)

(continued from previous page)

```
- Return to caller
```

The Counter/Timer thread will execute the following algorithm:

```
- Calculate the list of controllers involved and determine which controller has the␣
↪master channel
/FOR/ Each channel(s) implied in the data aquisition
    - Lock the channel object
/END FOR/

/FOR/ Each controller(s) implied in the data acquisition
    - Lock the controller object
/END FOR/

/FOR/ Each channel(s) implied in the data acquisition
    - Load configuration
/END FOR/

- Load the master channel - timer(monitor) - with the integration time(count)

/FOR/ Each controller(s) implied in the data acquisition
    - Call PreStartAllCT()
/END FOR/

/FOR/ Each channel(s), except for the master channel, implied in the data acquisition,
    - Call PreStartOneCT(channel)
    - Call StartOneCT(channel)
/END FOR/

/FOR/ Each controller(s) implied in the data aquisition
    - Call StartAllCT()
/END FOR/

- Call PreStartAllCT() on the controller which contains the master channel
- Call PreStartOneCT(master channel)
- Call StartOneCT(master channel)
- Call StartAllCT() on the controller which contains the master channel

/FOR/ Each controller(s) implied in the data aquisition
    - Unlock the controller object
/END FOR/

/FOR/ Each channel(s) implied in the data aquisition
    - Unlock the channel object
/END FOR/

- Send acknowledge to the Start method

/WHILE/ master channel state is MOVING (From controller)
     - Sleep for 10 * sleepTime mS

     /IF/ If master channel is not moving any more
        - Stop all channels
        - Send a Tango event on the state attribute to the requesting client
        - Leave the loop
```

(continues on next page)

(continued from previous page)

```
     /ENDIF/

     /IF/ it is time to read the channel values
        - Read the channel values
        - Send a change event on each value attribute to the requested client if
          the change event criterion is true
     /ENDIF/
/ENDWHILE/

- Read the channel values
- Send a forced change event on each value attribute to the requesting client
  with the value set to the one just read
```

Macro Server

Todo: document this chapter

Introduction

This paper describes the macro server Tango[414] *API*.

sardana

This package provides the sardana library

Packages

pool

This is the main device pool module

Modules

controller

This module contains the definition of the Controller base classes

Constants

Type = 'type'
Constant data type (to be used as a *key* in the definition of *axis_attributes* or *ctrl_attributes*)

[414] http://www.tango-controls.org/

Access = 'r/w type'
Constant data access (to be used as a *key* in the definition of `axis_attributes` or `ctrl_attributes`)

Description = 'description'
Constant description (to be used as a *key* in the definition of `axis_attributes` or `ctrl_attributes`)

DefaultValue = 'defaultvalue'
Constant default value (to be used as a *key* in the definition of `axis_attributes` or `ctrl_attributes`)

FGet = 'fget'
Constant for getter function (to be used as a *key* in the definition of `axis_attributes` or `ctrl_attributes`)

FSet = 'fset'
Constant for setter function (to be used as a *key* in the definition of `axis_attributes` or `ctrl_attributes`)

Memorize = 'memorized'
Constant memorize (to be used as a *key* in the definition of `axis_attributes` or `ctrl_attributes`) Possible values for this key are `Memorized`, `MemorizedNoInit` and `NotMemorized`

Memorized = 'true'
Constant memorized (to be used as a *value* in the `Memorize` field definition in `axis_attributes` or `ctrl_attributes`)

MemorizedNoInit = 'true_without_hard_applied'
Constant memorize but not write at initialization (to be used as a *value* in the `Memorize` field definition in `axis_attributes` or `ctrl_attributes`)

NotMemorized = 'false'
Constant not memorize (to be used as a *value* in the `Memorize` field definition in `axis_attributes` or `ctrl_attributes`)

MaxDimSize = 'maxdimsize'
Constant MaxDimSize (to be used as a *key* in the definition of `axis_attributes` or `ctrl_attributes`)

Interfaces

- `Readable`
- `Startable`
- `Stopable`
- `Loadable`
- `Synchronizer`

Classes

- `Controller`
- `PseudoController`
- `MotorController`
- `PseudoMotorController`
- `CounterTimerController`

- *ZeroDController*
- *OneDController*
- *TwoDController*
- *PseudoCounterController*
- *IORegisterController*

Readable interface

class Readable

Bases: object[415]

A Readable interface. A controller for which it's axis are 'readable' (like a motor, counter or 1D for example) should implement this interface

PreReadAll()

Controller API. Override if necessary. Called to prepare a read of the value of all axis. Default implementation does nothing.

PreReadOne(*axis*)

Controller API. Override if necessary. Called to prepare a read of the value of a single axis. Default implementation does nothing.

Parameters **axis** (*int*[416]) – axis number

ReadAll()

Controller API. Override if necessary. Called to read the value of all selected axis Default implementation does nothing.

ReadOne(*axis*)

Controller API. Override is MANDATORY! Default implementation raises NotImplementedError[417]

Parameters **axis** (*int*[418]) – axis number

Returns the axis value

Return type object[419]

[415] https://docs.python.org/dev/library/functions.html#object
[416] https://docs.python.org/dev/library/functions.html#int
[417] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[418] https://docs.python.org/dev/library/functions.html#int
[419] https://docs.python.org/dev/library/functions.html#object

Startable interface

class Startable

Bases: `object`[420]

A Startable interface. A controller for which it's axis are 'startable' (like a motor, for example) should implement this interface

PreStartAll()

Controller API. Override if necessary. Called to prepare a start of all axis (whatever pre-start means). Default implementation does nothing.

PreStartOne(*axis*, *value*)

Controller API. Override if necessary. Called to prepare a start of the given axis (whatever pre-start means). Default implementation returns True.

Parameters

- **axis** (*int*[421]) – axis number
- **value** (*float*[422]) – new value

Returns True means a successfull pre-start or False for a failure

Return type bool[423]

StartOne(*axis*, *value*)

Controller API. Override if necessary. Called to do a start of the given axis (whatever start means). Default implementation raises `NotImplementedError`[424]

Parameters

- **axis** (*int*[425]) – axis number
- **value** (*float*[426]) – new value

StartAll()

Controller API. Override is MANDATORY! Default implementation does nothing.

[420] https://docs.python.org/dev/library/functions.html#object
[421] https://docs.python.org/dev/library/functions.html#int
[422] https://docs.python.org/dev/library/functions.html#float
[423] https://docs.python.org/dev/library/functions.html#bool
[424] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[425] https://docs.python.org/dev/library/functions.html#int
[426] https://docs.python.org/dev/library/functions.html#float

Stopable interface

class Stopable

Bases: `object`[427]

A Stopable interface. A controller for which it's axis are 'stoppable' (like a motor, for example) should implement this interface

PreAbortAll()

Controller API. Override if necessary. Called to prepare a abort of all axis (whatever pre-abort means). Default implementation does nothing.

PreAbortOne(*axis*)

Controller API. Override if necessary. Called to prepare a abort of the given axis (whatever pre-abort means). Default implementation returns True.

Parameters **axis** (*int*[428]) – axis number

Returns True means a successfull pre-abort or False for a failure

Return type bool[429]

AbortOne(*axis*)

Controller API. Override is MANDATORY! Default implementation raises `NotImplementedError`[430]. Aborts one of the axis

Parameters **axis** (*int*[431]) – axis number

AbortAll()

Controller API. Override if necessary. Aborts all active axis of this controller. Default implementation does nothing.

PreStopAll()

Controller API. Override if necessary. Called to prepare a stop of all axis (whatever pre-stop means). Default implementation does nothing.

PreStopOne(*axis*)

Controller API. Override if necessary. Called to prepare a stop of the given axis (whatever pre-stop means). Default implementation returns True.

Parameters **axis** (*int*[432]) – axis number

Returns True means a successfull pre-stop or False for a failure

[427] https://docs.python.org/dev/library/functions.html#object
[428] https://docs.python.org/dev/library/functions.html#int
[429] https://docs.python.org/dev/library/functions.html#bool
[430] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[431] https://docs.python.org/dev/library/functions.html#int
[432] https://docs.python.org/dev/library/functions.html#int

Return type bool[433]

StopOne(*axis*)

Controller API. Override if necessary. Stops one of the axis. *This method is reserved for future implementation.* Default implementation calls `AbortOne()`.

Parameters **axis** (*int*[434]) – axis number

New in version 1.0.

StopAll()

Controller API. Override if necessary. Stops all active axis of this controller. Default implementation does nothing.

Loadable interface

class Loadable

Bases: `object`[435]

A Loadable interface. A controller for which it's axis are 'loadable' (like a counter, 1D or 2D for example) should implement this interface

PreLoadAll()

Controller API. Override if necessary. Called to prepare loading the integration time / monitor value. Default implementation does nothing.

PreLoadOne(*axis, value, repetitions*)

Controller API. Override if necessary. Called to prepare loading the master channel axis with the integration time / monitor value. Default implementation returns True.

Parameters

- **axis** (*int*[436]) – axis number
- **value** (*float*[437]) – integration time /monitor value
- **repetitions** (*int*[438]) – number of repetitions

Returns True means a successfull PreLoadOne or False for a failure

Return type bool[439]

[433] https://docs.python.org/dev/library/functions.html#bool
[434] https://docs.python.org/dev/library/functions.html#int
[435] https://docs.python.org/dev/library/functions.html#object
[436] https://docs.python.org/dev/library/functions.html#int
[437] https://docs.python.org/dev/library/functions.html#float
[438] https://docs.python.org/dev/library/functions.html#int
[439] https://docs.python.org/dev/library/functions.html#bool

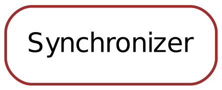
LoadAll()
Controller API. Override if necessary. Called to load the integration time / monitor value. Default implementation does nothing.

LoadOne(*axis, value, repetitions*)
Controller API. Override is MANDATORY! Called to load the integration time / monitor value. Default implementation raises `NotImplementedError`[440].

Parameters

- **axis** (*int*[441]) – axis number
- **value** (*float*[442]) – integration time /monitor value
- **repetitions** (*int*[443]) – number of repetitions
- **value** – integration time /monitor value

Synchronizer interface



class Synchronizer
Bases: `object`[444]

A Synchronizer interface. A controller for which its axis are 'Able to Synchronize' should implement this interface

PreSynchAll()
Controller API. Override if necessary. Called to prepare loading the synchronization description. Default implementation does nothing.

PreSynchOne(*axis, description*)
Controller API. Override if necessary. Called to prepare loading the axis with the synchronization description. Default implementation returns True.

Parameters

- **axis** (*int*[445]) – axis number
- **list<dict>** – synchronization description

Returns True means a successfull PreSynchOne or False for a failure

Return type bool[446]

[440] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[441] https://docs.python.org/dev/library/functions.html#int
[442] https://docs.python.org/dev/library/functions.html#float
[443] https://docs.python.org/dev/library/functions.html#int
[444] https://docs.python.org/dev/library/functions.html#object
[445] https://docs.python.org/dev/library/functions.html#int
[446] https://docs.python.org/dev/library/functions.html#bool

SynchAll()
: **Controller API**. Override if necessary. Called to load the synchronization description. Default implementation does nothing.

SynchOne(*axis*, *description*)
: **Controller API**. Override is MANDATORY! Called to load the axis with the synchronization description. Default implementation raises `NotImplementedError`[447].

Parameters

- **axis** (*int*[448]) – axis number
- **description** (*list<dict>*) – synchronization description

Abstract Controller

class Controller(*inst*, *props*, **args*, ***kwargs*)
Bases: `object`[449]

Base controller class. Do **NOT** inherit from this class directly

Parameters

- **inst** (*str*[450]) – controller instance name
- **props** (*dict*[451]) – a dictionary containning pairs of property name, property value
- **args** –
- **kwargs** –

class_prop = {}
: Deprecated since version 1.0.

use *ctrl_properties* instead

ctrl_features = []
: A sequence of `str`[452] representing the controller features

ctrl_extra_attributes = {}
: Deprecated since version 1.0.

use *axis_attributes* instead

[447] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[448] https://docs.python.org/dev/library/functions.html#int
[449] https://docs.python.org/dev/library/functions.html#object
[450] https://docs.python.org/dev/library/stdtypes.html#str
[451] https://docs.python.org/dev/library/stdtypes.html#dict
[452] https://docs.python.org/dev/library/stdtypes.html#str

ctrl_properties = {}

A dict[453] containing controller properties where:

- key : (str[454]) controller property name
- value : dict[455] with with three str[456] keys ("type", "description" and "defaultvalue" case insensitive):
  - for *Type*, value is one of the values described in *Data Type definition*
  - for *Description*, value is a str[457] description of the property. if is not given it defaults to empty string.
  - for *DefaultValue*, value is a python object or None if no default value exists for the property.

Example:

```
from sardana.pool.controller import MotorController, \
    Type, Description, DefaultValue

class MyCtrl(MotorController):

    ctrl_properties = \
    {
        'host' : { Type : str,
                   Description : "host name" },
        'port' : { Type : int,
                   Description : "port number",
                   DefaultValue: 5000 }
    }
```

ctrl_attributes = {}

A dict[458] containainning controller extra attributes where:

- key : (str[459]) controller attribute name
- value : dict[460] with str[461] possible keys: "type", "r/w type", "description", "fget", "fset" and "maxdimsize" (case insensitive):
  - for *Type*, value is one of the values described in *Data Type definition*
  - for *Access*, value is one of *DataAccess* ("read" or "read_write" (case insensitive) strings are also accepted) [default: ReadWrite]
  - for *Description*, value is a str[462] description of the attribute [default: "" (empty string)]
  - for *FGet*, value is a str[463] with the method name for the attribute getter [default: "get"<controller attribute name>]

[453] https://docs.python.org/dev/library/stdtypes.html#dict
[454] https://docs.python.org/dev/library/stdtypes.html#str
[455] https://docs.python.org/dev/library/stdtypes.html#dict
[456] https://docs.python.org/dev/library/stdtypes.html#str
[457] https://docs.python.org/dev/library/stdtypes.html#str
[458] https://docs.python.org/dev/library/stdtypes.html#dict
[459] https://docs.python.org/dev/library/stdtypes.html#str
[460] https://docs.python.org/dev/library/stdtypes.html#dict
[461] https://docs.python.org/dev/library/stdtypes.html#str
[462] https://docs.python.org/dev/library/stdtypes.html#str
[463] https://docs.python.org/dev/library/stdtypes.html#str

- for *FSet*, value is a str[464] with the method name for the attribute setter. [default, if *Access* = "read_write": "set"<controller attribute name>]
- for *DefaultValue*, value is a python object or None if no default value exists for the attribute. If given, the attribute is set when the controller is first created.
- for *Memorize*, value is a str[465] with possible values: *Memorized*, *MemorizedNoInit* and *NotMemorized* [default: *Memorized*]

  New in version 1.1.
- **for *MaxDimSize*, value is a tuple[466] with possible values:**
  - for scalar **must** be an empty tuple (() or []) [default: ()]
  - for 1D arrays a sequence with one value (example: (1024,)) [default: (2048,)]
  - for 1D arrays a sequence with two values (example: (1024, 1024)) [default: (2048, 2048)]

  New in version 1.1.

New in version 1.0.

Example:

```
from sardana.pool.controller import PseudoMotorController, \
    Type, Description, DefaultValue, DataAccess

class HKLCtrl(PseudoMotorController):

    ctrl_attributes = \
    {
        'ReflectionMatrix' : { Type : ( (float,), ),
                               Description : "The reflection matrix",
                               Access : DataAccess.ReadOnly,
                               FGet : 'getReflectionMatrix', },
    }

    def getReflectionMatrix(self):
        return ( (1.0, 0.0), (0.0, 1.0) )
```

axis_attributes = {}

A dict[467] containning controller extra attributes for each axis where:

- key : (str[468]) axis attribute name
- value : dict[469] with three str[470] keys ("type", "r/w type", "description" case insensitive):
  - for *Type*, value is one of the values described in *Data Type definition*
  - for *Access*, value is one of *DataAccess* ("read" or "read_write" (case insensitive) strings are also accepted)
  - for *Description*, value is a str[471] description of the attribute

[464] https://docs.python.org/dev/library/stdtypes.html#str
[465] https://docs.python.org/dev/library/stdtypes.html#str
[466] https://docs.python.org/dev/library/stdtypes.html#tuple
[467] https://docs.python.org/dev/library/stdtypes.html#dict
[468] https://docs.python.org/dev/library/stdtypes.html#str
[469] https://docs.python.org/dev/library/stdtypes.html#dict
[470] https://docs.python.org/dev/library/stdtypes.html#str
[471] https://docs.python.org/dev/library/stdtypes.html#str

- for `DefaultValue`, value is a python object or None if no default value exists for the attribute. If given, the attribute is set when the axis is first created.
- for `Memorize`, value is a `str`[472] with possible values: `Memorized`, `MemorizedNoInit` and `NotMemorized` [default: `Memorized`]

  New in version 1.1.
- **for `MaxDimSize`, value is a `tuple`[473] with possible values:**
  * for scalar **must** be an empty tuple (() or []) [default: ()]
  * for 1D arrays a sequence with one value (example: (1024,)) [default: (2048,)]
  * for 1D arrays a sequence with two values (example: (1024, 1024)) [default: (2048, 2048)]

  New in version 1.1.

New in version 1.0.

Example:

```
from sardana.pool.controller import MotorController, \
    Type, Description, DefaultValue, DataAccess

class MyMCtrl(MotorController):

    axis_attributes = \
    {
        'EncoderSource' : { Type : str,
                            Description : 'motor encoder source', },
    }

    def getAxisPar(self, axis, name):
        name = name.lower()
        if name == 'encodersource':
            return self._encodersource[axis]

    def setAxisPar(self, axis, name, value):
        name = name.lower()
        if name == 'encodersource':
            self._encodersource[axis] = value
```

standard_axis_attributes = {}
A `dict`[474] containing the standard attributes present on each axis device

gender = None
A `str`[475] representing the controller gender

model = 'Generic'
A `str`[476] representing the controller model name

organization = 'Sardana team'
A `str`[477] representing the controller organization

[472] https://docs.python.org/dev/library/stdtypes.html#str
[473] https://docs.python.org/dev/library/stdtypes.html#tuple
[474] https://docs.python.org/dev/library/stdtypes.html#dict
[475] https://docs.python.org/dev/library/stdtypes.html#str
[476] https://docs.python.org/dev/library/stdtypes.html#str
[477] https://docs.python.org/dev/library/stdtypes.html#str

image = None
A `str`[478] containning the path to the image file

logo = None
A `str`[479] containning the path to the image logo file

_findAPIVersion()
Internal. By default return the Pool Controller API version of the pool where the controller is running

_getPoolController()
Internal.

AddDevice(*axis*)
Controller API. Override if necessary. Default implementation does nothing.

Parameters **axis** (*int*[480]) – axis number

DeleteDevice(*axis*)
Controller API. Override if necessary. Default implementation does nothing.

Parameters **axis** (*int*[481]) – axis number

inst_name
Controller API. The controller instance name.

Deprecated since version 1.0: use *GetName()* instead

GetName()
Controller API. The controller instance name.

Returns the controller instance name

Return type str[482]

New in version 1.0.

GetAxisName(*axis*)
Controller API. The axis name.

Returns the axis name

Return type str[483]

New in version 1.0.

PreStateAll()
Controller API. Override if necessary. Called to prepare a read of the state of all axis. Default implementation does nothing.

PreStateOne(*axis*)
Controller API. Override if necessary. Called to prepare a read of the state of a single axis. Default implementation does nothing.

Parameters **axis** (*int*[484]) – axis number

[478] https://docs.python.org/dev/library/stdtypes.html#str
[479] https://docs.python.org/dev/library/stdtypes.html#str
[480] https://docs.python.org/dev/library/functions.html#int
[481] https://docs.python.org/dev/library/functions.html#int
[482] https://docs.python.org/dev/library/stdtypes.html#str
[483] https://docs.python.org/dev/library/stdtypes.html#str
[484] https://docs.python.org/dev/library/functions.html#int

StateAll()
Controller API. Override if necessary. Called to read the state of all selected axis. Default implementation does nothing.

StateOne(*axis*)
Controller API. Override is MANDATORY. Called to read the state of one axis. Default implementation raises `NotImplementedError`[485].

SetCtrlPar(*parameter*, *value*)
Controller API. Override if necessary. Called to set a parameter with a value. Default implementation sets this object member named '_'+parameter with the given value.

New in version 1.0.

GetCtrlPar(*parameter*)
Controller API. Override if necessary. Called to set a parameter with a value. Default implementation returns the value contained in this object's member named '_'+parameter.

New in version 1.0.

SetAxisPar(*axis*, *parameter*, *value*)
Controller API. Override is MANDATORY. Called to set a parameter with a value on the given axis. Default implementation calls deprecated *`SetPar()`* which, by default, raises `NotImplementedError`[486].

New in version 1.0.

GetAxisPar(*axis*, *parameter*)
Controller API. Override is MANDATORY. Called to get a parameter value on the given axis. Default implementation calls deprecated *`GetPar()`* which, by default, raises `NotImplementedError`[487].

New in version 1.0.

SetAxisExtraPar(*axis*, *parameter*, *value*)
Controller API. Override if necessary. Called to set a parameter with a value on the given axis. Default implementation calls deprecated *`SetExtraAttributePar()`* which, by default, raises `NotImplementedError`[488].

New in version 1.0.

GetAxisExtraPar(*axis*, *parameter*)
Controller API. Override if necessary. Called to get a parameter value on the given axis. Default implementation calls deprecated *`GetExtraAttributePar()`* which, by default, raises `NotImplementedError`[489].

New in version 1.0.

SetPar(*axis*, *parameter*, *value*)
Controller API. Called to set a parameter with a value on the given axis. Default implementation raises `NotImplementedError`[490].

Deprecated since version 1.0: use *`SetAxisPar()`* instead

[485] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[486] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[487] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[488] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[489] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[490] https://docs.python.org/dev/library/exceptions.html#NotImplementedError

GetPar(*axis, parameter*)
Controller API. Called to get a parameter value on the given axis. Default implementation raises `NotImplementedError`[491].

Deprecated since version 1.0: use *GetAxisPar()* instead

SetExtraAttributePar(*axis, parameter, value*)
Controller API. Called to set a parameter with a value on the given axis. Default implementation raises `NotImplementedError`[492].

Deprecated since version 1.0: use *SetAxisExtraPar()* instead

GetExtraAttributePar(*axis, parameter*)
Controller API. Called to get a parameter value on the given axis. Default implementation raises `NotImplementedError`[493].

Deprecated since version 1.0: use *GetAxisExtraPar()* instead

GetAxisAttributes(*axis*)
Controller API. Override if necessary. Returns a dictionary of all attributes per axis. Default implementation returns a new `dict`[494] with the standard attributes plus the *axis_attributes*

Parameters **axis** (*int*[495]) – axis number

Returns a dict containing attribute information as defined in *axis_attributes*

New in version 1.0.

SendToCtrl(*stream*)
Controller API. Override if necessary. Sends a string to the controller. Default implementation raises `NotImplementedError`[496].

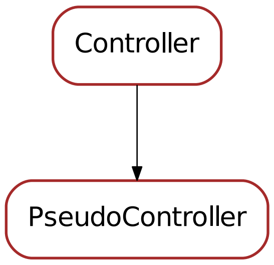
Parameters **stream** (*str*[497]) – stream to be sent

Returns any relevant information e.g. response of the controller

Return type str[498]

[491] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[492] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[493] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[494] https://docs.python.org/dev/library/stdtypes.html#dict
[495] https://docs.python.org/dev/library/functions.html#int
[496] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[497] https://docs.python.org/dev/library/stdtypes.html#str
[498] https://docs.python.org/dev/library/stdtypes.html#str

Abstract Pseudo Controller

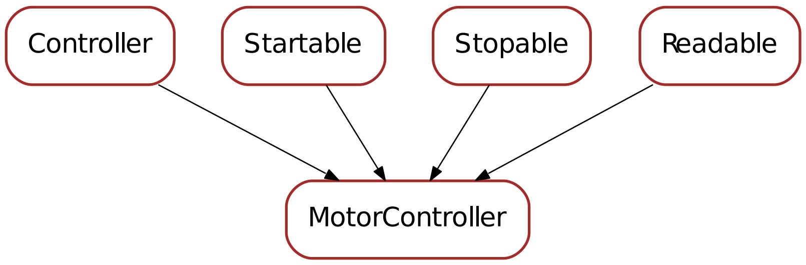


class PseudoController(*inst*, *props*, **args*, ***kwargs*)

Bases: *sardana.pool.controller.Controller*

Base class for all pseudo controllers.

Motor Controller API



class MotorController(*inst*, *props*, **args*, ***kwargs*)

Bases: *sardana.pool.controller.Controller*, *sardana.pool.controller.Startable*, *sardana.pool.controller.Stopable*, *sardana.pool.controller.Readable*

Base class for a motor controller. Inherit from this class to implement your own motor controller for the device pool.

A motor controller should support these axis parameters:

- acceleration
- deceleration
- velocity
- base_rate

- step_per_unit

These parameters are configured through the `GetAxisPar()`/`SetAxisPar()` API (in version <1.0 the methods were called `GetPar()`/`SetPar()`. Default `GetAxisPar()` and `SetAxisPar()` still call `GetPar()` and `SetPar()` respectively in order to maintain backward compatibility).

`NoLimitSwitch = 0`
A constant representing no active switch.

`HomeLimitSwitch = 1`
A constant representing an active *home* switch. You can *OR* two or more switches together. For example, to say both upper and lower limit switches are active:

```
limit_switches = self.HomeLimitSwitch | self.LowerLimitSwitch
```

`UpperLimitSwitch = 2`
A constant representing an active *upper limit* switch. You can *OR* two or more switches together. For example, to say both upper and lower limit switches are active:

```
limit_switches = self.UpperLimitSwitch | self.LowerLimitSwitch
```

`LowerLimitSwitch = 4`
A constant representing an active *lower limit* switch. You can *OR* two or more switches together. For example, to say both upper and lower limit switches are active:

```
limit_switches = self.UpperLimitSwitch | self.LowerLimitSwitch
```

`standard_axis_attributes = {'Acceleration': {'type': <type 'float'>, 'description':`
A `dict`[499] containing the standard attributes present on each axis device

`gender = 'Motor controller'`
A `str`[500] representing the controller gender

`GetAxisAttributes`(*axis*)
Motor Controller API. Override if necessary. Returns a sequence of all attributes per axis. Default implementation returns a `dict`[501] containning:

- Position
- DialPosition
- Offset
- Sign
- Step_per_unit
- Acceleration
- Deceleration
- Base_rate
- Velocity
- Backlash
- Limit_switches

[499] https://docs.python.org/dev/library/stdtypes.html#dict
[500] https://docs.python.org/dev/library/stdtypes.html#str
[501] https://docs.python.org/dev/library/stdtypes.html#dict

plus all attributes contained in *axis_attributes*

Note: Normally you don't need to Override this method. You just implement the class member *axis_attributes*. Typically, you will need to Override this method in two cases:

- certain axes contain a different set of extra attributes which cannot be simply defined in *axis_attributes*
- some axes (or all) don't implement a set of standard moveable parameters (ex.: if a motor controller is created to control a power supply, it may have a position (current) and a velocity (ramp speed) but it may not have acceleration)

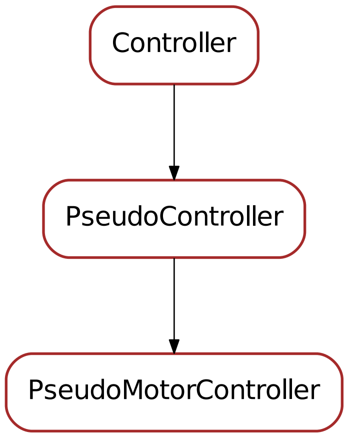
Parameters **axis** (*int*[502]) – axis number

Returns a dict containing attribute information as defined in *axis_attributes*

New in version 1.0.

DefinePosition(*axis*, *position*)
Motor Controller API. Override is recommended! This method is called to load a new motor position. Default implementation does nothing.

Pseudo Motor Controller API



class PseudoMotorController(*inst*, *props*, **args*, ***kwargs*)
Bases: *sardana.pool.controller.PseudoController*

Base class for a pseudo motor controller. Inherit from this class to implement your own pseudo motor controller for the device pool.

Every Pseudo Motor implementation must be a subclass of this class. Current procedure for a correct implementation of a Pseudo Motor class:

[502] https://docs.python.org/dev/library/functions.html#int

- **mandatory:**
  - define the class level attributes *pseudo_motor_roles*, *motor_roles*
  - write *CalcPseudo()* method
  - write *CalcPhysical()* method.
- **optional:**
  - write *CalcAllPseudo()* and *CalcAllPhysical()* if great performance gain can be achived

pseudo_motor_roles = ()
a sequence of strings describing the role of each pseudo motor axis in this controller

motor_roles = ()
a sequence of strings describing the role of each motor in this controller

standard_axis_attributes = {'Position': {'type': <type 'float'>, 'description': 'Po
A dict[503] containing the standard attributes present on each axis device

gender = 'Pseudo motor controller'
A str[504] representing the controller gender

CalcAllPseudo(*physical_pos, curr_pseudo_pos*)
Pseudo Motor Controller API. Override if necessary. Calculates the positions of all pseudo motors that belong to the pseudo motor system from the positions of the physical motors. Default implementation does a loop calling *PseudoMotorController.calc_pseudo()* for each pseudo motor role.

Parameters

- **physical_pos** (*sequence<float>*) – a sequence containing physical motor positions
- **curr_pseudo_pos** (*sequence<float>*) – a sequence containing the current pseudo motor positions

Returns a sequece of pseudo motor positions (one for each pseudo motor role)

Return type sequence<float>

New in version 1.0.

CalcAllPhysical(*pseudo_pos, curr_physical_pos*)
Pseudo Motor Controller API. Override if necessary. Calculates the positions of all motors that belong to the pseudo motor system from the positions of the pseudo motors. Default implementation does a loop calling *PseudoMotorController.calc_physical()* for each motor role.

Parameters

- **pseudo_pos** (*sequence<float>*) – a sequence containing pseudo motor positions
- **curr_physical_pos** (*sequence<float>*) – a sequence containing the current physical motor positions

Returns a sequece of motor positions (one for each motor role)

Return type sequence<float>

[503] https://docs.python.org/dev/library/stdtypes.html#dict
[504] https://docs.python.org/dev/library/stdtypes.html#str

New in version 1.0.

CalcPseudo (*axis*, *physical_pos*, *curr_pseudo_pos*)

Pseudo Motor Controller API. Override is **MANDATORY**. Calculate pseudo motor position given the physical motor positions

Parameters

- **axis** (*int*[505]) – the pseudo motor role axis
- **physical_pos** (*sequence<float>*) – a sequence containing motor positions
- **curr_pseudo_pos** (*sequence<float>*) – a sequence containing the current pseudo motor positions

Returns a pseudo motor position corresponding to the given axis pseudo motor role

Return type float[506]

New in version 1.0.

CalcPhysical (*axis*, *pseudo_pos*, *curr_physical_pos*)

Pseudo Motor Controller API. Override is **MANDATORY**. Calculate physical motor position given the pseudo motor positions.

Parameters

- **axis** (*int*[507]) – the motor role axis
- **pseudo_pos** (*sequence<float>*) – a sequence containing pseudo motor positions
- **curr_physical_pos** (*sequence<float>*) – a sequence containing the current physical motor positions

Returns a motor position corresponding to the given axis motor role

Return type float[508]

New in version 1.0.

calc_all_pseudo (*physical_pos*)

Pseudo Motor Controller API. Override if necessary. Calculates the positions of all pseudo motors that belong to the pseudo motor system from the positions of the physical motors. Default implementation does a loop calling `PseudoMotorController.calc_pseudo()` for each pseudo motor role.

Parameters **physical_pos** (*sequence<float>*) – a sequence of physical motor positions

Returns a sequece of pseudo motor positions (one for each pseudo motor role)

Return type sequence<float>

Deprecated since version 1.0: implement `CalcAllPseudo()` instead

calc_all_physical (*pseudo_pos*)

Pseudo Motor Controller API. Override if necessary. Calculates the positions of all motors that belong to the pseudo motor system from the positions of the pseudo motors. Default implementation does a loop calling `PseudoMotorController.calc_physical()` for each motor role.

505 https://docs.python.org/dev/library/functions.html#int
506 https://docs.python.org/dev/library/functions.html#float
507 https://docs.python.org/dev/library/functions.html#int
508 https://docs.python.org/dev/library/functions.html#float

Parameters **`pseudo_pos`** (*`sequence<float>`*) – a sequence of pseudo motor positions

Returns a sequece of motor positions (one for each motor role)

Return type sequence<float>

Deprecated since version 1.0: implement *`CalcAllPhysical()`* instead

`calc_pseudo`(*axis*, *physical_pos*)

Pseudo Motor Controller API. Override is **MANDATORY**. Calculate pseudo motor position given the physical motor positions

Parameters

- **`axis`** (*`int`*[509]) – the pseudo motor role axis
- **`physical_pos`** (*`sequence<float>`*) – a sequence of motor positions

Returns a pseudo motor position corresponding to the given axis pseudo motor role

Return type float[510]

Deprecated since version 1.0: implement *`CalcPseudo()`* instead

`calc_physical`(*axis*, *pseudo_pos*)

Pseudo Motor Controller API. Override is **MANDATORY**. Calculate physical motor position given the pseudo motor positions.

Parameters

- **`axis`** (*`int`*[511]) – the motor role axis
- **`pseudo_pos`** (*`sequence<float>`*) – a sequence of pseudo motor positions

Returns a motor position corresponding to the given axis motor role

Return type float[512]

Deprecated since version 1.0: implement *`CalcPhysical()`* instead

`GetMotor`(*index_or_role*)

Returns the motor for a given role/index.

Warning:

- Use with care: Executing motor methods can be dangerous!
- Since the controller is built before any element (including motors), this method will **FAIL** when called from the controller constructor

Parameters **`index_or_role`** (*`int`*[513] *`or`* *`str`*[514]) – index number or role name

Returns Motor object for the given role/index

Return type *`PoolMotor`*

[509] https://docs.python.org/dev/library/functions.html#int
[510] https://docs.python.org/dev/library/functions.html#float
[511] https://docs.python.org/dev/library/functions.html#int
[512] https://docs.python.org/dev/library/functions.html#float
[513] https://docs.python.org/dev/library/functions.html#int
[514] https://docs.python.org/dev/library/stdtypes.html#str

GetPseudoMotor(*index_or_role*)
Returns the pseudo motor for a given role/index.

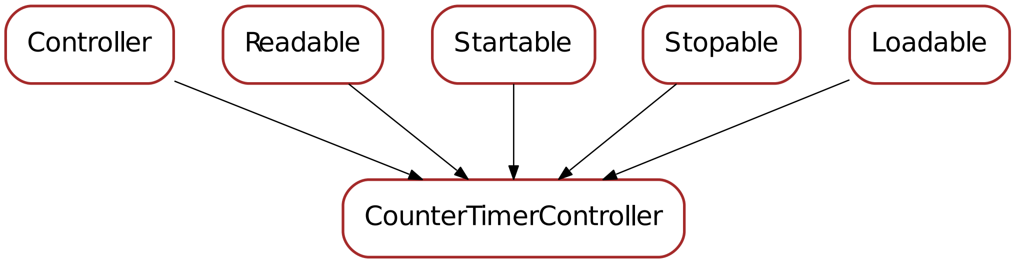
> **Warning:**
>
> - Use with care: Executing pseudo motor methods can be dangerous!
> - Since the controller is built before any element (including pseudo motors), this method will **FAIL** when called from the controller constructor

Parameters **index_or_role** (`int`[515] `or` `str`[516]) – index number or role name

Returns PseudoMotor object for the given role/index

Return type `PoolPseudoMotor`

Counter Timer Controller API



class CounterTimerController(*inst, props, *args, **kwargs*)
Bases: `sardana.pool.controller.Controller`, `sardana.pool.controller.Readable`, `sardana.pool.controller.Startable`, `sardana.pool.controller.Stopable`, `sardana.pool.controller.Loadable`

Base class for a counter/timer controller. Inherit from this class to implement your own counter/timer controller for the device pool.

A counter timer controller should support these controller parameters:

- timer
- monitor
- trigger_type

standard_axis_attributes = {'Data': {'type': <type 'str'>, 'description': 'Data'},
A `dict`[517] containing the standard attributes present on each axis device

[515] https://docs.python.org/dev/library/functions.html#int
[516] https://docs.python.org/dev/library/stdtypes.html#str
[517] https://docs.python.org/dev/library/stdtypes.html#dict

gender = 'Counter/Timer controller'
A `str`[518] representing the controller gender

get_trigger_type()

PreStartAllCT()
Counter/Timer Controller API. Override if necessary. Called to prepare an acquisition of all selected axis. Default implementation does nothing.

Deprecated since version 1.0: use *PreStartAll()* instead

PreStartOneCT(*axis*)
Counter/Timer Controller API. Override if necessary. Called to prepare an acquisition a single axis. Default implementation returns True.

Parameters **axis** (*int*[519]) – axis number

Returns True means a successfull PreStartOneCT or False for a failure

Return type bool[520]

Deprecated since version 1.0: use *PreStartOne()* instead

StartOneCT(*axis*)
Counter/Timer Controller API. Override if necessary. Called to start an acquisition of a selected axis. Default implementation does nothing.

Parameters **axis** (*int*[521]) – axis number

Deprecated since version 1.0: use *StartOne()* instead

StartAllCT()
Counter/Timer Controller API. Override is MANDATORY! Called to start an acquisition of a selected axis. Default implementation raises `NotImplementedError`[522].

Deprecated since version 1.0: use *StartAll()* instead

PreStartAll()
Controller API. Override if necessary. Called to prepare a write of the position of all axis. Default implementation calls deprecated *PreStartAllCT()* which, by default, does nothing.

New in version 1.0.

PreStartOne(*axis*, *value=None*)
Controller API. Override if necessary. Called to prepare a write of the position of a single axis. Default implementation calls deprecated *PreStartOneCT()* which, by default, returns True.

Parameters

- **axis** (*int*[523]) – axis number
- **value** (*float*[524]) – the value

Returns True means a successfull pre-start or False for a failure

Return type bool[525]

New in version 1.0.

[518] https://docs.python.org/dev/library/stdtypes.html#str
[519] https://docs.python.org/dev/library/functions.html#int
[520] https://docs.python.org/dev/library/functions.html#bool
[521] https://docs.python.org/dev/library/functions.html#int
[522] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[523] https://docs.python.org/dev/library/functions.html#int
[524] https://docs.python.org/dev/library/functions.html#float
[525] https://docs.python.org/dev/library/functions.html#bool

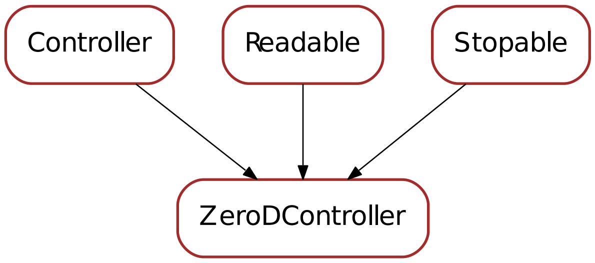
StartOne (*axis*, *value=None*)
: **Controller API**. Override if necessary. Called to write the position of a selected axis. Default implementation calls deprecated `StartOneCT()` which, by default, does nothing.

  Parameters

  - **axis** (`int`[526]) – axis number
  - **value** (`float`[527]) – the value

StartAll ()
: **Controller API**. Override is MANDATORY! Default implementation calls deprecated `StartAllCT()` which, by default, raises `NotImplementedError`[528].

0D Controller API



class ZeroDController (*inst*, *props*, **args*, ***kwargs*)
Bases: `sardana.pool.controller.Controller`, `sardana.pool.controller.Readable`, `sardana.pool.controller.Stopable`

Base class for a 0D controller. Inherit from this class to implement your own 0D controller for the device pool.

standard_axis_attributes = {'Data': {'type': <type 'str'>, 'description': 'Data'},
: A `dict`[529] containing the standard attributes present on each axis device

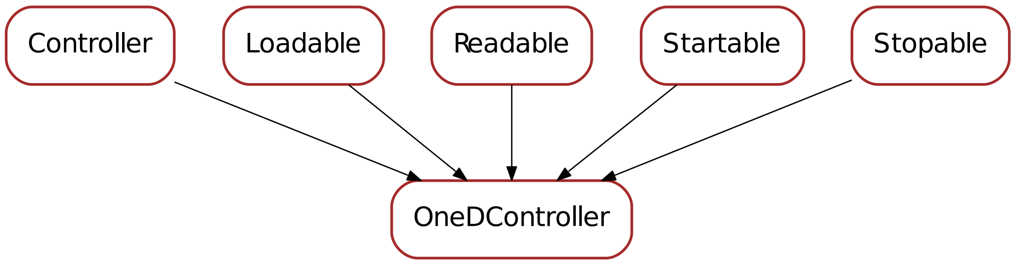
gender = '0D controller'
: A `str`[530] representing the controller gender

AbortOne (*axis*)
: This method is not executed by the system. Default implementation does nothing.

  Parameters axis (`int`[531]) – axis number

[526] https://docs.python.org/dev/library/functions.html#int
[527] https://docs.python.org/dev/library/functions.html#float
[528] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[529] https://docs.python.org/dev/library/stdtypes.html#dict
[530] https://docs.python.org/dev/library/stdtypes.html#str
[531] https://docs.python.org/dev/library/functions.html#int

1D Controller API



class OneDController(*inst*, *props*, **args*, ***kwargs*)

Bases: *sardana.pool.controller.Controller*, *sardana.pool.controller.Readable*, *sardana.pool.controller.Startable*, *sardana.pool.controller.Stopable*, *sardana.pool.controller.Loadable*

Base class for a 1D controller. Inherit from this class to implement your own 1D controller for the device pool.

New in version 1.2.

standard_axis_attributes = {'Data': {'type': <type 'str'>, 'description': 'Data'},

gender = '1D controller'

A `str`[532] representing the controller gender

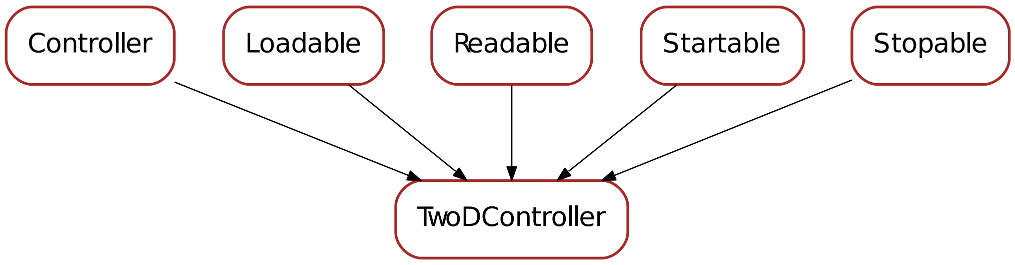
GetAxisPar(*axis*, *parameter*)

Controller API. Override is MANDATORY. Called to get a parameter value on the given axis. If parameter == 'data_source', default implementation returns None, meaning let sardana decide the proper URI for accessing the axis value. Otherwise, default implementation calls deprecated *GetPar()* which, by default, raises `NotImplementedError`[533].

New in version 1.2.

[532] https://docs.python.org/dev/library/stdtypes.html#str
[533] https://docs.python.org/dev/library/exceptions.html#NotImplementedError

2D Controller API



class TwoDController(*inst, props, *args, **kwargs*)

Bases: *sardana.pool.controller.Controller*, *sardana.pool.controller.Readable*, *sardana.pool.controller.Startable*, *sardana.pool.controller.Stopable*, *sardana.pool.controller.Loadable*

Base class for a 2D controller. Inherit from this class to implement your own 2D controller for the device pool.

standard_axis_attributes = {'Value': {'maxdimsize': (4096, 4096), 'type': ((<type '

gender = '2D controller'

A `str`[534] representing the controller gender

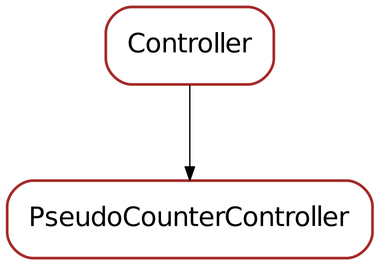
GetAxisPar(*axis, parameter*)

Controller API. Override is MANDATORY. Called to get a parameter value on the given axis. If parameter == 'data_source', default implementation returns None, meaning let sardana decide the proper URI for accessing the axis value. Otherwise, default implementation calls deprecated *GetPar()* which, by default, raises `NotImplementedError`[535].

New in version 1.2.

[534] https://docs.python.org/dev/library/stdtypes.html#str
[535] https://docs.python.org/dev/library/exceptions.html#NotImplementedError

Pseudo Counter Controller API



`class PseudoCounterController`(*inst, props, *args, **kwargs*)

Bases: *`sardana.pool.controller.Controller`*

Base class for a pseudo counter controller. Inherit from this class to implement your own pseudo counter controller for the device pool.

Every Pseudo Counter implementation must be a subclass of this class. Current procedure for a correct implementation of a Pseudo Counter class:

- **mandatory:**
  - define the class level attributes *`counter_roles`*,
  - write *`Calc()`* method

`pseudo_counter_roles = ()`

a sequence of strings describing the role of each pseudo counter axis in this controller

`counter_roles = ()`

a sequence of strings describing the role of each counter in this controller

`standard_axis_attributes = {'Data': {'type': <type 'str'>, 'description': 'Data'},`

A `dict`[536] containing the standard attributes present on each axis device

`gender = 'Pseudo counter controller'`

A `str`[537] representing the controller gender

`Calc`(*axis, values*)

Pseudo Counter Controller API. Override is **MANDATORY**. Calculate pseudo counter position given the counter values.

Parameters

- **`axis`** (*`int`*[538]) – the pseudo counter role axis
- **`values`** (*`sequence<float>`*) – a sequence containing current values of underlying elements

Returns a pseudo counter value corresponding to the given axis pseudo counter role

[536] https://docs.python.org/dev/library/stdtypes.html#dict
[537] https://docs.python.org/dev/library/stdtypes.html#str
[538] https://docs.python.org/dev/library/functions.html#int

Return type float[539]

New in version 1.0.

calc(*axis*, *values*)

Pseudo Counter Controller API. Override is **MANDATORY**. Calculate pseudo counter value given the counter values.

Parameters

- **axis** (*int*[540]) – the pseudo counter role axis
- **values** (*sequence<float>*) – a sequence containing current values of underlying elements

Returns a pseudo counter value corresponding to the given axis pseudo counter role

Return type float[541]

Deprecated since version 1.0: implement `Calc()` instead

CalcAll(*values*)

Pseudo Counter Controller API. Override if necessary. Calculates all pseudo counter values from the values of counters. Default implementation does a loop calling `PseudoCounterController.Calc()` for each pseudo counter role.

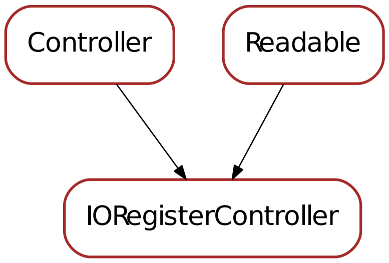
Parameters values (*sequence<float>*) – a sequence containing current values of underlying elements

Returns a sequece of pseudo counter values (one for each pseudo counter role)

Return type sequence<float>

New in version 1.2.

IO Register Controller API



class IORegisterController(*inst*, *props*, **args*, ***kwargs*)

Bases: `sardana.pool.controller.Controller`, `sardana.pool.controller.Readable`

[539] https://docs.python.org/dev/library/functions.html#float
[540] https://docs.python.org/dev/library/functions.html#int
[541] https://docs.python.org/dev/library/functions.html#float

Base class for a IORegister controller. Inherit from this class to implement your own IORegister controller for the device pool.

`predefined_values = ()`
Deprecated since version 1.0.

use *`axis_attributes`* instead

`standard_axis_attributes = {'Value': {'type': <type 'float'>, 'description': 'Value`
A `dict`[542] containing the standard attributes present on each axis device

`gender = 'I/O register controller'`
A `str`[543] representing the controller gender

`WriteOne` (*axis*, *value*)
IORegister Controller API. Override if necessary.

`pool`

This module contains the main pool class

Functions

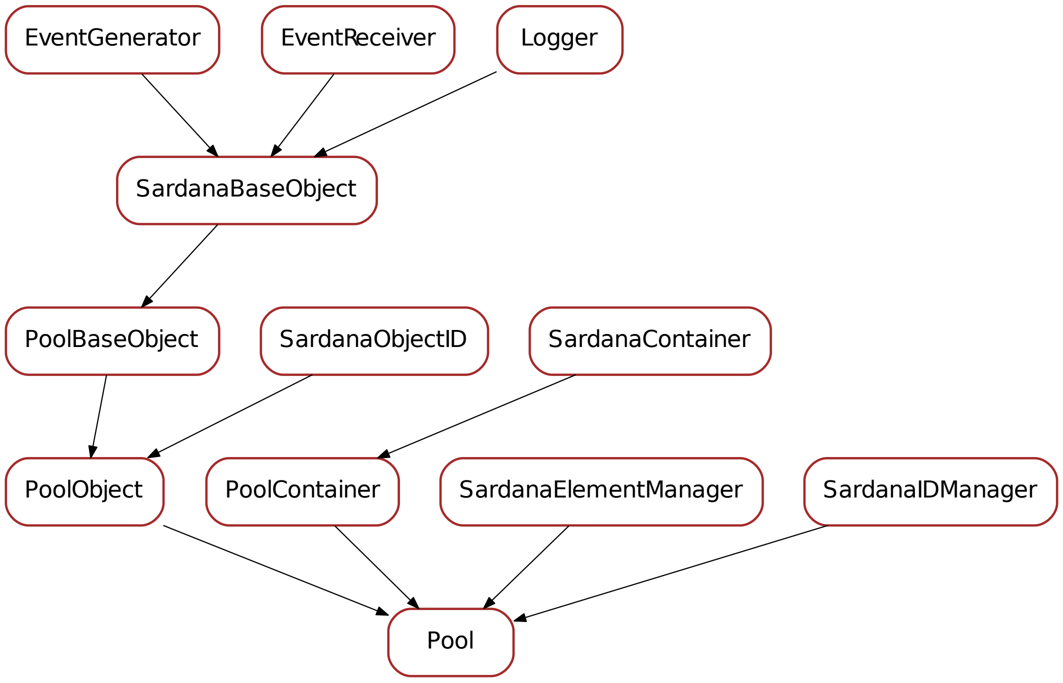
- `get_thread_pool()`

Classes

- *`Pool`*

[542] https://docs.python.org/dev/library/stdtypes.html#dict
[543] https://docs.python.org/dev/library/stdtypes.html#str

Pool



class Pool(*full_name*, *name=None*)

Bases: *sardana.pool.poolcontainer.PoolContainer*, *sardana.pool.poolobject.PoolObject*, *sardana.sardanamanager.SardanaElementManager*, `sardana.sardanamanager.SardanaIDManager`

The central pool class.

Default_MotionLoop_StatesPerPosition = 10
Default value representing the number of state reads per position read during a motion loop

Default_MotionLoop_SleepTime = 0.01
Default value representing the sleep time for each motion loop

Default_AcqLoop_StatesPerValue = 10
Default value representing the number of state reads per value read during a motion loop

Default_AcqLoop_SleepTime = 0.01
Default value representing the sleep time for each acquisition loop

Default_DriftCorrection = True

init_local_logging()

clear_remote_logging()

init_remote_logging(*host=None*, *port=None*)
Initializes remote logging.

Parameters

- **host** (*str*[544]) – host name [default: None, meaning use the machine host name as returned by `socket.gethostname()`[545]].
- **port** – port number [default: None, meaning use `logging.handlers.DEFAULT_TCP_LOGGING_PORT`

serialize(**args, **kwargs*)

set_motion_loop_sleep_time(*motion_loop_sleep_time*)

get_motion_loop_sleep_time()

motion_loop_sleep_time
motion sleep time (s)

set_motion_loop_states_per_position(*motion_loop_states_per_position*)

get_motion_loop_states_per_position()

motion_loop_states_per_position
Number of State reads done before doing a position read in the motion loop

set_acq_loop_sleep_time(*acq_loop_sleep_time*)

get_acq_loop_sleep_time()

acq_loop_sleep_time
acquisition sleep time (s)

set_acq_loop_states_per_value(*acq_loop_states_per_value*)

get_acq_loop_states_per_value()

acq_loop_states_per_value
Number of State reads done before doing a value read in the acquisition loop

set_drift_correction(*drift_correction*)

get_drift_correction()

drift_correction
drift correction

monitor

ctrl_manager

set_python_path(*path*)

set_path(*path*)

get_controller_libs()

get_controller_lib_names()

get_controller_class_names()

get_controller_classes()

get_controller_class_info(*name*)

get_controller_classes_info(*names*)

get_controller_libs_summary_info()

[544] https://docs.python.org/dev/library/stdtypes.html#str
[545] https://docs.python.org/dev/library/socket.html#socket.gethostname

get_controller_classes_summary_info()

get_elements_str_info(*obj_type=None*)

get_elements_info(*obj_type=None*)

get_acquisition_elements_info()

get_acquisition_elements_str_info()

create_controller(***kwargs*)

create_element(***kwargs*)

create_motor_group(***kwargs*)

create_measurement_group(***kwargs*)

rename_element(*old_name, new_name*)
Rename an object

Parameters

- **old_name** (str[546]) – old object name
- **new_name** (str[547]) – new object name

delete_element(*name*)

create_instrument(*full_name, klass_name, id=None*)

stop()

abort()

reload_controller_lib(*lib_name*)

reload_controller_class(*class_name*)

get_element_id_graph()

get_moveable_id_graph()

get_moveable_graph()

poolacquisition

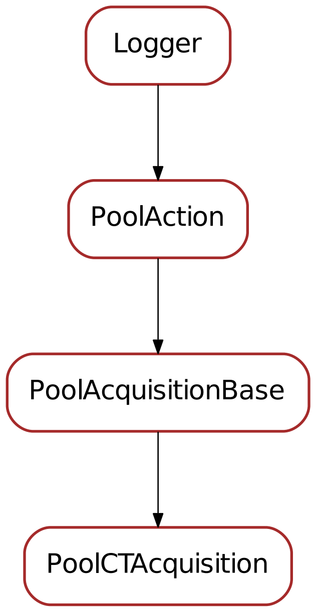
This module is part of the Python Pool libray. It defines the class for an acquisition

Classes

- *PoolCTAcquisition*

[546] https://docs.python.org/dev/library/stdtypes.html#str
[547] https://docs.python.org/dev/library/stdtypes.html#str

PoolCTAcquisition



class PoolCTAcquisition(*main_element*, *name='CTAcquisition'*, *slaves=None*)
Bases: `sardana.pool.poolacquisition.PoolAcquisitionBase`

get_read_value_loop_ctrls()

in_acquisition(*states*)
Determines if we are in acquisition or if the acquisition has ended based on the current unit trigger modes and states returned by the controller(s)

Parameters states (*dict<PoolElement, State>*) – a map containing state information as returned by read_state_info

Returns returns True if in acquisition or False otherwise

Return type bool[548]

action_loop

poolaction

This module is part of the Python Pool libray. It defines the class for an abstract action over a set of pool elements

[548] https://docs.python.org/dev/library/functions.html#bool

Functions

- *get_thread_pool()*

Classes

- *PoolAction*
- *OperationInfo*
- *PoolActionItem*
- *ActionContext*

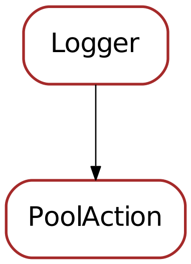
get_thread_pool()

Returns the global pool of threads for Sardana

Returns the global pool of threads object

Return type taurus.core.util.ThreadPool[549]

PoolAction



class PoolAction(*main_element*, *name='GlobalAction'*)

Bases: taurus.core.util.log.Logger

A generic class to handle any type of operation (like motion or acquisition)

get_main_element()

Returns the main element for this action

Returns sardana.pool.poolelement.PoolElement

main_element

Returns the main element for this action

Returns sardana.pool.poolelement.PoolElement

get_pool()

Returns the pool object for this action

Returns sardana.pool.pool.Pool

[549] http://taurus-scada.org/devel/api/taurus/core/util/_ThreadPool.html#taurus.core.util.ThreadPool

pool
Returns the pool object for this action

Returns sardana.pool.pool.Pool

clear_elements()
Clears all elements from this action

add_element(*element*)
Adds a new element to this action.

Parameters **element** (`sardana.pool.poolelement.PoolElement`) – the new element to be added

remove_element(*element*)
Removes an element from this action. If the element is not part of this action, a ValueError is raised.

Parameters **element** (`sardana.pool.poolelement.PoolElement`) – the new element to be removed

Raises ValueError

get_elements(*copy_of=False*)
Returns a sequence of all elements involved in this action.

Parameters **copy_of** (*bool*[550]) – If False (default) the internal container of elements is returned. If True, a copy of the internal container is returned instead

Returns a sequence of all elements involved in this action.

Return type seq<sardana.pool.poolelement.PoolElement>

get_pool_controller_list()
Returns a list of all controller elements involved in this action.

Returns a list of all controller elements involved in this action.

Return type list<sardana.pool.poolelement.PoolController>

get_pool_controllers()
Returns a dict of all controller elements involved in this action.

Returns a dict of all controller elements involved in this action.

Return type dict<sardana.pool.poolelement.PoolController, seq<sardana.pool.poolelement.PoolElement>>

is_running()
Determines if this action is running or not

Returns True if action is running or False otherwise

Return type bool[551]

run(**args*, ***kwargs*)
Runs this action

start_action(**args*, ***kwargs*)
Start procedure for this action. Default implementation raises NotImplementedError

Raises NotImplementedError

[550] https://docs.python.org/dev/library/functions.html#bool
[551] https://docs.python.org/dev/library/functions.html#bool

set_finish_hooks(*hooks*)
Set finish hooks for this action.

Parameters hooks (*OrderedDict or None*[552]) – an ordered dictionary where keys are the hooks and values is a flag if the hook is permanent (not removed after the execution)

add_finish_hook(*hook*, *permanent=True*)
Append one finish hook to this action.

Parameters

- **hook** (*callable*) – hook to be appended
- **permanent** (*boolean*) – flag if the hook is permanent (not removed after the execution)

remove_finish_hook(*hook*)
Remove finish hook.

finish_action()
Finishes the action execution. If a finish hook is defined it safely executes it. Otherwise nothing happens

stop_action(**args*, ***kwargs*)
Stop procedure for this action.

abort_action(**args*, ***kwargs*)
Aborts procedure for this action

emergency_break()
Tries to execute a stop. If it fails try an abort

was_stopped()
Determines if the action has been stopped from outside

Returns True if action has been stopped from outside or False otherwise

Return type bool[553]

was_aborted()
Determines if the action has been aborted from outside

Returns True if action has been aborted from outside or False otherwise

Return type bool[554]

was_action_interrupted()
Determines if the action has been interruped from outside (either from an abort or a stop).

Returns True if action has been interruped from outside or False otherwise

Return type bool[555]

action_loop()
Action loop for this action. Default implementation raises NotImplementedError

Raises NotImplementedError

read_state_info(*ret=None*, *serial=False*)
Reads state information of all elements involved in this action

[552] https://docs.python.org/dev/library/constants.html#None
[553] https://docs.python.org/dev/library/functions.html#bool
[554] https://docs.python.org/dev/library/functions.html#bool
[555] https://docs.python.org/dev/library/functions.html#bool

Parameters

- **`ret`** (*`dict`*[556]) – output map parameter that should be filled with state information. If None is given (default), a new map is created an returned
- **`serial`** (*`bool`*[557]) – If False (default) perform controller HW state requests in parallel. If True, access is serialized.

Returns a map containing state information per element

Return type dict<sardana.pool.poolelement.PoolElement, stateinfo>

`raw_read_state_info`(*ret=None*, *serial=False*)

Unsafe. Reads state information of all elements involved in this action

Parameters

- **`ret`** (*`dict`*[558]) – output map parameter that should be filled with state information. If None is given (default), a new map is created an returned
- **`serial`** (*`bool`*[559]) – If False (default) perform controller HW state requests in parallel. If True, access is serialized.

Returns a map containing state information per element

Return type dict<sardana.pool.poolelement.PoolElement, stateinfo>

`get_read_value_ctrls`()

`read_value`(*ret=None*, *serial=False*)

Reads value information of all elements involved in this action

Parameters

- **`ret`** (*`dict`*[560]) – output map parameter that should be filled with value information. If None is given (default), a new map is created an returned
- **`serial`** (*`bool`*[561]) – If False (default) perform controller HW value requests in parallel. If True, access is serialized.

Returns a map containing value information per element

Return type dict<:class:~'sardana.pool.poolelement.PoolElement', (value object, Exception[562] or None)>

`raw_read_value`(*ret=None*, *serial=False*)

Unsafe. Reads value information of all elements involved in this action

Parameters

- **`ret`** (*`dict`*[563]) – output map parameter that should be filled with value information. If None is given (default), a new map is created an returned
- **`serial`** (*`bool`*[564]) – If False (default) perform controller HW value requests in parallel. If True, access is serialized.

Returns a map containing value information per element

[556] https://docs.python.org/dev/library/stdtypes.html#dict
[557] https://docs.python.org/dev/library/functions.html#bool
[558] https://docs.python.org/dev/library/stdtypes.html#dict
[559] https://docs.python.org/dev/library/functions.html#bool
[560] https://docs.python.org/dev/library/stdtypes.html#dict
[561] https://docs.python.org/dev/library/functions.html#bool
[562] https://docs.python.org/dev/library/exceptions.html#Exception
[563] https://docs.python.org/dev/library/stdtypes.html#dict
[564] https://docs.python.org/dev/library/functions.html#bool

Return type dict<:class:~'sardana.pool.poolelement.PoolElement, `sardana.sardanavalue.SardanaValue`>

get_read_value_loop_ctrls()

read_value_loop(*ret=None*, *serial=False*)
Reads value information of all elements involved in this action

Parameters

- **ret** (*dict*[565]) – output map parameter that should be filled with value information. If None is given (default), a new map is created an returned
- **serial** (*bool*[566]) – If False (default) perform controller HW value requests in parallel. If True, access is serialized.

Returns a map containing value information per element

Return type dict<:class:~'sardana.pool.poolelement.PoolElement', (value object, Exception[567] or None)>

raw_read_value_loop(*ret=None*, *serial=False*)
Unsafe. Reads value information of all elements involved in this action

Parameters

- **ret** (*dict*[568]) – output map parameter that should be filled with value information. If None is given (default), a new map is created an returned
- **serial** (*bool*[569]) – If False (default) perform controller HW value requests in parallel. If True, access is serialized.

Returns a map containing value information per element

Return type dict<:class:~'sardana.pool.poolelement.PoolElement, `sardana.sardanavalue.SardanaValue`>

OperationInfo

class OperationInfo
Bases: `object`[570]

Stores synchronization data for a certain operation

[565] https://docs.python.org/dev/library/stdtypes.html#dict
[566] https://docs.python.org/dev/library/functions.html#bool
[567] https://docs.python.org/dev/library/exceptions.html#Exception
[568] https://docs.python.org/dev/library/stdtypes.html#dict
[569] https://docs.python.org/dev/library/functions.html#bool
[570] https://docs.python.org/dev/library/functions.html#object

init(*count*)
Initializes this operation with a certain count

wait(*timeout=None*)
waits for the operation to finish

finish_one()
Notifies this operation that one step was finished

acquire()
Acquires this operation lock

release()
Releases this operation lock

PoolActionItem

class PoolActionItem(*element*)
Bases: `object`[571]

The base class for an atomic action item

get_element()
Returns the element associated with this item

set_element(*element*)
Sets the element for this item

element
Returns the element associated with this item

ActionContext

[571] https://docs.python.org/dev/library/functions.html#object

class ActionContext(*pool_action*)

Bases: object[572]

Stores an atomic action context

enter()

Enters operation

exit()

Leaves operation

poolbasechannel

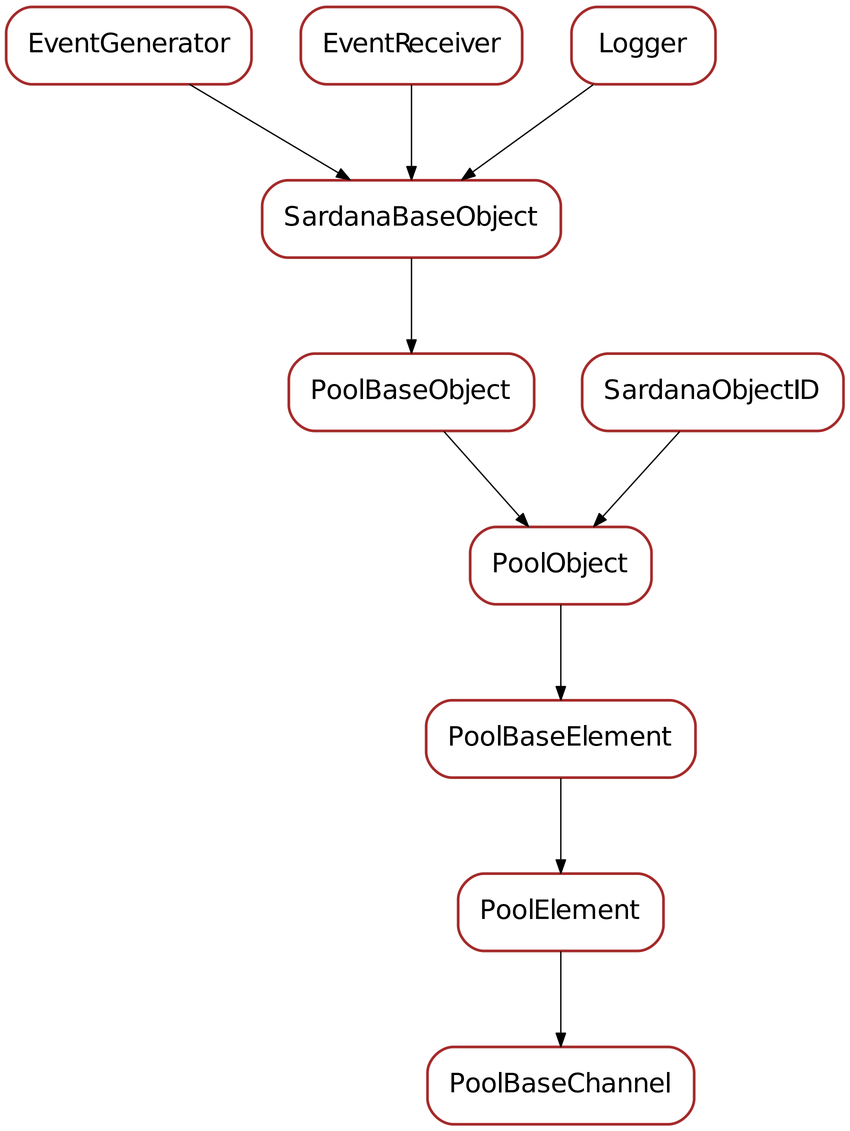
This module is part of the Python Pool library. It defines the base classes for experiment channels

Classes

- *PoolBaseChannel*

[572] https://docs.python.org/dev/library/functions.html#object

PoolBaseChannel



class PoolBaseChannel(***kwargs*)

Bases: *sardana.pool.poolelement.PoolElement*

ValueAttributeClass
alias of `Value`

ValueBufferClass
alias of `ValueBuffer`

AcquisitionClass

alias of `sardana.pool.poolacquisition.`

`has_pseudo_elements`()
Informs whether this channel forms part of any pseudo element e.g. pseudo counter.

Returns has pseudo elements

Return type bool[573]

`get_pseudo_elements`()
Returns list of pseudo elements e.g. pseudo counters that this channel belongs to.

Returns pseudo elements

Return type seq<*`PoolPseudoCounter`*>

`add_pseudo_element`(*element*)
Adds pseudo element e.g. pseudo counter that this channel belongs to.

Parameters **`element`** (*`PoolPseudoCounter`*) – pseudo element

`remove_pseudo_element`(*element*)
Removes pseudo element e.g. pseudo counters that this channel belongs to.

Parameters **`element`** (*`PoolPseudoCounter`*) – pseudo element

`get_value_attribute`()
Returns the value attribute object for this experiment channel

Returns the value attribute

Return type *`SardanaAttribute`*

`get_value_buffer`()
Returns the value attribute object for this experiment channel

Returns the value attribute

Return type *`SardanaAttribute`*

`on_change`(*evt_src*, *evt_type*, *evt_value*)

`get_default_attribute`()

`get_acquisition`()

`acquisition`
acquisition object

`read_value`()
Reads the channel value from hardware.

Returns a *`SardanaValue`* containing the channel value

Return type *`SardanaValue`*

`put_value`(*value*, *propagate=1*)
Sets a value.

Parameters

- **`value`** (*`SardanaValue`*) – the new value
- **`propagate`** (*`int`*[574]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

[573] https://docs.python.org/dev/library/functions.html#bool
[574] https://docs.python.org/dev/library/functions.html#int

get_value(*cache=True*, *propagate=1*)
Returns the channel value.

Parameters

- **cache** (*bool*[575]) – if True (default) return value in cache, otherwise read value from hardware
- **propagate** (*int*[576]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

Returns the channel value

Return type *SardanaAttribute*

set_value(*value*)
Starts an acquisition on this channel

Parameters **value** (Number[577]) – the value to count

value
channel value

extend_value_buffer(*values*, *idx=None*, *propagate=1*)
Extend value buffer with new values assigning them consecutive indexes starting with idx. If idx is omitted, then the new values will be added right after the last value in the buffer. Also update the read value of the attribute with the last element of values.

Parameters

- **values** (*SardanaValue*) – values to be added to the buffer
- **propagate** (*int*[578]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

append_value_buffer(*value*, *idx=None*, *propagate=1*)
Extend value buffer with new values assigning them consecutive indexes starting with idx. If idx is omitted, then the new value will be added with right after the last value in the buffer. Also update the read value.

Parameters

- **value** (*SardanaValue*) – value to be added to the buffer
- **propagate** (*int*[579]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

clear_value_buffer()

start_acquisition(*value=None*)

poolbaseobject

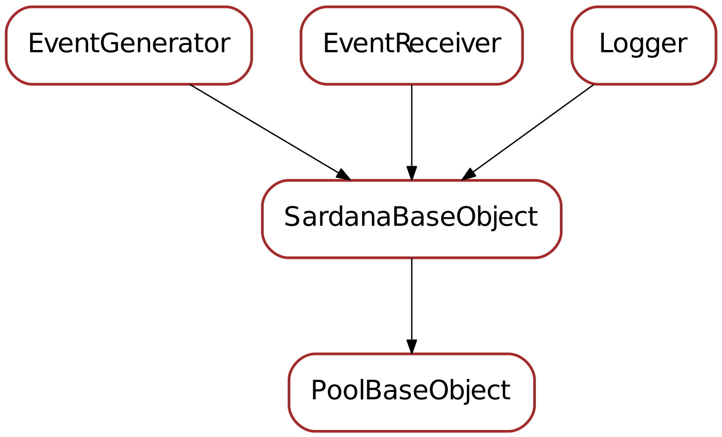
This module is part of the Python Pool library. It defines the base classes for Pool object

[575] https://docs.python.org/dev/library/functions.html#bool
[576] https://docs.python.org/dev/library/functions.html#int
[577] https://docs.python.org/dev/library/numbers.html#numbers.Number
[578] https://docs.python.org/dev/library/functions.html#int
[579] https://docs.python.org/dev/library/functions.html#int

Classes

- *PoolBaseObject*

PoolBaseObject



class PoolBaseObject(***kwargs*)
Bases: `sardana.sardanabase.SardanaBaseObject`

The Pool most abstract object.

get_pool()
Return the `sardana.pool.pool.Pool` which *owns* this pool object.

Returns the pool which *owns* this pool object.

Return type `sardana.pool.pool.Pool`

serialize(**args*, ***kwargs*)

pool
reference to the `sardana.pool.pool.Pool`

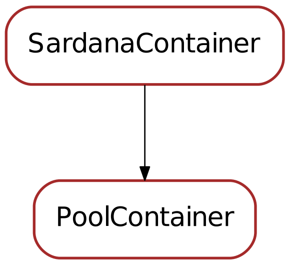
poolcontainer

This module is part of the Python Pool libray. It defines the base classes for a pool container element

Classes

- *PoolContainer*

PoolContainer



class PoolContainer

Bases: *sardana.sardanacontainer.SardanaContainer*

A container class for pool elements

get_controller_class(***kwargs*)

get_controller_class_by_id(*eid*, ***kwargs*)

get_controller_class_by_name(*name*, ***kwargs*)

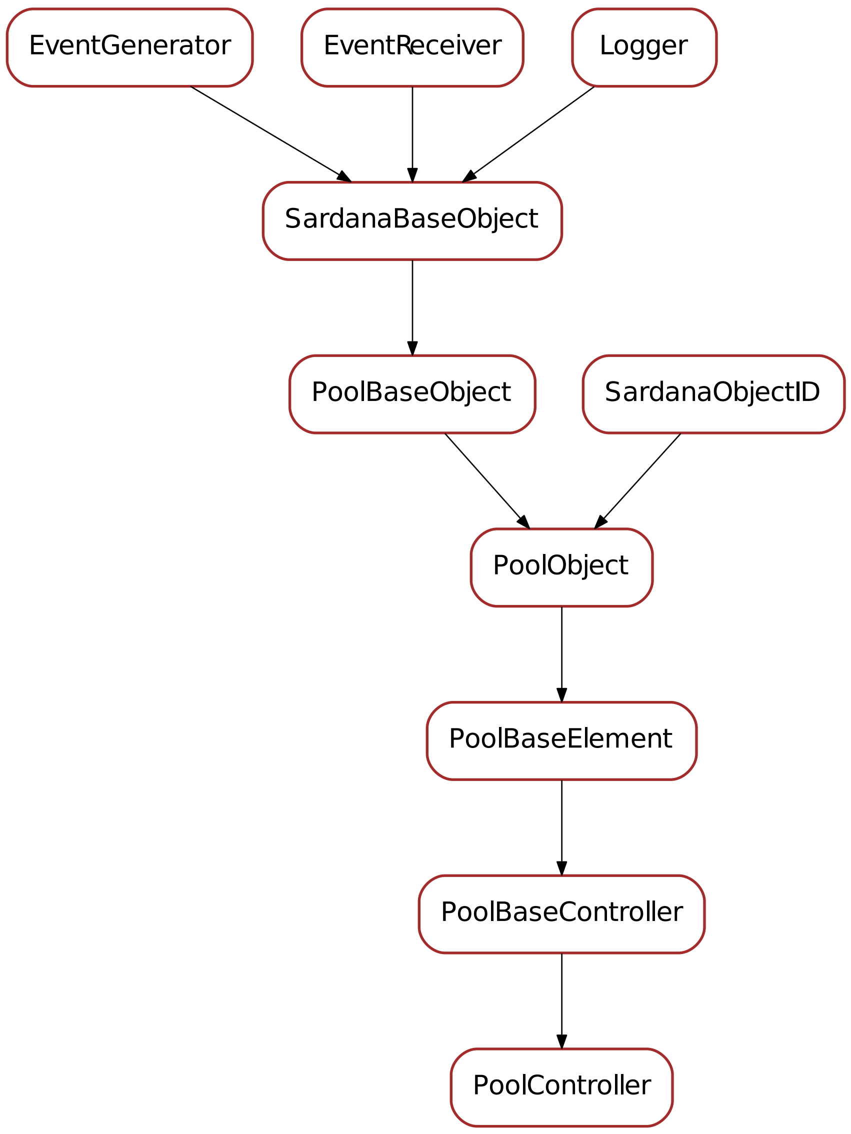
poolcontroller

This module is part of the Python Pool library. It defines the base classes for

Classes

- *PoolController*
- *PoolPseudoMotorController*
- *PoolPseudoCounterController*

PoolController



class PoolController(***kwargs*)

Bases: `sardana.pool.poolcontroller.PoolBaseController`

Controller class mediator for sardana controller plugins

serialize(**args, **kwargs*)

re_init()

get_ctrl_types()

is_timerable()

is_pseudo()

is_online()

get_ctrl()

set_ctrl(*ctrl*)

ctrl
actual controller object

get_ctrl_info()

ctrl_info
controller information object

set_operator(*operator*)
Defines the current operator object for this controller. For example, in acquisition, it should be a `PoolMeasurementGroup` object.

Parameters **operator** (*object*[580]) – the new operator object

get_operator()

operator
current controller operator

set_log_level(**args*, ***kwargs*)

get_log_level(**args*, ***kwargs*)

get_library_name()

get_class_name()

get_axis_attributes(**args*, ***kwargs*)

get_ctrl_attr(**args*, ***kwargs*)

set_ctrl_attr(**args*, ***kwargs*)

get_axis_attr(**args*, ***kwargs*)

set_axis_attr(**args*, ***kwargs*)

set_ctrl_par(**args*, ***kwargs*)

get_ctrl_par(**args*, ***kwargs*)

set_axis_par(**args*, ***kwargs*)

get_axis_par(**args*, ***kwargs*)

raw_read_axis_states(*axes=None*, *ctrl_states=None*)
Unsafe method. Reads the state for the given axes. If axes is None, reads the state of all active axes.

Parameters **axes** (*seq<int> or None*[581]) – the list of axis to get the state. Default is None meaning all active axis in this controller

Returns a tuple of two elements: a map containing the controller state information for each axis and a boolean telling if an error occured

[580] https://docs.python.org/dev/library/functions.html#object
[581] https://docs.python.org/dev/library/constants.html#None

Return type dict<PoolElement, state info>, bool[582]

read_axis_states(**args, **kwargs*)
Reads the state for the given axes. If axes is None, reads the state of all active axes.

Parameters **axes** (*seq<int> or None*[583]) – the list of axis to get the state. Default is None meaning all active axis in this controller

Returns a map containing the controller state information for each axis

Return type dict<PoolElement, state info>

raw_read_axis_values(*axes=None, ctrl_values=None*)
Unsafe method. Reads the value for the given axes. If axes is None, reads the value of all active axes.

Parameters **axes** (*seq<int> or None*[584]) – the list of axis to get the value. Default is None meaning all active axis in this controller

Returns a map containing the controller value information for each axis

Return type dict<PoolElement, SardanaValue>

read_axis_values(**args, **kwargs*)
Reads the value for the given axes. If axes is None, reads the value of all active axes.

Parameters **axes** (*seq<int> or None*[585]) – the list of axis to get the value. Default is None meaning all active axis in this controller

Returns a map containing the controller value information for each axis

Return type dict<PoolElement, SardanaValue>

stop_axes(*axes*)
Stops the given axes.

Parameters **axes** (*list<axes>*) – the list of axes to stopped.

Returns list of axes that could not be stopped

Return type list<int>

stop_element(**args, **kwargs*)
Stops the given element.

Parameters **element** (PoolElement) – the list of elements to stop

Raises **Exception**[586] – not able to stop element

stop_elements(**args, **kwargs*)
Stops the given elements. If elements is None, stops all active elements.

Parameters **elements** (*seq<PoolElement> or None*[587]) – the list of elements to stop. Default is None meaning all active elements in this controller

Returns list of elements that could not be stopped

Return type list<PoolElements>

[582] https://docs.python.org/dev/library/functions.html#bool
[583] https://docs.python.org/dev/library/constants.html#None
[584] https://docs.python.org/dev/library/constants.html#None
[585] https://docs.python.org/dev/library/constants.html#None
[586] https://docs.python.org/dev/library/exceptions.html#Exception
[587] https://docs.python.org/dev/library/constants.html#None

stop(**args*, ***kwargs*)
Stops the given elements. If elements is None, stops all active elements.

Parameters **elements** (*seq<PoolElement> or None*[588]) – the list of elements to stop. Default is None meaning all active elements in this controller

Returns list of elements that could not be stopped

Return type list<PoolElements>

abort_axes(**args*, ***kwargs*)
Aborts the given axes.

Parameters **axes** (*list<axes>*) – the list of axes to aborted.

Returns list of axes that could not be aborted

Return type list<int>

abort_element(**args*, ***kwargs*)
Aborts the given elements.

Parameters **element** (*PoolElement*) – the list of elements to abort

Raises **Exception**[589] – not able to abort element

abort_elements(**args*, ***kwargs*)
Abort the given elements. If elements is None, stops all active elements.

Parameters **elements** (*seq<PoolElement> or None*[590]) – the list of elements to stop. Default is None meaning all active elements in this controller

Returns list of elements that could not be aborted

Return type list<PoolElements>

abort(**args*, ***kwargs*)
Abort the given elements. If elements is None, stops all active elements.

Parameters **elements** (*seq<PoolElement> or None*[591]) – the list of elements to stop. Default is None meaning all active elements in this controller

Returns list of elements that could not be aborted

Return type list<PoolElements>

emergency_break(**args*, ***kwargs*)
Stops the given elements. If elements is None, stops all active elements. If stop could not be executed, an abort is attempted.

Parameters **elements** – the list of elements to stop. Default is None meaning all active elements in this controller

Returns elements that could neither be stopped nor aborted

Return type list<PoolElement>

send_to_controller(**args*, ***kwargs*)

raw_move(*axis_pos*)

move(**args*, ***kwargs*)

[588] https://docs.python.org/dev/library/constants.html#None
[589] https://docs.python.org/dev/library/exceptions.html#Exception
[590] https://docs.python.org/dev/library/constants.html#None
[591] https://docs.python.org/dev/library/constants.html#None

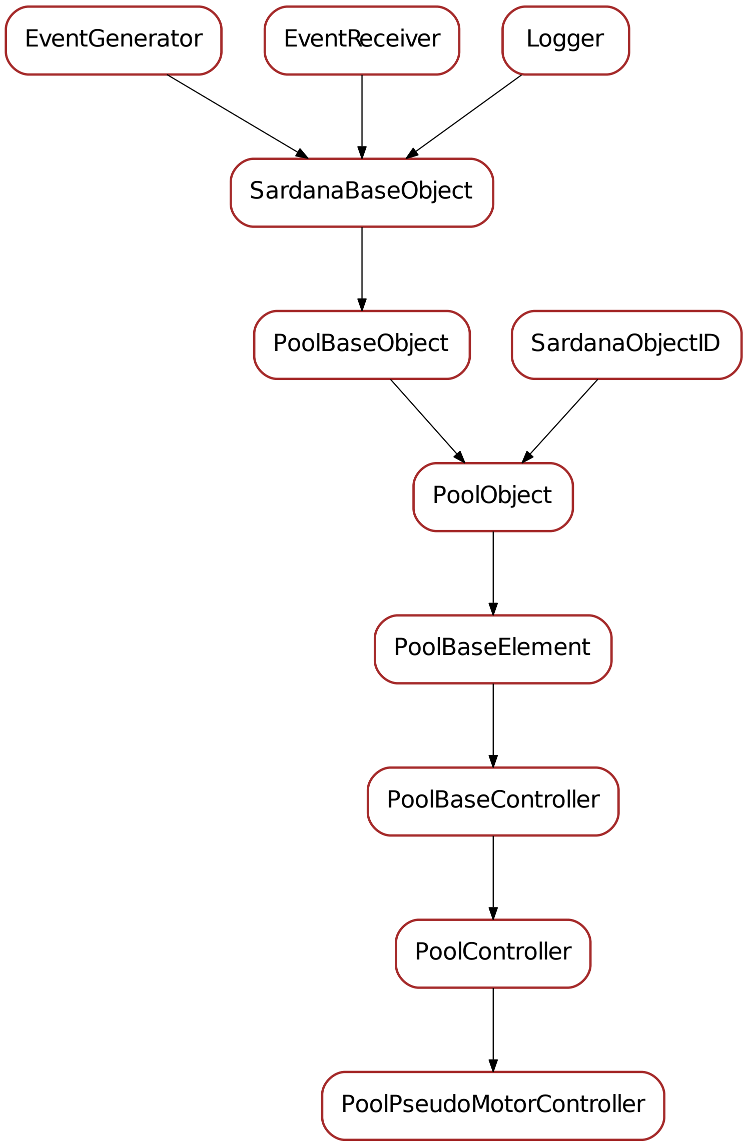
has_backlash()

wants_rounding()

define_position(**args, **kwargs*)

write_one(*axis, value*)

PoolPseudoMotorController



class PoolPseudoMotorController(***kwargs*)

Bases: `sardana.pool.poolcontroller.PoolController`

`serialize`(**args*, ***kwargs*)

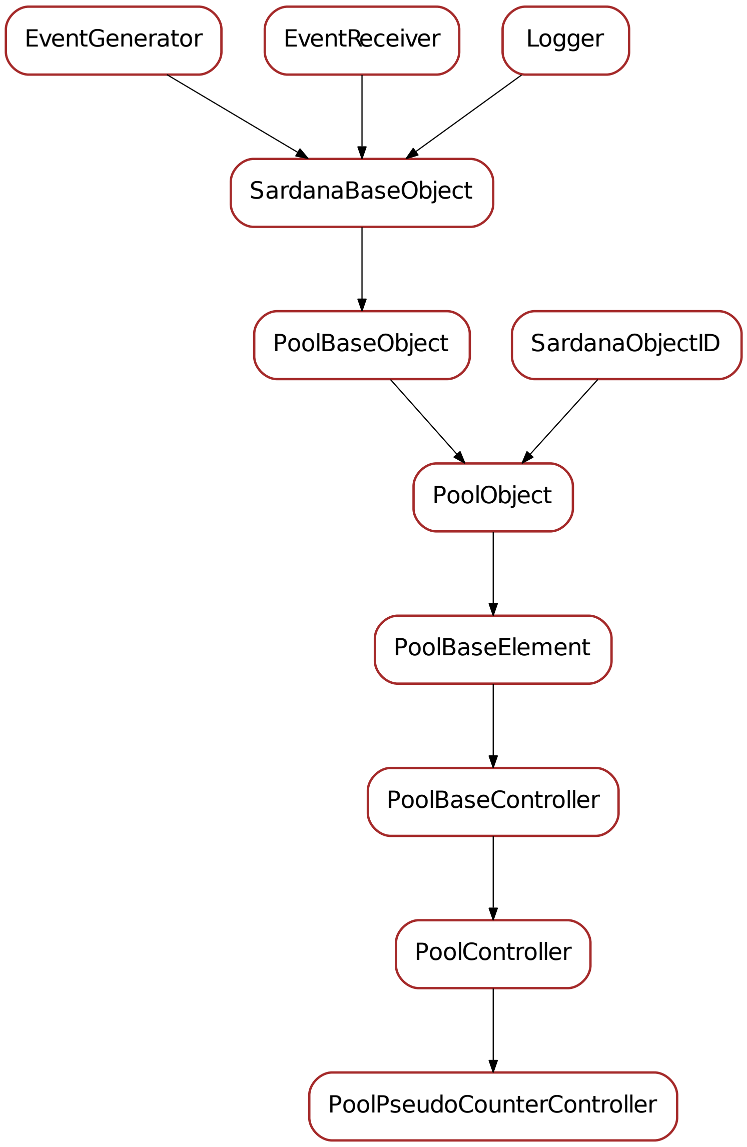
`calc_all_pseudo`(**args*, ***kwargs*)

`calc_all_physical`(**args*, ***kwargs*)

`calc_pseudo`(**args*, ***kwargs*)

`calc_physical`(**args*, ***kwargs*)

PoolPseudoCounterController



class PoolPseudoCounterController(***kwargs*)

Bases: *sardana.pool.poolcontroller.PoolController*

serialize(**args, **kwargs*)

calc(**args, **kwargs*)

calc_all(*values*)

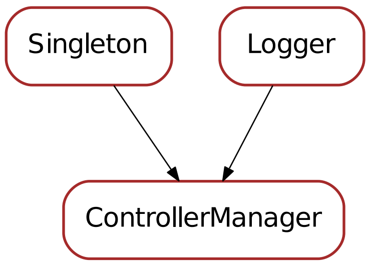
poolcontrollermanager

This module is part of the Python Pool library. It defines the class which controls finding, loading/unloading of device pool controller plug-ins.

Classes

- *ControllerManager*

ControllerManager



class ControllerManager

Bases: `taurus.core.util.singleton.Singleton`, `taurus.core.util.log.Logger`

The singleton class responsible for managing controller plug-ins.

DEFAULT_CONTROLLER_DIRECTORIES = ('poolcontrollers',)

init (**args, **kwargs*)
Singleton instance initialization.

reInit ()
Singleton re-initialization.

cleanUp ()
Singleton clean up.

set_pool (*pool*)

get_pool ()

setControllerPath (*controller_path, reload=True*)
Registers a new list of controller directories in this manager.

Parameters **controller_path** (*seq<str>*) – a sequence of absolute paths where this manager should look for controllers

> **Warning:** as a consequence all the controller modules will be reloaded. This means that if any reference to an old controller object was kept it will refer to an old module (which could possibly generate problems of type class A != class A).

getControllerPath()
Returns the current sequence of absolute paths used to look for controllers.

Returns sequence of absolute paths

Return type seq<str>

getOrCreateControllerLib(*lib_name*, *controller_name=None*)
Gets the exiting controller lib or creates a new controller lib file. If name is not None, a controller template code for the given controller name is appended to the end of the file.

Parameters

- **lib_name** (*str*[592]) – module name, python file name, or full file name (with path)
- **controller_name** (*str*[593]) – an optional controller name. If given a controller template code is appended to the end of the file [default: None, meaning no controller code is added)

Returns a sequence with three items: full_filename, code, line number line number is 0 if no controller is created or n representing the first line of code for the given controller name.

Return type tuple<str, str[594], int>

setControllerLib(*lib_name*, *code*)
Creates a new controller library file with the given name and code. The new module is imported and becomes imediately available.

Parameters

- **lib_name** (*str*[595]) – name of the new library
- **code** (*str*[596]) – python code of the new library

createControllerLib(*lib_name*, *path=None*)
Creates a new empty controller library (python module)

createController(*lib_name*, *controller_name*)
Creates a new controller

reloadController(*controller_name*, *path=None*)
Reloads the module corresponding to the given controller name

Raises *sardana.pool.poolexception.UnknownController* in case the controller is unknown or ImportError[597] if the reload process is not successfull

Parameters

- **controller_name** (*str*[598]) – controller class name

[592] https://docs.python.org/dev/library/stdtypes.html#str
[593] https://docs.python.org/dev/library/stdtypes.html#str
[594] https://docs.python.org/dev/library/stdtypes.html#str
[595] https://docs.python.org/dev/library/stdtypes.html#str
[596] https://docs.python.org/dev/library/stdtypes.html#str
[597] https://docs.python.org/dev/library/exceptions.html#ImportError
[598] https://docs.python.org/dev/library/stdtypes.html#str

- **path** (*seq<str>*) – a list of absolute path to search for libraries [default: None, meaning the current ControllerPath will be used]

reloadControllers(*controller_names*, *path=None*)
Reloads the modules corresponding to the given controller names

Raises *sardana.pool.poolexception.UnknownController* in case the controller is unknown or ImportError[599] if the reload process is not successful

Parameters

- **controller_names** (*seq<str>*) – a list of controller class names
- **path** (*seq<str>*) – a list of absolute path to search for libraries [default: None, meaning the current ControllerPath will be used]

reloadControllerLibs(*module_names*, *path=None*, *reload=True*)
Reloads the given library(=module) names

Raises *sardana.pool.poolexception.UnknownController* in case the controller is unknown or ImportError[600] if the reload process is not successful

Parameters

- **module_names** (*seq<str>*) – a list of module names
- **path** (*seq<str>*) – a list of absolute path to search for libraries [default: None, meaning the current ControllerPath will be used]

reloadControllerLib(*module_name*, *path=None*, *reload=True*)
Reloads the given library(=module) names

Raises *sardana.pool.poolexception.UnknownController* in case the controller is unknown or ImportError[601] if the reload process is not successful

Parameters

- **module_name** (*str*[602]) – controller library name (=python module name)
- **path** (*seq<str>*) – a list of absolute path to search for libraries [default: None, meaning the current ControllerPath will be used]

Returns the ControllerLib object for the reloaded controller lib

Return type *sardana.pool.poolmetacontroller.ControllerLibrary*

addController(*controller_lib*, *klass*)
Adds a new controller class

getControllerNames()

getControllerLibNames()

getControllerLibs(*filter=None*)

getControllers(*filter=None*)

getControllerMetaClass(*controller_name*)

getControllerMetaClasses(*controller_names*)

getControllerLib(*name*)

[599] https://docs.python.org/dev/library/exceptions.html#ImportError
[600] https://docs.python.org/dev/library/exceptions.html#ImportError
[601] https://docs.python.org/dev/library/exceptions.html#ImportError
[602] https://docs.python.org/dev/library/stdtypes.html#str

getControllerClass(*controller_name*)

decodeControllerParameters(*in_par_list*)

strControllerParamValues(*par_list*)
Creates a short string representation of the parameter values list.

Parameters **par_list** (*list<str>*) – list of strings representing the parameter values.

Returns a list containning an abreviated version of the par_list argument.

Return type list<str>

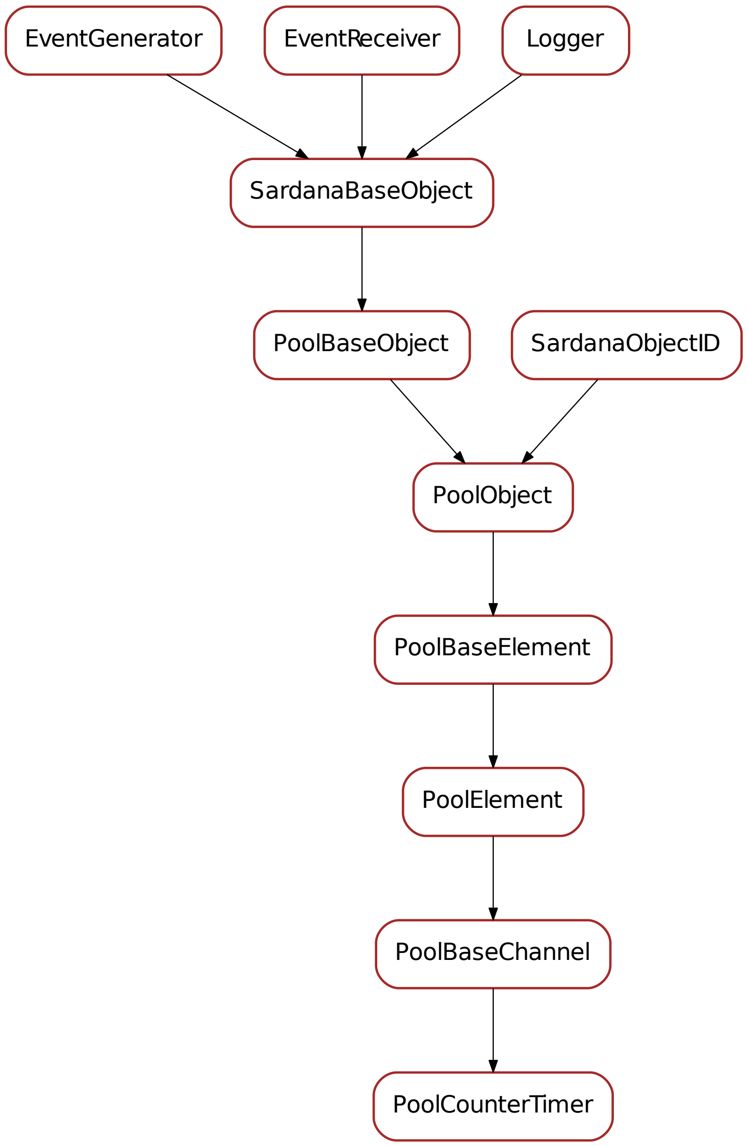
poolcountertimer

This module is part of the Python Pool library. It defines the base classes for CounterTimer

Classes

- *PoolCounterTimer*

PoolCounterTimer



class PoolCounterTimer(***kwargs*)
Bases: *sardana.pool.poolbasechannel.PoolBaseChannel*

set_write_value(*w_value*, *timestamp=None*, *propagate=1*)
Sets a new write value for the value.

Parameters

- **w_value** (Number[603]) – the new write value for value

[603] https://docs.python.org/dev/library/numbers.html#numbers.Number

- **propagate** (*int*[604]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

pooldefs

This file contains the basic pool definitions.

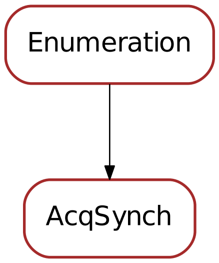
Constants

ControllerAPI = 1.1
A constant defining the controller API version currently supported

Classes

- *AcqSynch*
- *SynchParam*
- *SynchDomain*

AcqSynch



class AcqSynch(**a, **kw*)
Bases: `taurus.core.util.enumeration.Enumeration`

SoftwareTrigger = 0

HardwareTrigger = 1

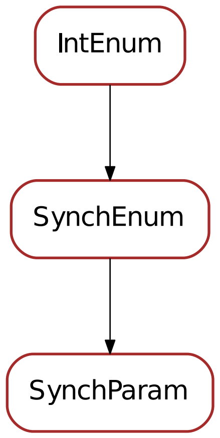
SoftwareGate = 2

HardwareGate = 3

classmethod from_synch_type(*software, synch_type*)
Helper obtain AcqSynch from information about software/hardware nature of synchronization element and AcqSynchType

[604] https://docs.python.org/dev/library/functions.html#int

SynchParam



class SynchParam(**a, **kw*)

Bases: `sardana.pool.pooldefs.SynchEnum`

Enumeration of synchronization's group parameters.

- Delay - initial delay (relative to the synchronization start)
- Total - total interval
- Active - active interval (part of the total interval)
- Repeats - number of repetitions within the group
- Initial - initial point (absolute)

Note: The SynchParam class has been included in Sardana on a provisional basis. Backwards incompatible changes (up to and including removal of the class) may occur if deemed necessary by the core developers.

Delay = 0

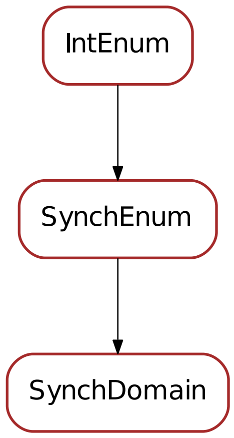
Total = 1

Active = 2

Repeats = 3

Initial = 4

SynchDomain



class SynchDomain(**a*, ***kw*)

Bases: `sardana.pool.pooldefs.SynchEnum`

Enumeration of synchronization domains.

- Time - describes the synchronization in time domain
- Position - describes the synchronization in position domain
- Monitor - not used at the moment but foreseen for synchronization on monitor

Note: The SynchDomain class has been included in Sardana on a provisional basis. Backwards incompatible changes (up to and including removal of the class) may occur if deemed necessary by the core developers.

Time = 0

Position = 1

Monitor = 2

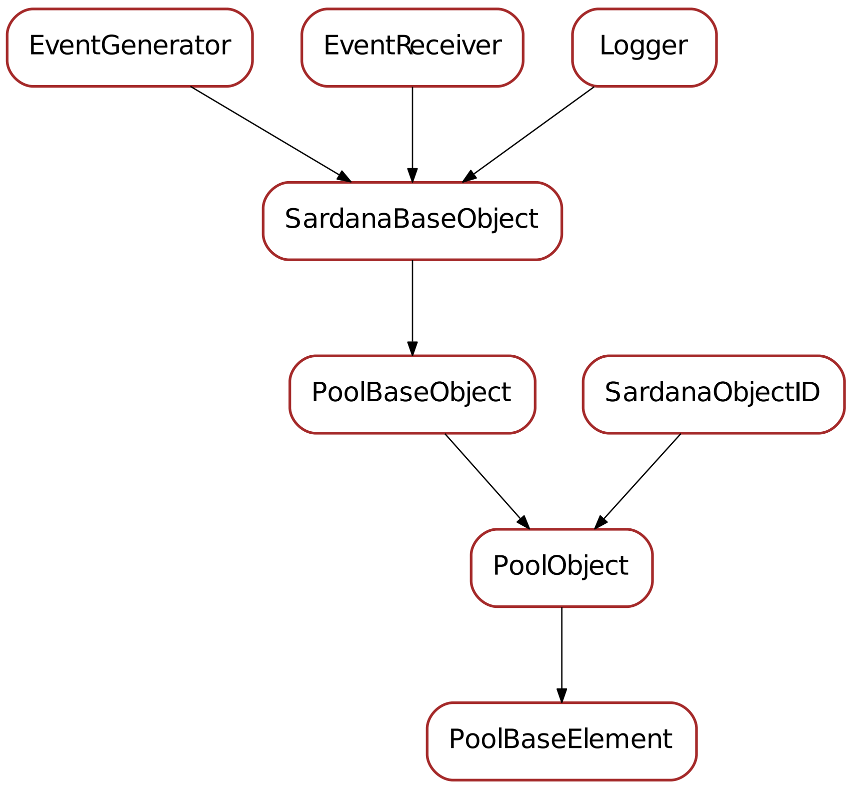
`poolelement`

This module is part of the Python Pool library. It defines the base classes for

Classes

- *PoolBaseElement*
- *PoolElement*

PoolBaseElement



class PoolBaseElement(***kwargs*)

Bases: `sardana.pool.poolobject.PoolObject`

A Pool object that besides the name, reference to the pool, ID, full_name and user_full_name has:

- _simulation_mode : boolean telling if in simulation mode
- _state : element state
- _status : element status

lock(*blocking=True*)

Acquires the this element lock

Parameters blocking (`bool`[605]) – whether or not to block if lock is already acquired [default: True]

unlock()

get_action_cache()

Returns the internal action cache object

serialize(**args*, ***kwargs*)

[605] https://docs.python.org/dev/library/functions.html#bool

get_simulation_mode(*cache=True*, *propagate=1*)
Returns the simulation mode for this object.

Parameters

- **cache** (*bool*[606]) – not used [default: True]
- **propagate** (*int*[607]) – [default: 1]

Returns the current simulation mode

Return type bool[608]

set_simulation_mode(*simulation_mode*, *propagate=1*)

put_simulation_mode(*simulation_mode*)

simulation_mode
element simulation mode

get_state(*cache=True*, *propagate=1*)
Returns the state for this object. If cache is True (default) it returns the current state stored in cache (it will force an update if cache is empty). If propagate > 0 and if the state changed since last read, it will propagate the state event to all listeners.

Parameters

- **cache** (*bool*[609]) – tells if return value from local cache or update from HW read [default: True]
- **propagate** (*int*[610]) – if > 0 propagates the event in case it changed since last HW read. Values bigger that mean the event if sent should be a priority event [default: 1]

Returns the current object state

Return type `sardana.State`

inspect_state()
Looks at the current cached value of state

Returns the current object state

Return type `sardana.State`

set_state(*state*, *propagate=1*)

put_state(*state*)

state
element state

inspect_status()
Looks at the current cached value of status

Returns the current object status

Return type str[611]

[606] https://docs.python.org/dev/library/functions.html#bool
[607] https://docs.python.org/dev/library/functions.html#int
[608] https://docs.python.org/dev/library/functions.html#bool
[609] https://docs.python.org/dev/library/functions.html#bool
[610] https://docs.python.org/dev/library/functions.html#int
[611] https://docs.python.org/dev/library/stdtypes.html#str

get_status(*cache=True*, *propagate=1*)
Returns the status for this object. If cache is True (default) it returns the current status stored in cache (it will force an update if cache is empty). If propagate > 0 and if the status changed since last read, it will propagate the status event to all listeners.

Parameters

- **cache** (*bool*[612]) – tells if return value from local cache or update from HW read [default: True]
- **propagate** (*int*[613]) – if > 0 propagates the event in case it changed since last HW read. Values bigger that mean the event if sent should be a priority event [default: 1]

Returns the current object status

Return type str[614]

set_status(*status*, *propagate=1*)

put_status(*status*)

status
element status

calculate_state_info(*status_info=None*)
Transforms the given state information. This specific base implementation transforms the given state,status tuple into a state, new_status tuple where new_status is "*self.name* is *state* plus the given status. It is assumed that the given status comes directly from the controller status information.

Parameters **status_info** (*tuple<State, str>*) – given status information [default: None, meaning use current state status.

Returns a transformed state information

Return type tuple<State, str>

set_state_info(*state_info*, *propagate=1*)

read_state_info()

put_state_info(*state_info*)

get_default_attribute()

get_default_acquisition_channel()

stop()

was_stopped()

abort()

was_aborted()

was_interrupted()
Tells if action ended by an abort or stop

is_action_running()
Determines if the element action is running or not.

612 https://docs.python.org/dev/library/functions.html#bool
613 https://docs.python.org/dev/library/functions.html#int
614 https://docs.python.org/dev/library/stdtypes.html#str

is_in_operation()
Returns True if this element is involved in any operation

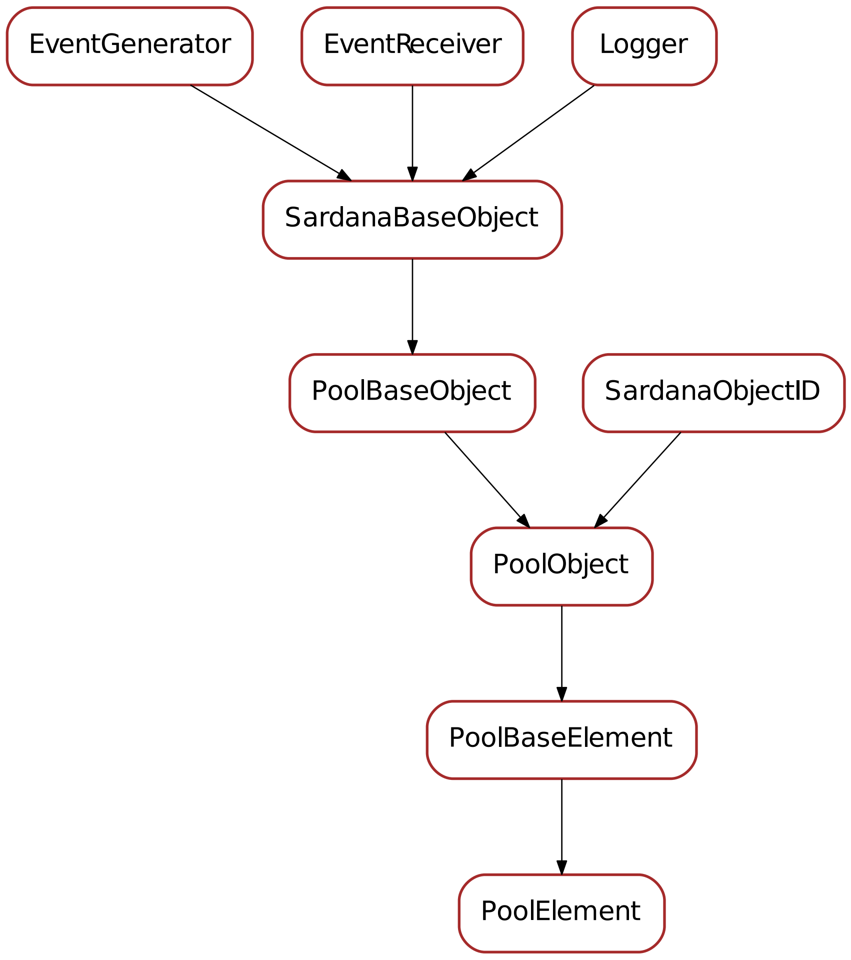
is_in_local_operation()

get_operation()

set_operation(*operation*)

clear_operation()

PoolElement



class PoolElement(***kwargs*)
Bases: `sardana.pool.poolbaseelement.PoolBaseElement`

A Pool element is an Pool object which is controlled by a controller. Therefore it contains a _ctrl_id and a _axis (the id of the element in the controller).

serialize(**args, **kwargs*)

get_parent()
Returns this pool object parent.

Returns this objects parent

Return type *SardanaBaseObject*

get_controller()

get_controller_id()

get_axis()

set_action_cache(*action_cache*)

get_source()

get_instrument()

set_instrument(*instrument, propagate=1*)

stop()

abort()

get_par(*name*)

set_par(*name, value*)

get_extra_par(*name*)

set_extra_par(*name, value*)

axis
element axis

controller
element controller

controller_id
element controller id

instrument
element instrument

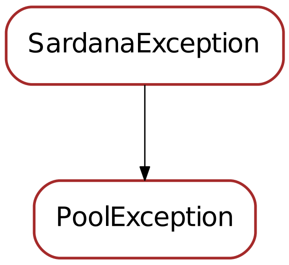
poolexception

This module is part of the Python Pool libray. It defines the base classes for pool exceptions

Classes

- *PoolException*
- *UnknownController*
- *UnknownControllerLibrary*

PoolException

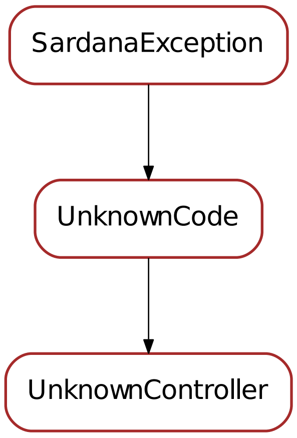


exception PoolException(**args, **kwargs*)

args

message

UnknownController

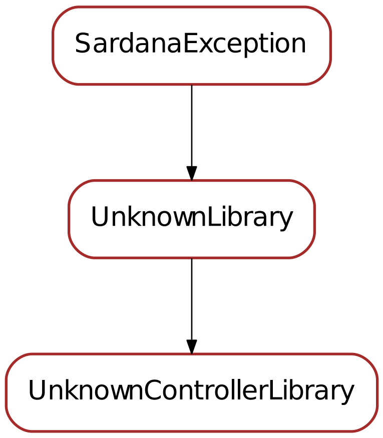


exception UnknownController(**args, **kwargs*)

args

message

UnknownControllerLibrary



exception UnknownControllerLibrary(**args, **kwargs*)

> **args**
>
> **message**

poolexternal

This module is part of the Python Pool libray. It defines the base classes for external objects to the pool (like tango objects)

Functions

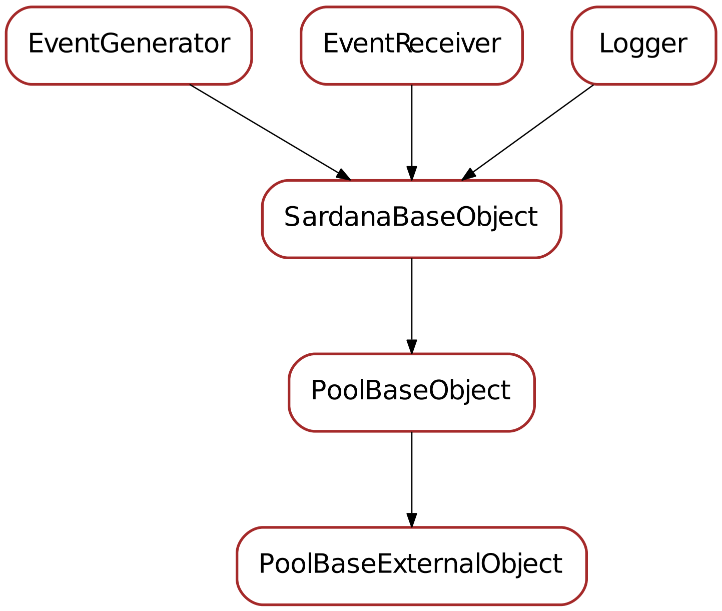
- *PoolExternalObject()*

Classes

- *PoolBaseExternalObject*
- *PoolTangoObject*

PoolExternalObject(***kwargs*)

PoolBaseExternalObject



class PoolBaseExternalObject(***kwargs*)

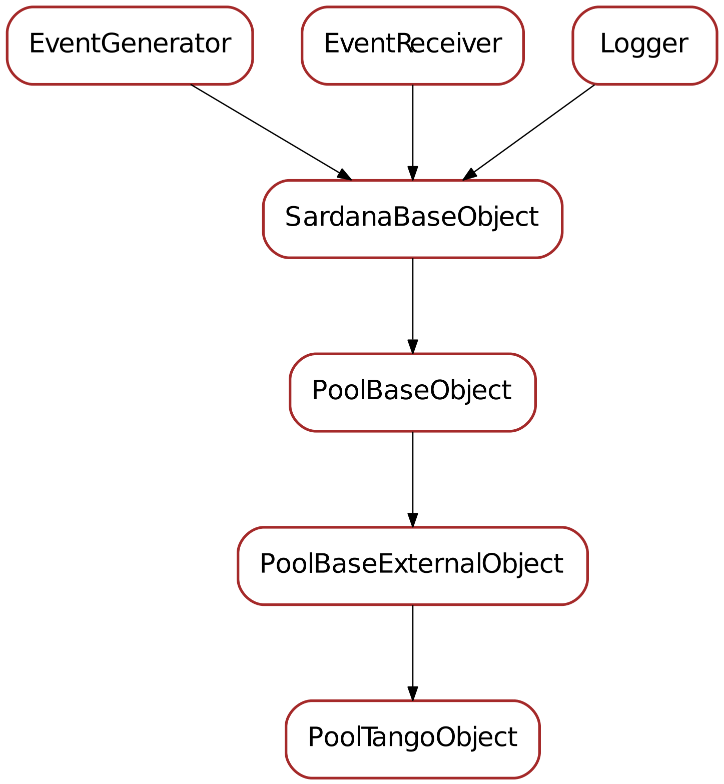
Bases: `sardana.pool.poolbaseobject.PoolBaseObject`

TODO

get_source()

get_config()

PoolTangoObject



class PoolTangoObject(***kwargs*)

Bases: *sardana.pool.poolexternal.PoolBaseExternalObject*

TODO

get_device_name()

get_attribute_name()

get_device()

get_config()

device_name

attribute_name

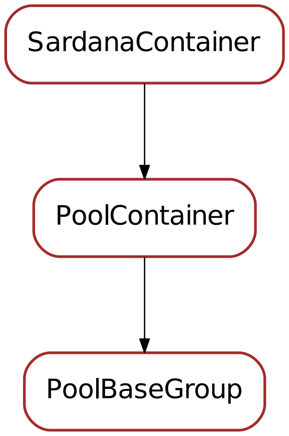
poolgroupelement

This module is part of the Python Pool library. It defines the base classes for

Classes

- *PoolBaseGroup*
- *PoolGroupElement*

PoolBaseGroup



class PoolBaseGroup(***kwargs*)

Bases: *sardana.pool.poolcontainer.PoolContainer*

on_element_changed(*evt_src, evt_type, evt_value*)

set_user_element_ids(*new_element_ids*)

get_user_element_ids()

Returns the sequence of user element IDs

Returns the sequence of user element IDs

Return type sequence< int[615]>

user_element_ids

Returns the sequence of user element IDs

Returns the sequence of user element IDs

Return type sequence< int[616]>

get_user_elements()

Returns the sequence of user elements

Returns the sequence of user elements

Return type sequence< *PoolElement*>

[615] https://docs.python.org/dev/library/functions.html#int
[616] https://docs.python.org/dev/library/functions.html#int

get_user_elements_attribute_iterator()
Returns an iterator over the main attribute of each user element.

Returns an iterator over the main attribute of each user element.

Return type iter< *SardanaAttribute* >

get_user_elements_attribute()
Returns an iterator over the main attribute of each user element.

Returns an iterator over the main attribute of each user element.

Return type iter< *SardanaAttribute* >

get_user_elements_attribute_sequence()
Returns a sequence of main attribute of each user element.

In loops use preferably *get_user_elements_attribute_iterator()* for performance and memory reasons.

Returns a sequence of main attribute of each user element.

Return type sequence< *SardanaAttribute* >

get_user_elements_attribute_map()
Returns a dictionary of main attribute of each user element.

Returns a dictionary of main attribute of each user element.

Return type dict< *PoolElement*, *SardanaAttribute* >

get_physical_elements()
Returns a dictionary or physical elements where key is a controller object and value is a sequence of pool elements

Returns a dictionary of physical elements

Return type dict< PoolElement>

get_physical_elements_iterator()
Returns an iterator over the physical elements.

Warning: The order is non deterministic.

Returns an iterator over the physical elements.

Return type iter<*PoolElement* >

get_physical_elements_attribute_iterator()
Returns an iterator over the main attribute of each physical element.

Warning: The order is non deterministic.

Returns an iterator over the main attribute of each physical element.

Return type iter< *SardanaAttribute* >

get_physical_elements_set()

add_user_element(*element*, *index=None*)

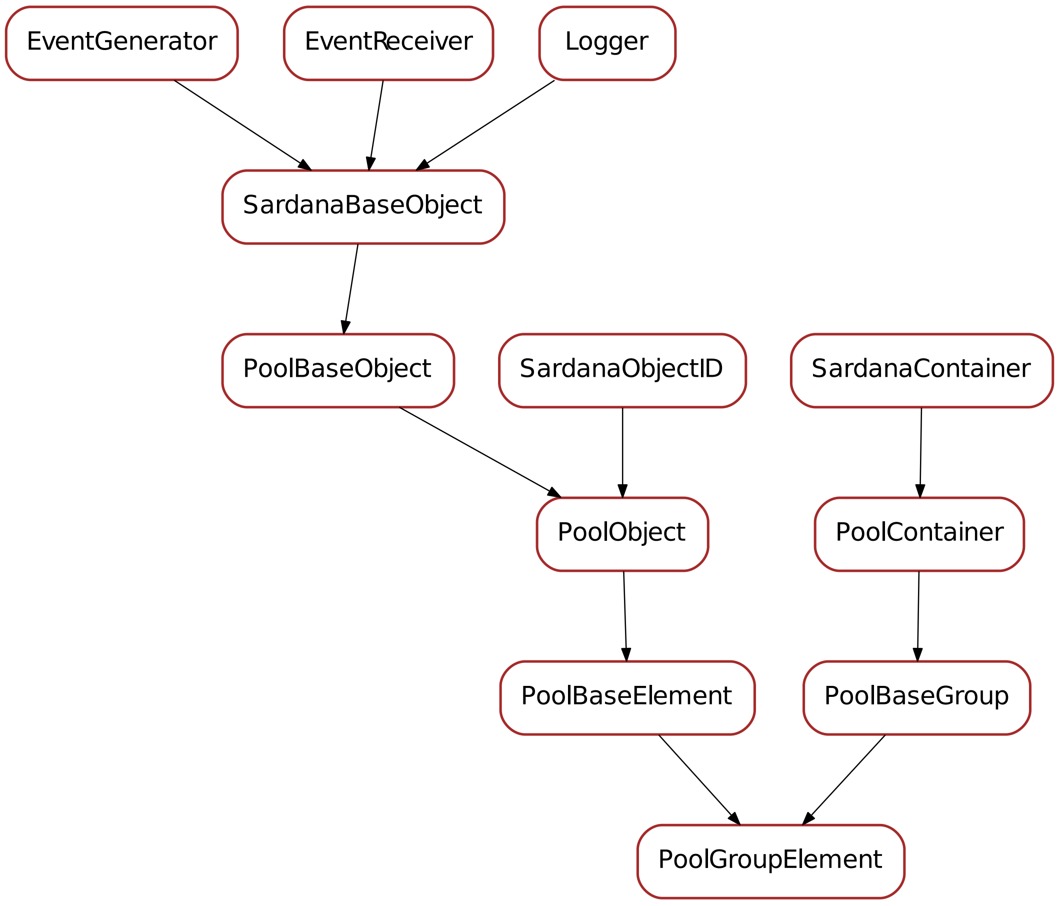
clear_user_elements()

stop()

abort()

get_operation()

PoolGroupElement



class PoolGroupElement(***kwargs*)

Bases: `sardana.pool.poolbaseelement.PoolBaseElement`, `sardana.pool.poolbasegroup.PoolBaseGroup`

serialize(**args*, ***kwargs*)

get_action_cache()

Returns the internal action cache object

set_action_cache(*action_cache*)

read_state_info()

stop()

abort()

get_operation()

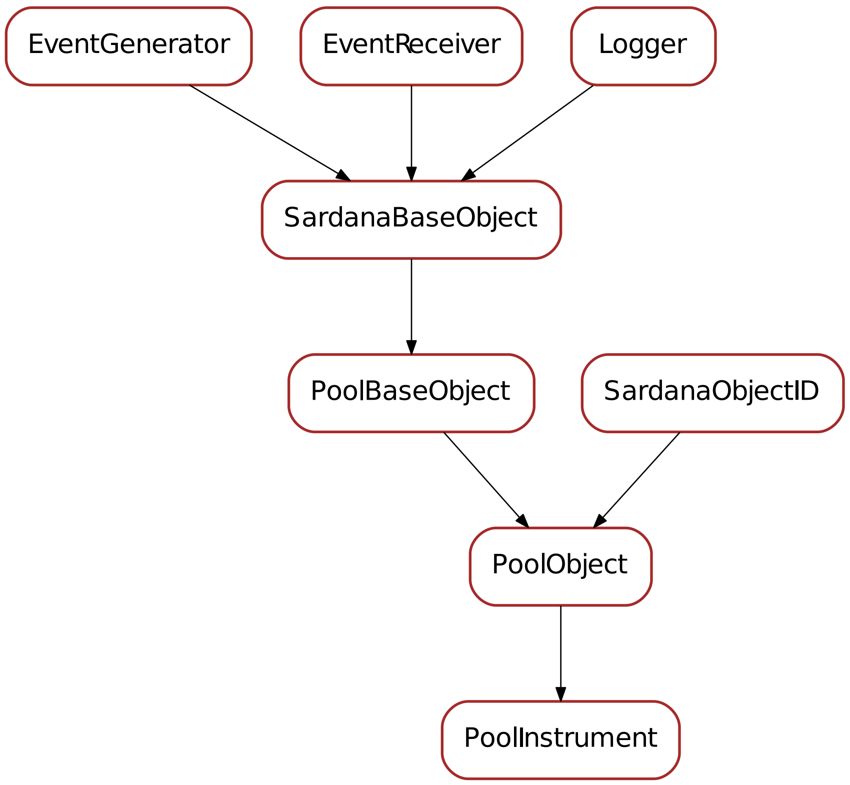
poolinstrument

This module is part of the Python Pool library. It defines the base classes for instrument

Classes

- *PoolInstrument*

PoolInstrument



class PoolInstrument(***kwargs*)
Bases: *sardana.pool.poolobject.PoolObject*

get_parent()
Returns this pool object parent.

Returns this objects parent

Return type *SardanaBaseObject*

serialize(**args*, ***kwargs*)

get_instrument_class()

add_instrument(*instrument*)

remove_instrument(*instrument*)

get_instruments()

set_parent_instrument(*instrument*)

get_parent_instrument()

has_parent_instrument()

add_element(*element*)

remove_element(*element*)

get_elements()

has_instruments()

has_elements()

instruments

elements

instrument_class

parent_instrument

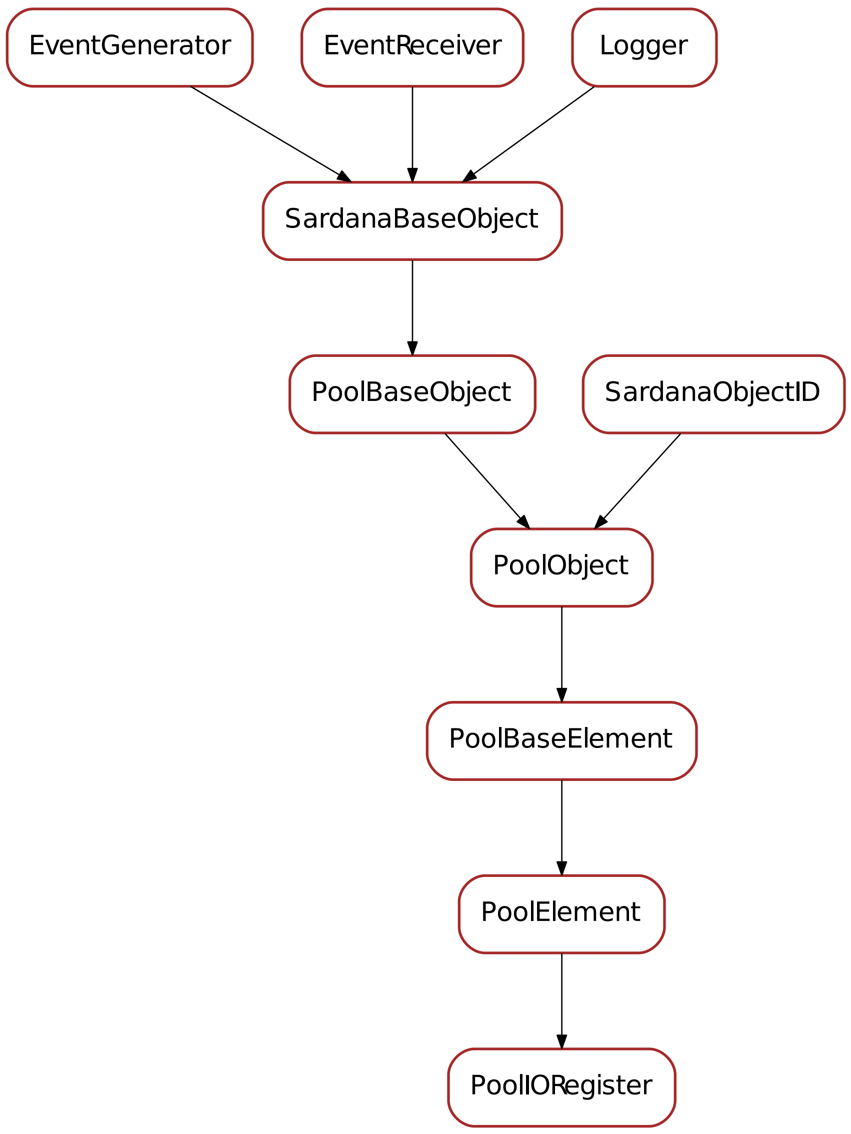
poolioregister

This module is part of the Python Pool libray. It defines the base classes for

Classes

- *PoolIORegister*

PoolIORegister



class PoolIORegister(***kwargs*)
Bases: `sardana.pool.poolelement.PoolElement`

get_value_attribute()
Returns the value attribute object for this IO register

Returns the value attribute

Return type `SardanaAttribute`

on_change(*evt_src*, *evt_type*, *evt_value*)

get_default_attribute()

read_value()
Reads the IO register value from hardware.

Returns a *SardanaValue* containing the IO register value

Return type *SardanaValue*

put_value(*value*, *propagate=1*)
Sets a value.

Parameters

- **value** (*SardanaValue*) – the new value
- **propagate** (*int*[617]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

get_value(*cache=True*, *propagate=1*)

set_value(*value*, *timestamp=None*)

set_write_value(*w_value*, *timestamp=None*, *propagate=1*)
Sets a new write value for the IO registere

Parameters

- **w_value** (Number[618]) – the new write value for IO register
- **propagate** (*int*[619]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

value
ioregister value

write_register(*value*, *timestamp=None*)

poolmeasurementgroup

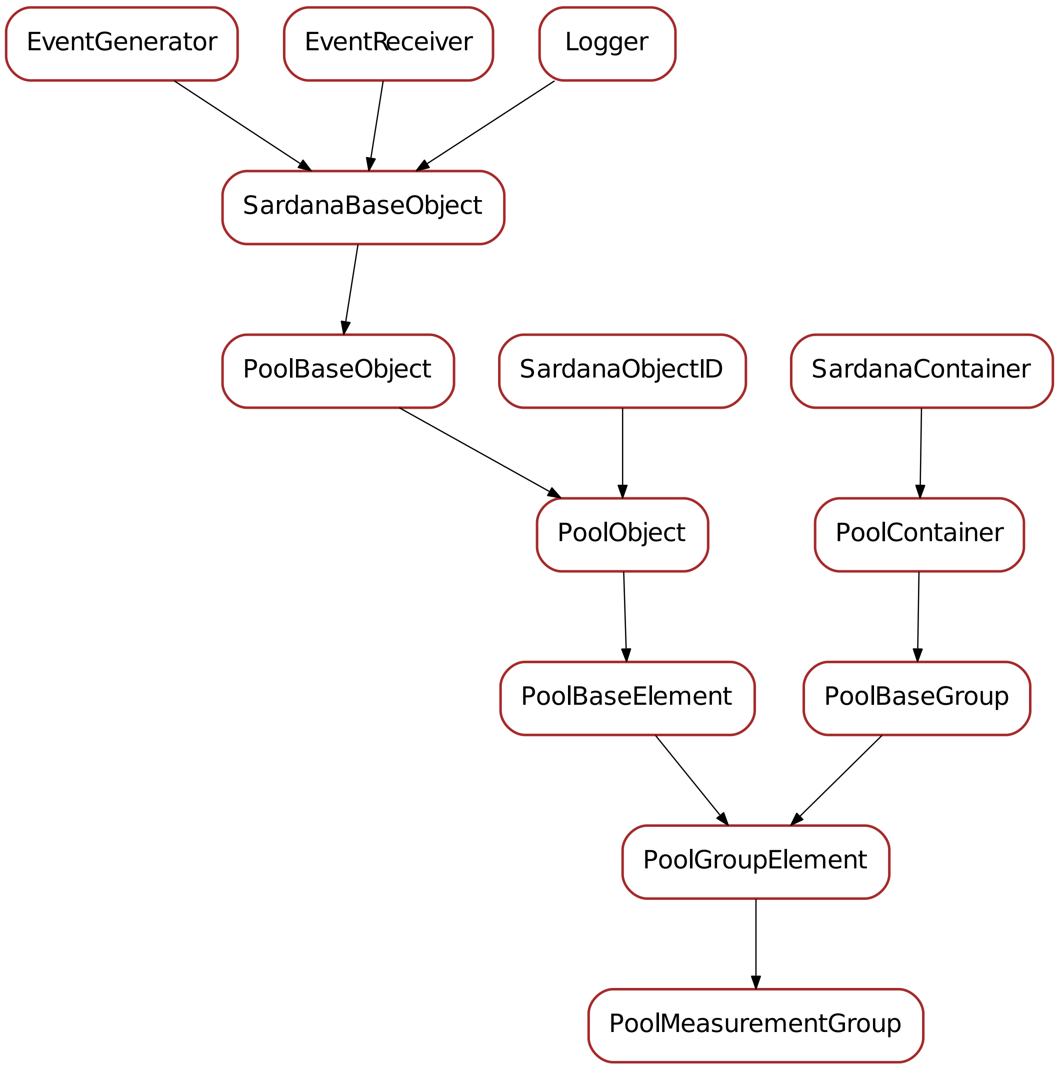
This module is part of the Python Pool library. It defines the base classes for

Classes

- *PoolMeasurementGroup*

[617] https://docs.python.org/dev/library/functions.html#int
[618] https://docs.python.org/dev/library/numbers.html#numbers.Number
[619] https://docs.python.org/dev/library/functions.html#int

PoolInstrument



class PoolMeasurementGroup(***kwargs*)

Bases: `sardana.pool.poolgroupelement.PoolGroupElement`

DFT_DESC = 'General purpose measurement group'

on_element_changed(*evt_src, evt_type, evt_value*)

get_pool_controllers()

get_pool_controller_by_name(*name*)

add_user_element(*element, index=None*)

Override the base behavior, so the TriggerGate elements are silently skipped if used multiple times in the group

set_configuration(*config=None, propagate=1, to_fqdn=True*)

set_configuration_from_user(*cfg, propagate=1, to_fqdn=True*)

get_configuration()

get_user_configuration()

load_configuration(*force=False*)
Loads the current configuration to all involved controllers

get_timer()

timer

get_integration_time()

set_integration_time(*integration_time, propagate=1*)

integration_time
the current integration time

get_monitor_count()

set_monitor_count(*monitor_count, propagate=1*)

monitor_count
the current monitor count

get_acquisition_mode()

set_acquisition_mode(*acquisition_mode, propagate=1*)

acquisition_mode
the current acquisition mode

get_synchronization()

set_synchronization(*synchronization, propagate=1*)

synchronization
the current acquisition mode

get_moveable()

set_moveable(*moveable, propagate=1, to_fqdn=True*)

moveable
moveable source used in synchronization

get_latency_time()

latency_time
latency time between two consecutive acquisitions

start_acquisition(*value=None, multiple=1*)

set_acquisition(*acq_cache*)

get_acquisition()

acquisition
acquisition object

stop()

poolmetacontroller

This module is part of the Python Pool libray. It defines the base classes for

Classes

- *DataInfo*
- *TypeData*
- *ControllerLibrary*
- *ControllerClass*

DataInfo

class DataInfo(*name, dtype, dformat=<_mock._Mock object>, access=<_mock._Mock object>, description='', default_value=None, memorized='true', fget=None, fset=None, maxdimsize=None*)

Bases: object[620]

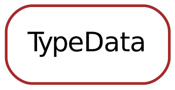
copy()

classmethod toDataInfo(*name, info*)

toDict()

serialize(**args, **kwargs*)

TypeData



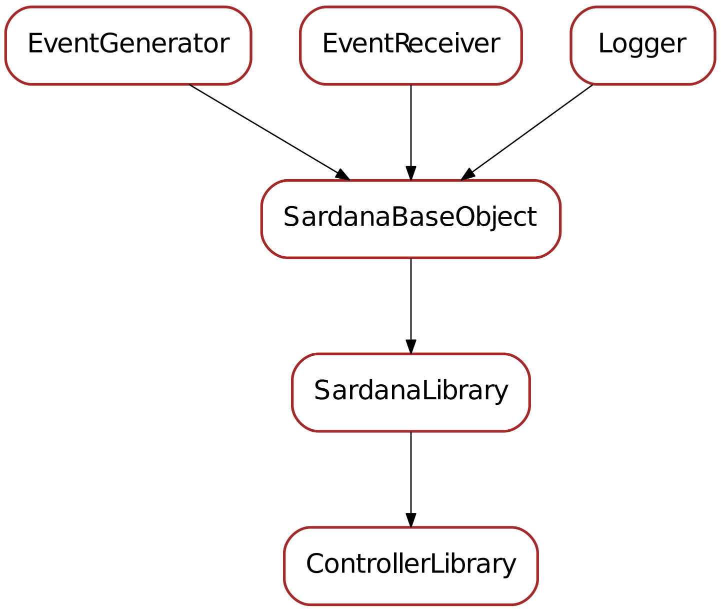
class TypeData(***kwargs*)

Bases: object[621]

[620] https://docs.python.org/dev/library/functions.html#object
[621] https://docs.python.org/dev/library/functions.html#object

Information for a specific Element type

ControllerLib



class ControllerLibrary(***kwargs*)

Bases: `sardana.sardanameta.SardanaLibrary`

Object representing a python module containning controller classes. Public members:

- module - reference to python module
- f_path - complete (absolute) path and filename
- f_name - filename (including file extension)
- path - complete (absolute) path
- name - module name (without file extension)
- controller_list - list<ControllerClass>
- **exc_info - exception information if an error occured when loading** the module

add_controller(*meta_class*)

Adds a new :class:~'sardana.sardanameta.SardanaClass' to this library.

Parameters meta_class (`:class:~`sardana.sardanameta.SardanaClass``) – the meta class to be added to this library

get_controller(*meta_class_name*)

Returns a :class:~'sardana.sardanameta.SardanaClass' for the given meta class name or None if the meta class does not exist in this library.

Parameters **meta_class_name** (*str*[622]) – the meta class name

Returns a meta class or None

Return type :class:~'sardana.sardanameta.SardanaClass'

get_controllers()

Returns a sequence of the meta classes that belong to this library.

Returns a sequence of meta classes that belong to this library

Return type seq<:class:~'sardana.sardanameta.SardanaClass'>

has_controller(*meta_class_name*)

Returns True if the given meta class name belongs to this library or False otherwise.

Parameters **meta_class_name** (*str*[623]) – the meta class name

Returns True if the given meta class name belongs to this library or False otherwise

Return type bool[624]

serialize(**args*, ***kwargs*)

Returns a serializable object describing this object.

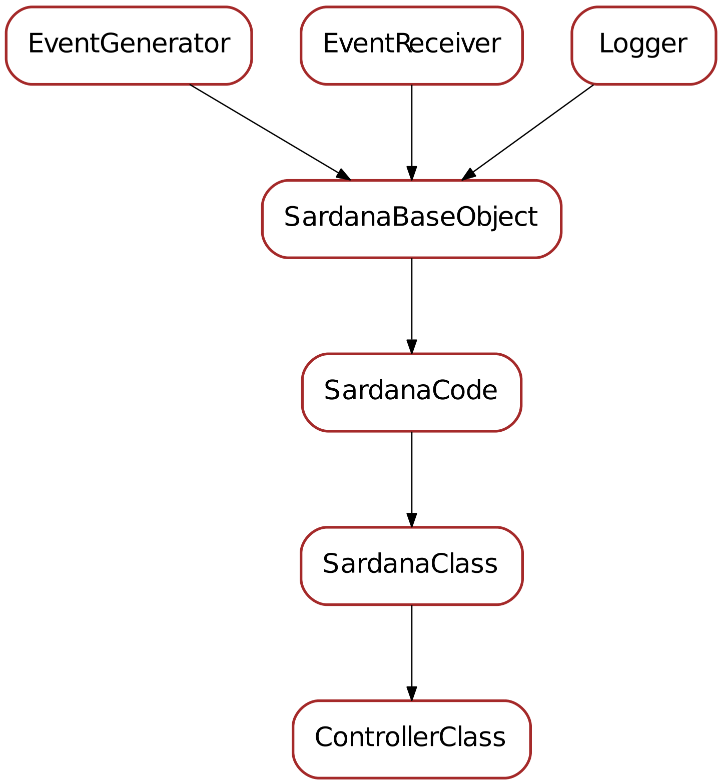
Returns a serializable dict

Return type dict[625]

controllers

[622] https://docs.python.org/dev/library/stdtypes.html#str
[623] https://docs.python.org/dev/library/stdtypes.html#str
[624] https://docs.python.org/dev/library/functions.html#bool
[625] https://docs.python.org/dev/library/stdtypes.html#dict

ControllerClass



class ControllerClass(***kwargs*)

Bases: `sardana.sardanameta.SardanaClass`

Object representing a python controller class. Public members:

- name - class name
- klass - python class object
- lib - ControllerLibrary object representing the module where the controller is.

serialize(**args, **kwargs*)

Returns a serializable object describing this object.

Returns a serializable dict

Return type dict[626]

controller_class

gender

model

[626] https://docs.python.org/dev/library/stdtypes.html#dict

organization

Constants

CONTROLLER_TEMPLATE = 'class @controller_name@(@controller_type@):\n """@controller_name@ c
String containing template code for a controller class

CTRL_TYPE_MAP = {<_mock._Mock object at 0x7f07d78a0050>: <class 'sardana.pool.poolcontrol
a dictionary dict<`ElementType`, class> mapping element type enumeration with the corresponding controller pool class (*`PoolController`* or sub-class of it).

TYPE_MAP = {<_mock._Mock object at 0x7f07d7880d90>: ('IORegister', 'IORegister', <class 's
dictionary dict<`ElementType`, `tuple`[627]> where tuple is a sequence:

1. type string representation
2. family
3. internal pool class
4. automatic full name
5. controller class

TYPE_MAP_OBJ = {<_mock._Mock object at 0x7f07d7880d90>: <sardana.pool.poolmetacontroller.T
dictionary dict<`ElementType`, *`TypeData`*>

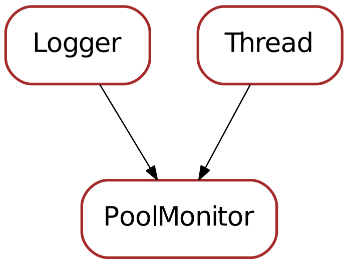
poolmonitor

This file contains the pool monitor class

Classes

- *`PoolMonitor`*

PoolMonitor



[627] https://docs.python.org/dev/library/stdtypes.html#tuple

class PoolMonitor(*pool, name='PoolMonitor', period=5.0, min_sleep=1.0, auto_start=True*)
Bases: `taurus.core.util.log.Logger`, `threading.Thread`[629]

MIN_THREADS = 1

MAX_THREADS = 10

on_pool_changed(*evt_src, evt_type, evt_value*)

update_state_info()
Update state information of every element.

stop()

pause()

resume()

monitor()

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

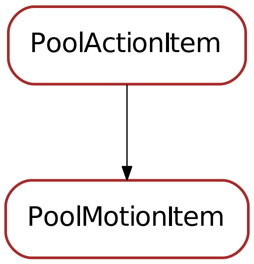
poolmotion

This module is part of the Python Pool libray. It defines the class for a motion

Classes

- *PoolMotionItem*
- *PoolMotion*

PoolMotionItem



[629] https://docs.python.org/dev/library/threading.html#threading.Thread

class PoolMotionItem(*moveable, position, dial_position, do_backlash, backlash, instability_time=None*)

Bases: `sardana.pool.poolaction.PoolActionItem`

An item involved in the motion. Maps directly to a motor object

has_instability_time()

in_motion()

get_moveable()

moveable

get_state_info()

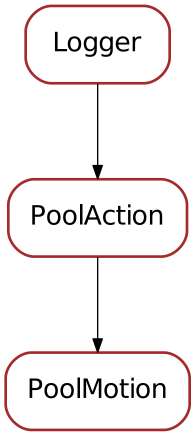
start(*new_state*)

stopped(*timestamp*)

handle_instability(*timestamp*)

on_state_switch(*state_info, timestamp=None*)

PoolMotion



class PoolMotion(*main_element, name='GlobalMotion'*)

Bases: `sardana.pool.poolaction.PoolAction`

This class manages motion actions

pre_start_all(*pool_ctrls*)

pre_start_one(*moveables, items*)

start_one(*moveables, motion_info*)

start_all(*pool_ctrls, moveables, motion_info*)

start_action (**args, **kwargs*)
kwargs['items'] is a dict<moveable, (pos, dial, do_backlash, backlash)

backlash_item (*motion_item*)

action_loop

read_dial_position (*ret=None, serial=False*)

raw_read_dial_position (*ret=None, serial=False*)

Enumerations

MotionState = <taurus.core.util.enumeration.Enumeration object>

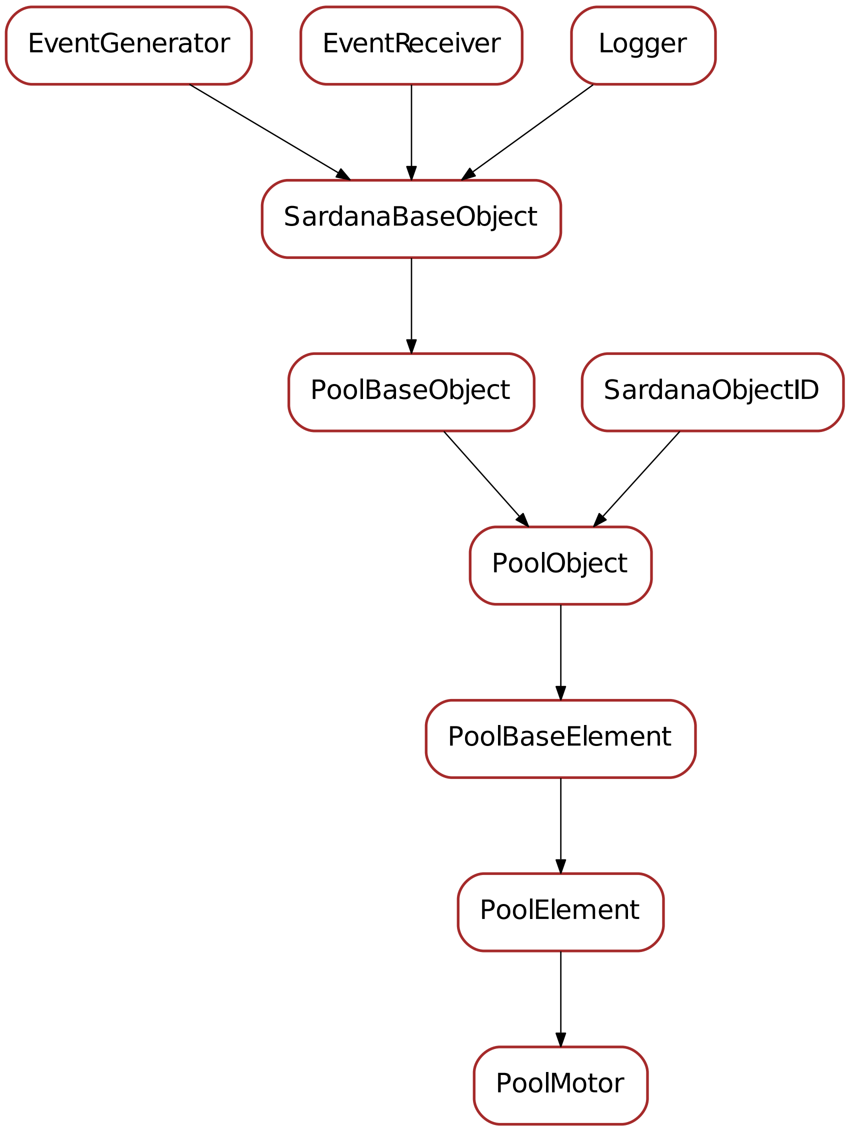
poolmotor

This module is part of the Python Pool libray. It defines the base classes for

Classes

- *PoolMotor*

PoolMotor



class PoolMotor(***kwargs*)

Bases: `sardana.pool.poolelement.PoolElement`

An internal Motor object. **NOT** part of the official API. Accessing this object from a controller plug-in may lead to undetermined behavior like infinite recursion.

on_change(*evt_src, evt_type, evt_value*)

calculate_state_info(*state_info=None*)

Transforms the given state information. This specific base implementation transforms the given state,status tuple into a state, new_status tuple where new_status is "*self.name* is *state* plus the given status. It is assumed that the given status comes directly from the controller status information.

Parameters **status_info** (*tuple<State, str>*) – given status information [default: None, meaning use current state status.

Returns a transformed state information

Return type tuple<State, str>

inspect_limit_switches()
returns the current (cached value of the limit switches

Returns the current limit switches flags

get_limit_switches(*cache=True*, *propagate=1*)
Returns the motor limit switches state.

Parameters

- **cache** (*bool*[630]) – if True (default) return value in cache, otherwise read value from hardware
- **propagate** (*int*[631]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

Returns the motor limit switches state

Return type *SardanaAttribute*

set_limit_switches(*ls*, *propagate=1*)

put_limit_switches(*ls*, *propagate=1*)

limit_switches
motor limit switches

has_instability_time(*cache=True*)

get_instability_time(*cache=True*)

set_instability_time(*instability_time*, *propagate=1*)

instability_time
motor instability time

has_backlash(*cache=True*)

is_backlash_positive(*cache=True*)

is_backlash_negative(*cache=True*)

get_backlash(*cache=True*)

set_backlash(*backlash*, *propagate=1*)

backlash
motor backlash

get_offset_attribute()

get_offset(*cache=True*)

set_offset(*offset*, *propagate=1*)

[630] https://docs.python.org/dev/library/functions.html#bool
[631] https://docs.python.org/dev/library/functions.html#int

offset
motor offset

get_sign_attribute()

get_sign(*cache=True*)

set_sign(*sign*, *propagate=1*)

sign
motor sign

get_step_per_unit(*cache=True*, *propagate=1*)

set_step_per_unit(*step_per_unit*, *propagate=1*)

read_step_per_unit()

step_per_unit
motor steps per unit

get_acceleration(*cache=True*, *propagate=1*)

set_acceleration(*acceleration*, *propagate=1*)

read_acceleration()

acceleration
motor acceleration

get_deceleration(*cache=True*, *propagate=1*)

set_deceleration(*deceleration*, *propagate=1*)

read_deceleration()

deceleration
motor deceleration

get_base_rate(*cache=True*, *propagate=1*)

set_base_rate(*base_rate*, *propagate=1*)

read_base_rate()

base_rate
motor base rate

get_velocity(*cache=True*, *propagate=1*)

set_velocity(*velocity*, *propagate=1*)

read_velocity()

velocity
motor velocity

define_position(*position*)

get_position_attribute()
Returns the position attribute object for this motor

Returns the position attribute

Return type `SardanaAttribute`

get_position(*cache=True*, *propagate=1*)
Returns the user position.

Parameters

- **cache** (*bool*[632]) – if True (default) return value in cache, otherwise read value from hardware
- **propagate** (*int*[633]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

Returns the user position

Return type *SardanaAttribute*

set_position(*position*)
Moves the motor to the specified user position

Parameters **position** (Number[634]) – the user position to move to

set_write_position(*w_position*, *timestamp=None*, *propagate=1*)
Sets a new write value for the user position.

Parameters

- **w_position** (Number[635]) – the new write value for user position
- **propagate** (*int*[636]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

read_dial_position()
Reads the dial position from hardware.

Returns a *SardanaValue* containing the dial position

Return type *SardanaValue*

put_dial_position(*dial_position_value*, *propagate=1*)
Sets a new dial position.

Parameters

- **dial_position_value** (*SardanaValue*) – the new dial position value
- **propagate** (*int*[637]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

get_dial_position_attribute()
Returns the dial position attribute object for this motor

Returns the dial position attribute

Return type *SardanaAttribute*

get_dial_position(*cache=True*, *propagate=1*)
Returns the dial position.

Parameters

- **cache** (*bool*[638]) – if True (default) return value in cache, otherwise read value from hardware

[632] https://docs.python.org/dev/library/functions.html#bool
[633] https://docs.python.org/dev/library/functions.html#int
[634] https://docs.python.org/dev/library/numbers.html#numbers.Number
[635] https://docs.python.org/dev/library/numbers.html#numbers.Number
[636] https://docs.python.org/dev/library/functions.html#int
[637] https://docs.python.org/dev/library/functions.html#int
[638] https://docs.python.org/dev/library/functions.html#bool

- **propagate** (*int*[639]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

Returns the dial position

Return type *SardanaAttribute*

position
motor user position

dial_position
motor dial position

get_default_attribute()

get_motion()

motion
motion object

calculate_motion(*new_position*, *items=None*, *calculated=None*)
Calculate the motor position, dial position, backlash for the given final position. Items specifies the where to put the calculated values, calculated is not used by physical motors

start_move(*new_position*)

poolmotorgroup

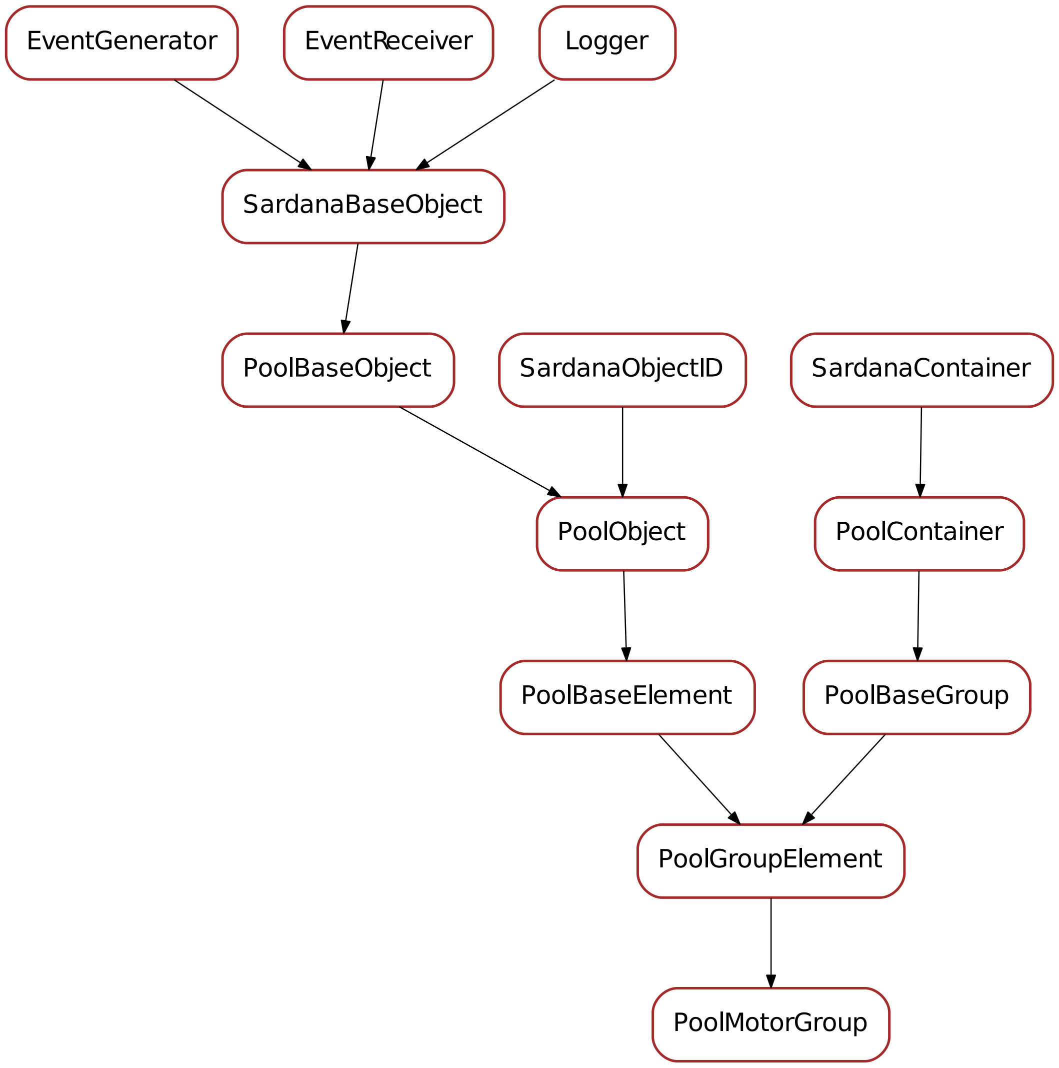
This module is part of the Python Pool library. It defines the base classes for

Classes

- *PoolMotorGroup*

[639] https://docs.python.org/dev/library/functions.html#int

PoolMotorGroup



class PoolMotorGroup(***kwargs*)
Bases: `sardana.pool.poolgroupelement.PoolGroupElement`

on_change(*evt_src, evt_type, evt_value*)

on_element_changed(*evt_src, evt_type, evt_value*)

add_user_element(*element, index=None*)

get_position_attribute()

get_low_level_physical_position_attribute_iterator()

get_physical_position_attribute_iterator()

get_physical_positions_attribute_sequence()

get_physical_positions_attribute_map()

get_position(*cache=True*, *propagate=1*)
Returns the user position.

Parameters

- **cache** (*bool*[640]) – if `True` (default) return value in cache, otherwise read value from hardware
- **propagate** (*int*[641]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

Returns the user position

Return type *SardanaAttribute*

set_position(*positions*)
Moves the motor group to the specified user positions

Parameters **positions** (sequence< `Number`[642] >) – the user positions to move to

set_write_position(*w_position*, *timestamp=None*, *propagate=1*)
Sets a new write value for the user position.

Parameters

- **w_position** (sequence< `Number`[643] >) – the new write value for user position
- **propagate** (*int*[644]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

position
motor group positions

get_default_attribute()

get_motion()

motion
motion object

calculate_motion(*new_positions*, *items=None*)

start_move(*new_position*)

poolmoveable

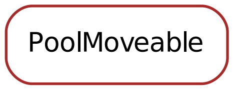
This module is part of the Python Pool libray. It defines the base classes for moveable elements

Classes

- *PoolMoveable*

[640] https://docs.python.org/dev/library/functions.html#bool
[641] https://docs.python.org/dev/library/functions.html#int
[642] https://docs.python.org/dev/library/numbers.html#numbers.Number
[643] https://docs.python.org/dev/library/numbers.html#numbers.Number
[644] https://docs.python.org/dev/library/functions.html#int

PoolMoveable



class PoolMoveable
Bases: `object`[645]

get_size()

calc_move(*positions, ctrl_map, trust=False*)

set_value(*v, propagate=True*)

get_value(*cache=True*)

poolobject

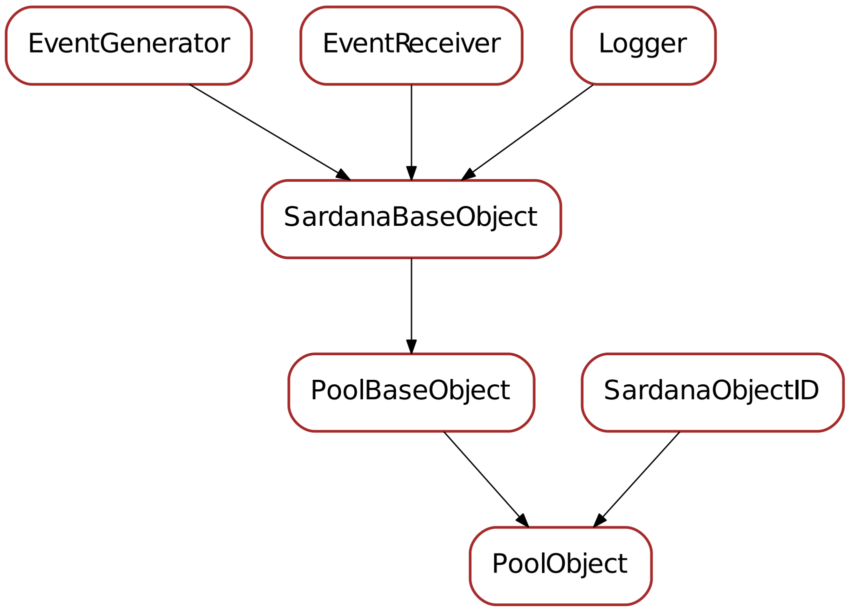
This module is part of the Python Pool library. It defines the base classes for Pool object

Classes

- *PoolObject*

[645] https://docs.python.org/dev/library/functions.html#object

PoolObject



class PoolObject(***kwargs*)

Bases: *sardana.sardanabase.SardanaObjectID*, *sardana.pool.poolbaseobject.PoolBaseObject*

A Pool object that besides the name and reference to the pool has:

- _id : the internal identifier

serialize(**args*, ***kwargs*)

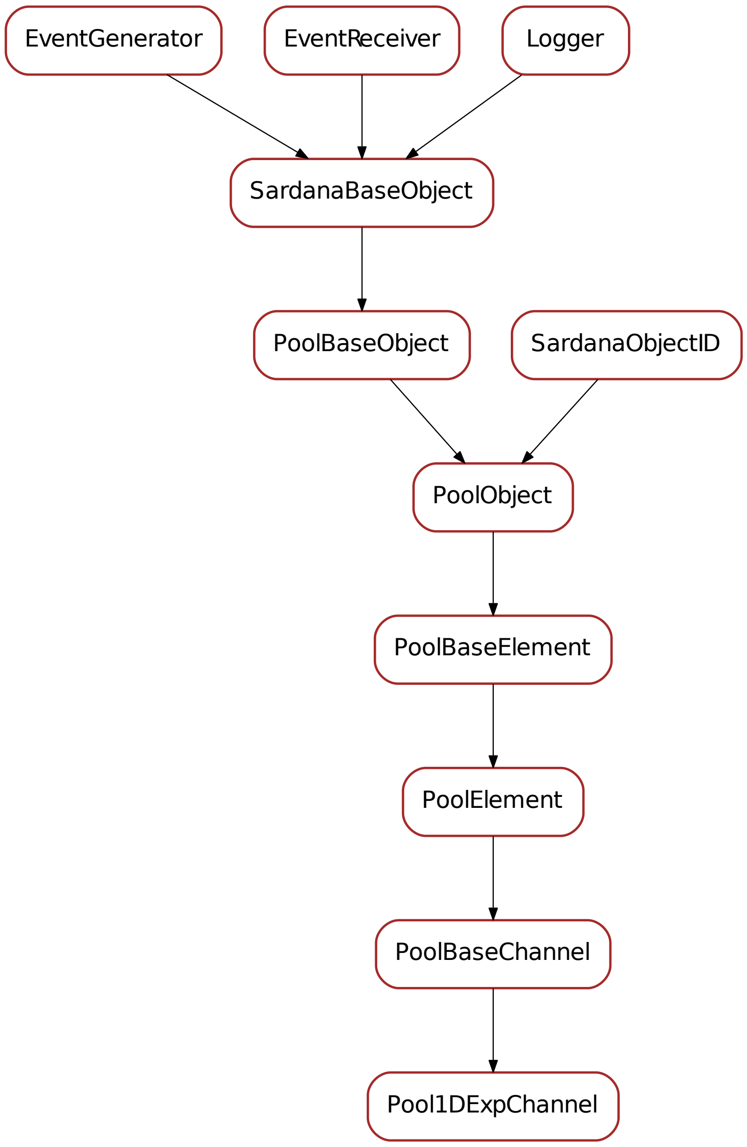
poolonedexpchannel

This module is part of the Python Pool library. It defines the base classes for OneDExpChannel

Classes

- *Pool1DExpChannel*

Pool1DExpChannel



class Pool1DExpChannel(***kwargs*)

Bases: `sardana.pool.poolbasechannel.PoolBaseChannel`

get_data_source(*cache=True, propagate=1*)

read_data_source()

data_source

source identifier for the 1D data

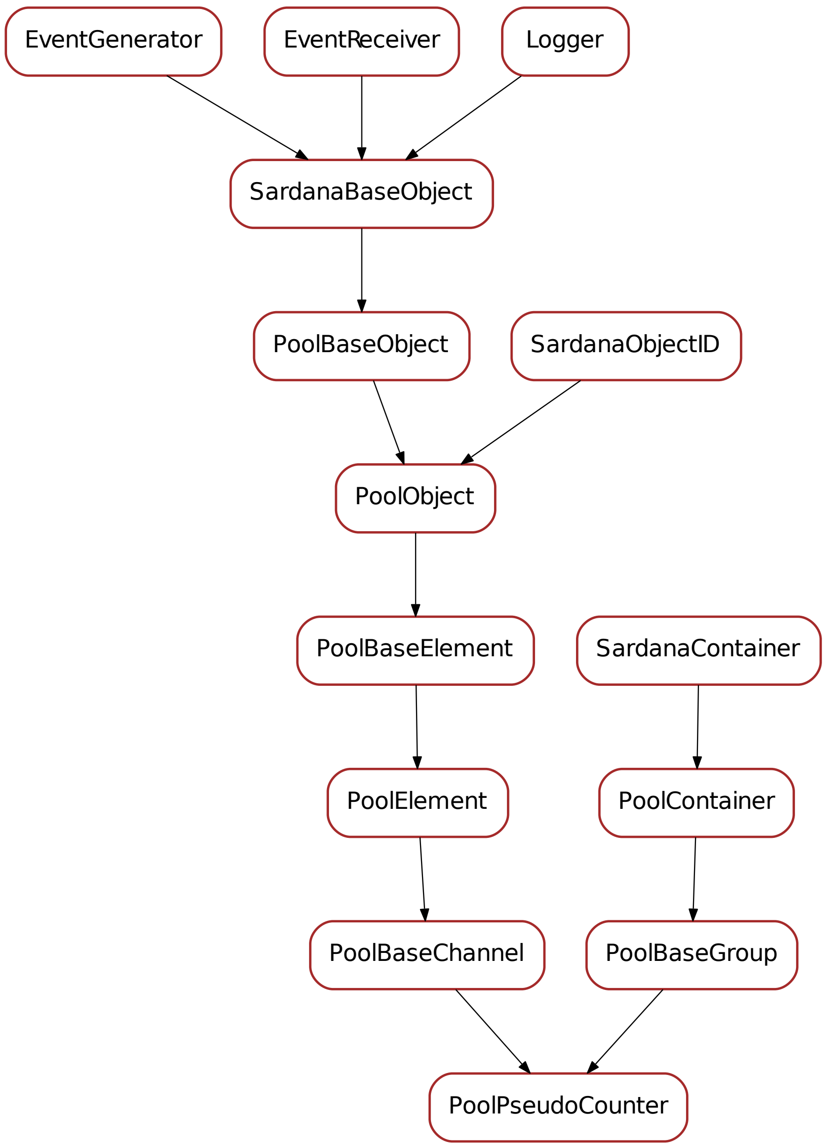
`poolpseudocounter`

This module is part of the Python Pool library. It defines the PoolPseudoCounter class

Classes

- *`PoolPseudoCounter`*

PoolPseudoCounter



class PoolPseudoCounter(***kwargs*)

Bases: `sardana.pool.poolbasegroup.PoolBaseGroup`, *`sardana.pool.poolbasechannel.PoolBaseChannel`*

A class representing a Pseudo Counter in the Sardana Device Pool

ValueAttributeClass

alias of `Value`

ValueBufferClass

alias of `ValueBuffer`

`AcquisitionClass = None`

`serialize`(**args*, ***kwargs*)

`add_user_element`(*element*, *index=None*)

`on_element_changed`(*evt_src*, *evt_type*, *evt_value*)

`get_action_cache`()
Returns the internal action cache object

`set_action_cache`(*action_cache*)

`get_siblings`()

`siblings`
the siblings for this pseudo counter

`calc`(*physical_values=None*)

`calc_all`(*physical_values=None*)

`get_low_level_physical_value_attribute_iterator`()

`get_physical_value_attribute_iterator`()

`get_physical_values_attribute_sequence`()

`get_physical_values_attribute_map`()

`get_physical_value_buffer_iterator`()
Returns an iterator over the value buffer of each user element.

Returns an iterator over the value buffer of each user element.

Return type iter< `SardanaBuffer` >

`get_physical_values`(*cache=True*, *propagate=1*)
Get value for underlying elements.

Parameters

- **`cache`** (*`bool`*[646]) – if `True` (default) return value in cache, otherwise read value from hardware
- **`propagate`** (*`int`*[647]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

Returns the physical value

Return type dict <PoolElement, *`SardanaAttribute`* >

`get_siblings_values`(*use=None*)
Get the last values for all siblings.

Parameters **`use`** (dict <PoolElement, *`SardanaValue`* >) – the already calculated values. If a sibling is in this dictionary, the value stored here is used instead

Returns a dictionary with siblings values

Return type dict <PoolElement, value(float?) >

`get_value`(*cache=True*, *propagate=1*)
Returns the pseudo counter value.

[646] https://docs.python.org/dev/library/functions.html#bool
[647] https://docs.python.org/dev/library/functions.html#int

Parameters

- **`cache`** (*`bool`*[648]) – if `True` (default) return value in cache, otherwise read value from hardware
- **`propagate`** (*`int`*[649]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

Returns the pseudo counter value

Return type *`SardanaAttribute`*

`set_value`(*value*, *propagate=1*)

Starts an acquisition on this channel

Parameters **`value`** (`Number`[650]) – the value to count

`value`

pseudo counter value

`calculate_state_info`(*status_info=None*)

Transforms the given state information. This specific base implementation transforms the given state,status tuple into a state, new_status tuple where new_status is "*self.name* is *state* plus the given status. It is assumed that the given status comes directly from the controller status information.

Parameters **`status_info`** (*`tuple<State, str>`*) – given status information [default: None, meaning use current state status.

Returns a transformed state information

Return type tuple<State, str>

`read_state_info`(*state_info=None*)

`poolpseudomotor`

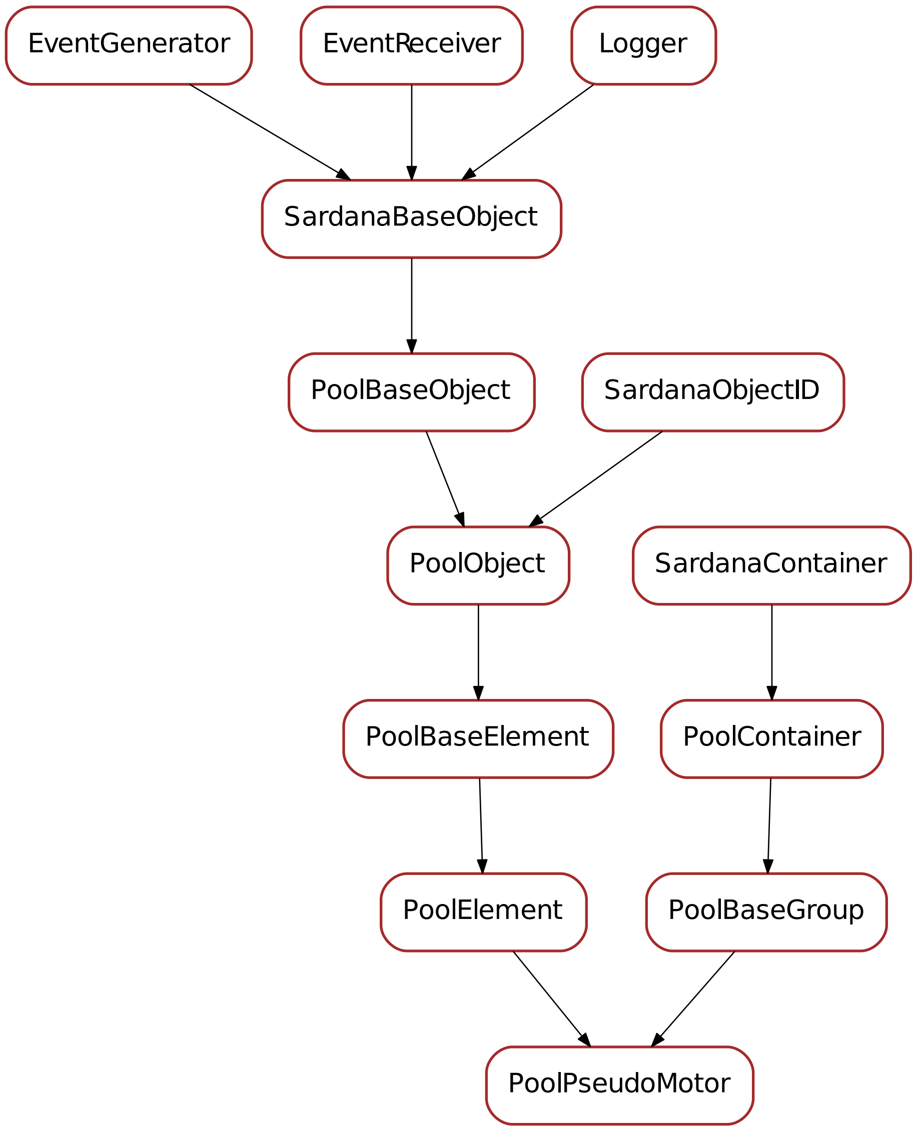
This module is part of the Python Pool library. It defines the PoolPseudoMotor class

Classes

- *`PoolPseudoMotor`*

[648] https://docs.python.org/dev/library/functions.html#bool
[649] https://docs.python.org/dev/library/functions.html#int
[650] https://docs.python.org/dev/library/numbers.html#numbers.Number

PoolPseudoMotor



class PoolPseudoMotor(***kwargs*)

Bases: `sardana.pool.poolbasegroup.PoolBaseGroup`, `sardana.pool.poolelement.PoolElement`

A class representing a Pseudo Motor in the Sardana Device Pool

on_change(*evt_src, evt_type, evt_value*)

serialize(**args, **kwargs*)

set_drift_correction(*drift_correction*)

get_drift_correction()

drift_correction
drift correction

get_action_cache()
Returns the internal action cache object

set_action_cache(*action_cache*)

get_siblings()

siblings
the siblings for this pseudo motor

on_element_changed(*evt_src*, *evt_type*, *evt_value*)

add_user_element(*element*, *index=None*)

calc_pseudo(*physical_positions=None*)

calc_physical(*new_position*)

calc_all_pseudo(*physical_positions=None*)

get_position_attribute()

get_low_level_physical_position_attribute_iterator()

get_physical_position_attribute_iterator()

get_physical_positions_attribute_sequence()

get_physical_positions_attribute_map()

get_physical_positions(*cache=True*, *propagate=1*)
Get positions for underlying elements.

Parameters

- **cache** (`bool`[651]) – if `True` (default) return value in cache, otherwise read value from hardware
- **propagate** (`int`[652]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

Returns the physical positions

Return type dict <PoolElement, `SardanaAttribute`>

get_siblings_positions(*use=None*, *write_pos=True*)
Get the last positions for all siblings. If write_pos is True and a sibling has already been moved before, it's last write position is used. Otherwise its read position is used instead.

Parameters

- **use** (dict <PoolElement, `SardanaValue`>) – the already calculated positions. If a sibling is in this dictionary, the position stored here is used instead
- **write_pos** (`bool`[653]) – determines if should try to use the last set point [default: True]

Returns a dictionary with siblings write positions

[651] https://docs.python.org/dev/library/functions.html#bool
[652] https://docs.python.org/dev/library/functions.html#int
[653] https://docs.python.org/dev/library/functions.html#bool

Return type dict <PoolElement, position(float?) >

get_position(*cache=True*, *propagate=1*)
Returns the user position.

Parameters

- **cache** (*bool*[654]) – if `True` (default) return value in cache, otherwise read value from hardware
- **propagate** (*int*[655]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

Returns the user position

Return type *SardanaAttribute*

set_position(*position*)
Moves the motor to the specified user position

Parameters **position** (`Number`[656]) – the user position to move to

set_write_position(*w_position*, *timestamp=None*, *propagate=1*)
Sets a new write value for the user position.

Parameters

- **w_position** (`Number`[657]) – the new write value for user position
- **propagate** (*int*[658]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

position
pseudo motor position

calculate_state_info(*status_info=None*)
Transforms the given state information. This specific base implementation transforms the given state,status tuple into a state, new_status tuple where new_status is "*self.name* is *state* plus the given status. It is assumed that the given status comes directly from the controller status information.

Parameters **status_info** (*tuple<State, str>*) – given status information [default: None, meaning use current state status.

Returns a transformed state information

Return type tuple<State, str>

read_state_info(*state_info=None*)

get_default_attribute()

get_motion()

motion
motion object

calculate_motion(*new_position*, *items=None*, *calculated=None*)

start_move(*new_position*)

[654] https://docs.python.org/dev/library/functions.html#bool
[655] https://docs.python.org/dev/library/functions.html#int
[656] https://docs.python.org/dev/library/numbers.html#numbers.Number
[657] https://docs.python.org/dev/library/numbers.html#numbers.Number
[658] https://docs.python.org/dev/library/functions.html#int

stop()

abort()

get_operation()

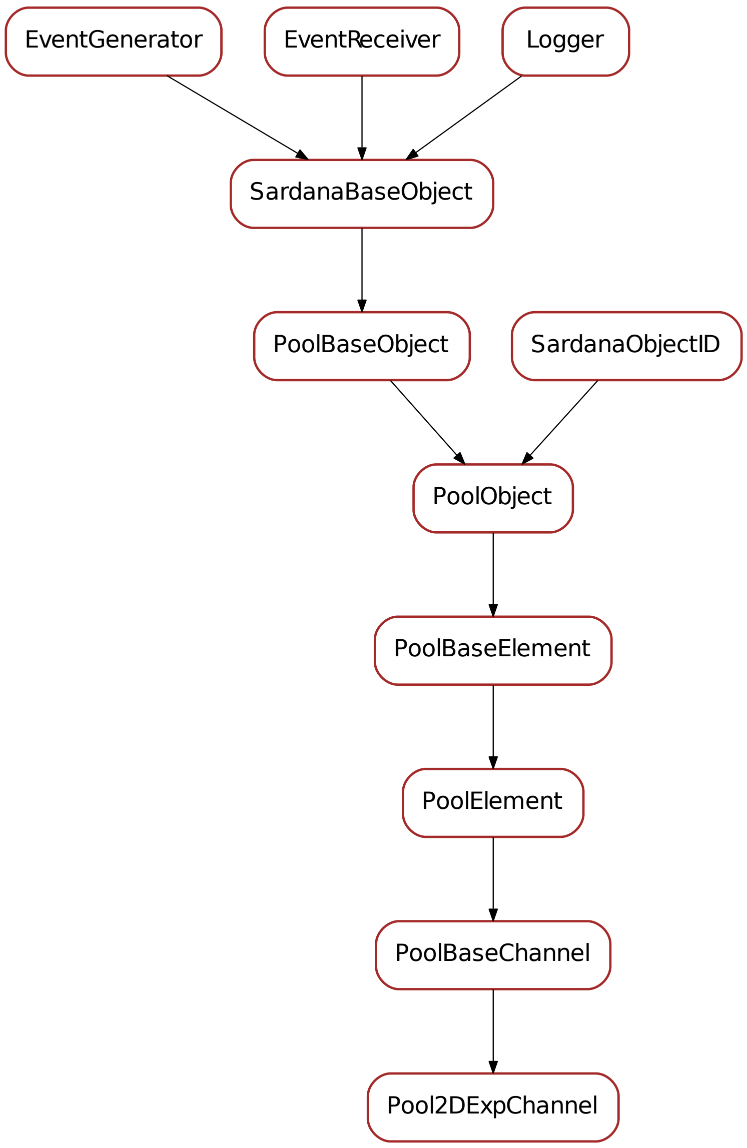
pooltwodexpchannel

This module is part of the Python Pool library. It defines the base classes for TwoDExpChannel

Classes

- *Pool2DExpChannel*

Pool2DExpChannel



class Pool2DExpChannel(***kwargs*)

Bases: *sardana.pool.poolbasechannel.PoolBaseChannel*

get_data_source(*cache=True, propagate=1*)

read_data_source()

data_source

source identifier for the 2D data

`poolutil`

Pool utils

Classes

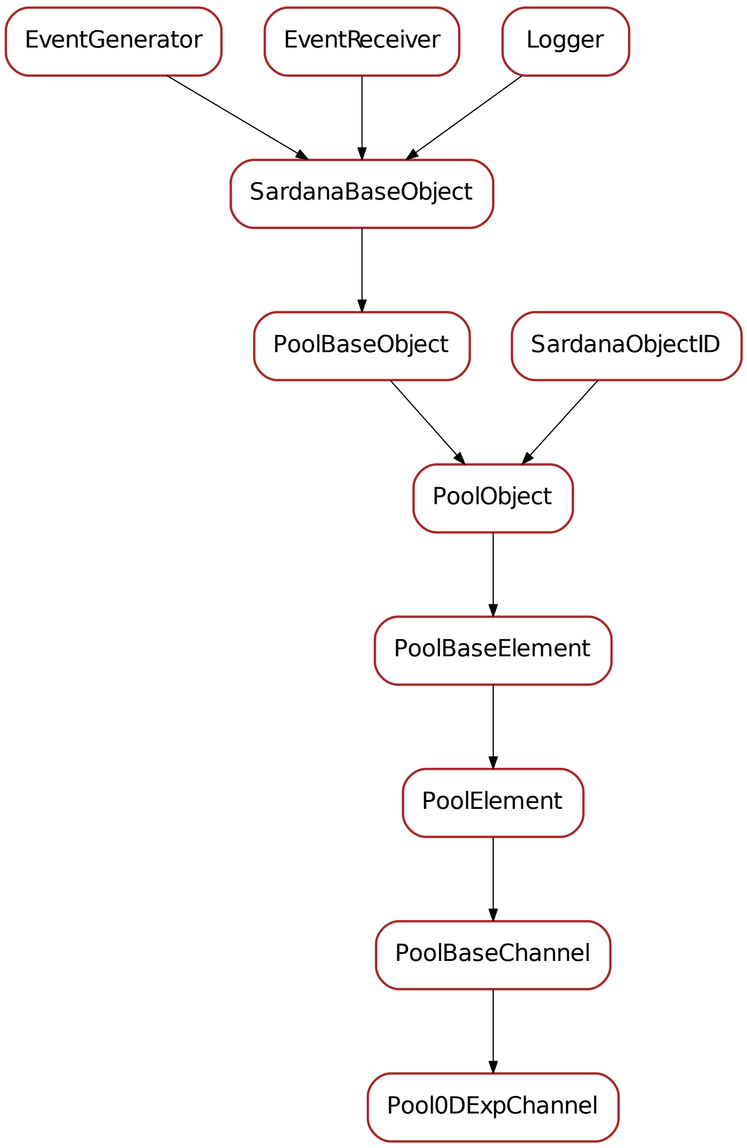
`poolzerodexpchannel`

This module is part of the Python Pool library. It defines the base classes for ZeroDExpChannel

Classes

- *`Pool0DExpChannel`*

Pool0DExpChannel



class Pool0DExpChannel(***kwargs*)

Bases: *sardana.pool.poolbasechannel.PoolBaseChannel*

ValueAttributeClass

alias of `Value`

AcquisitionClass

alias of `sardana.pool.poolacquisition.`

get_accumulation_type()

get_accumulation()

set_accumulation_type(*ctype*)

accumulation

get_accumulated_value_attribute()
Returns the accumulated value attribute object for this 0D.

Returns the accumulated value attribute

Return type *SardanaAttribute*

get_current_value_attribute()
Returns the current value attribute object for this 0D.

Returns the current value attribute

Return type *SardanaAttribute*

get_accumulated_value()
Gets the accumulated value for this 0D.

Returns a *SardanaValue* containing the 0D value

Return type *SardanaAttribute*

Raises Exception if no acquisition has been done yet on this 0D

read_current_value()
Reads the 0D value from hardware.

Returns a *SardanaValue* containing the counter value

Return type *SardanaValue*

put_current_value(*value*, *propagate=1*)
Put a current value.

Parameters

- **value** (*SardanaValue*) – the new value
- **propagate** (*int*[659]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

get_current_value(*cache=True*, *propagate=1*)
Returns the counter value.

Returns the 0D accumulated value

Return type *SardanaAttribute*

current_value
0D value

accumulated_value
0D value

clear_buffer()

get_accumulation_buffer()

accumulation_buffer

get_time_buffer()

[659] https://docs.python.org/dev/library/functions.html#int

time_buffer

start_acquisition(*value=None*)

Classes

- *Controller*
- *MotorController*
- *CounterTimerController*
- *PseudoMotorController*

Constants

- *ControllerAPI*

macroserver

This is the main macro server module

Modules

macros

Modules

communication

This is the communication macro module

class put(**args, **kwargs*)
Sends a string to the communication channel

run(*comch, data*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class get(**args, **kwargs*)
Reads and outputs the data from the communication channel

run(*comch, maxlen*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

demo

This is the demo macro module

clear_sar_demo(*self*)
Undoes changes done with sar_demo

sar_demo(*self*)
Sets up a demo environment. It creates many elements for testing

clear_sar_demo_hkl(*self*)
Undoes changes done with sar_demo

sar_demo_hkl(*self*)
Sets up a demo environment. It creates many elements for testing

env

Environment related macros

class dumpenv(**args*, ***kwargs*)
Dumps the complete environment

run()
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class lsvo(**args*, ***kwargs*)
Lists the view options

run()
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class setvo(**args*, ***kwargs*)
Sets the given view option to the given value

run(*name*, *value*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class usetvo(**args*, ***kwargs*)
Resets the value of the given view option

run(*name*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class lsenv(**args*, ***kwargs*)
Lists the environment in alphabetical order

prepare(*macro_list*, ***opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run(*macro_list*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class senv(**args*, ***kwargs*)
Sets the given environment variable to the given value

run(*env*, *value*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class usenv(**args*, ***kwargs*)
Unsets the given environment variable

run(*env*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class load_env(**args*, ***kwargs*)
Read environment variables from config_env.xml file

run()
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class lsgh(**args*, ***kwargs*)
List general hooks.

Note: The *lsgh* macro has been included in Sardana on a provisional basis. Backwards incompatible changes (up to and including its removal) may occur if deemed necessary by the core developers.

run()
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class defgh(**args*, ***kwargs*)
Define general hook:

```
>>> defgh "mv [[mot02 9]]" pre-scan
>>> defgh "ct 0.1" pre-scan
>>> defgh lsm pre-scan
>>> defgh "mv mot03 10" pre-scan
>>> defgh "Print 'Hello world'" pre-scan
```

Note: The *defgh* macro has been included in Sardana on a provisional basis. Backwards incompatible changes (up to and including its removal) may occur if deemed necessary by the core developers.

run(*macro_name*, *position*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class udefgh(**args*, ***kwargs*)
Undefine general hook. Without arguments undefine all.

Note: The *lsgh* macro has been included in Sardana on a provisional basis. Backwards incompatible changes (up to and including its removal) may occur if deemed necessary by the core developers.

run(*macro_name*, *hook_pos*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

expert

Expert macros

class defm(**args*, ***kwargs*)
Creates a new motor in the active pool

run(*name*, *controller*, *axis*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class defmeas(**args*, ***kwargs*)
Create a new measurement group. First channel in channel_list MUST be an internal sardana channel. At least one channel MUST be a Counter/Timer (by default, the first Counter/Timer in the list will become the master).

prepare(*name*, *channel_list*, ***opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run(*name*, *channel_list*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class udefmeas(**args*, ***kwargs*)
Deletes existing measurement groups

run(*mntgrps*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class defelem(**args*, ***kwargs*)
Creates an element on a controller with an axis

run(*name*, *ctrl*, *axis*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class udefelem(**args*, ***kwargs*)
Deletes existing elements

run (*elements*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class defctrl (**args, **kwargs*)
Creates a new controller 'role_prop' is a sequence of roles and/or properties. - A role is defined as <role name>=<role value> (only applicable to pseudo controllers) - A property is defined as <property name> <property value>

If both roles and properties are supplied, all roles must come before properties. All controller properties that don't have default values must be given.

Example of creating a motor controller (with a host and port properties):

[1]: defctrl SuperMotorController myctrl host homer.springfield.com port 5000

Example of creating a Slit pseudo motor (sl2t and sl2b motor roles, Gap and Offset pseudo motor roles):

[1]: defctrl Slit myslit sl2t=mot01 sl2b=mot02 Gap=gap01 Offset=offset01

run (*ctrl_class, name, props*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class udefctrl (**args, **kwargs*)
Deletes existing controllers

run (*controllers*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class send2ctrl (**args, **kwargs*)
Sends the given data directly to the controller

run (*controller, data*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class edctrlcls (**args, **kwargs*)
Returns the contents of the library file which contains the given controller code.

run (*ctrlclass*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class edctrllib (**args, **kwargs*)
Returns the contents of the given library file

run (*filename*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class commit_ctrllib(**args, **kwargs*)
Puts the contents of the given data in a file inside the pool

run(*filename, username, comment, filedata*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class prdef(**args, **kwargs*)
Returns the the macro code for the given macro name.

run(*macro_data*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class relctrllib(**args, **kwargs*)
Reloads the given controller library code from the pool server filesystem.

run(*ctrl_library*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class addctrllib(**args, **kwargs*)
Adds the given controller library code to the pool server filesystem.

run(*ctrl_library_name*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class relctrlcls(**args, **kwargs*)
Reloads the given controller class code from the pool server filesystem.

run(*ctrl_class*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class rellib(**args, **kwargs*)
Reloads the given python library code from the macro server filesystem.

Warning: use with extreme care! Accidentally reloading a system module or an installed python module may lead to unpredictable behavior

Warning: Prior to the Sardana version 1.6.0 this macro was successfully reloading python libraries located in the MacroPath. The MacroPath is not a correct place to locate your python libraries. They may be successfully loaded on the MacroServer startup, but this can not be guaranteed. In order to use python libraries within your macro code, locate them in either of valid system PYTHONPATH or MacroServer PythonPath property (of the host where MacroServer runs). In order to achieve the previous behavior, just configure the the same directory in both system PYTHONPATH (or MacroServer's PythonPath) and MacroPath.

Note: if python module is used by any macro, don't forget to reload the corresponding macros afterward so the changes take effect.

run(*module_name*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class **relmaclib**(**args*, ***kwargs*)
Reloads the given macro library code from the macro server filesystem.

run(*macro_library*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class **addmaclib**(**args*, ***kwargs*)
Loads a new macro library.

Warning: Keep in mind that macros from the new library can override macros already present in the system.

prepare(*macro_library_name*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run(*macro_library_name*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class **relmac**(**args*, ***kwargs*)
Reloads the given macro code from the macro server filesystem. Attention: All macros inside the same file will also be reloaded.

run(*macro_code*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class **sar_info**(**args*, ***kwargs*)
Prints details about the given sardana object

run(*obj*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

hkl

Macro library containning diffractometer related macros for the macros server Tango device server as part of the Sardana project.

class br(**args*, ***kwargs*)
Move the diffractometer to the reciprocal space coordinates given by H, K and L. If a fourth parameter is given, the combination of angles to be set is the correspondig to the given index. The index of the angles combinations are then changed.

prepare(*H*, *K*, *L*, *AnglesIndex*, *FlagNotBlocking*, *FlagPrinting*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run(*H*, *K*, *L*, *AnglesIndex*, *FlagNotBlocking*, *FlagPrinting*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class ubr(**args*, ***kwargs*)
Move the diffractometer to the reciprocal space coordinates given by H, K and L und update.

prepare(*hh*, *kk*, *ll*, *AnglesIndex*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run(*hh*, *kk*, *ll*, *AnglesIndex*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class ca(**args*, ***kwargs*)
Calculate motor positions for given H K L according to the current operation mode (trajectory 0).

prepare(*H*, *K*, *L*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run(*H*, *K*, *L*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class caa(**args*, ***kwargs*)
Calculate motor positions for given H K L according to the current operation mode (all trajectories)

prepare(*H*, *K*, *L*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run(*H*, *K*, *L*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class ci(**args*, ***kwargs*)
Calculate hkl for given angle values

prepare(*mu*, *theta*, *chi*, *phi*, *gamma*, *delta*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run(*mu*, *theta*, *chi*, *phi*, *gamma*, *delta*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class pa(**args*, ***kwargs*)
Prints information about the active diffractometer.

prepare()
: **Macro API**. Prepare phase. Overwrite as necessary. Default implementation does nothing

run()
: **Macro API**. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

    Raises RuntimeError

class wh(**args*, ***kwargs*)
: Show principal axes and reciprocal space positions.

  Prints the current reciprocal space coordinates (H K L) and the user positions of the principal motors. Depending on the diffractometer geometry, other parameters such as the angles of incidence and reflection (ALPHA and BETA) and the incident wavelength (LAMBDA) may be displayed.

  prepare()
  : **Macro API**. Prepare phase. Overwrite as necessary. Default implementation does nothing

  run()
  : **Macro API**. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

      Raises RuntimeError

class freeze(**args*, ***kwargs*)
: Set psi value for psi constant modes

  prepare(*parameter*, *value*)
  : **Macro API**. Prepare phase. Overwrite as necessary. Default implementation does nothing

  run(*parameter*, *value*)
  : **Macro API**. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

      Raises RuntimeError

class setmode(**args*, ***kwargs*)
: Set operation mode.

  prepare(*new_mode*)
  : **Macro API**. Prepare phase. Overwrite as necessary. Default implementation does nothing

  run(*new_mode*)
  : **Macro API**. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

      Raises RuntimeError

class getmode(**args*, ***kwargs*)
: Get operation mode.

  prepare()
  : **Macro API**. Prepare phase. Overwrite as necessary. Default implementation does nothing

  run()
  : **Macro API**. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

      Raises RuntimeError

class setlat(**args*, ***kwargs*)
: Set the crystal lattice parameters a, b, c, alpha, beta and gamma for the currently active diffraction pseudo motor controller.

prepare(*a*, *b*, *c*, *alpha*, *beta*, *gamma*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run(*a*, *b*, *c*, *alpha*, *beta*, *gamma*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class **or0**(**args*, ***kwargs*)
Set primary orientation reflection.

prepare(*H*, *K*, *L*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run(*H*, *K*, *L*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class **or1**(**args*, ***kwargs*)
Set secondary orientation reflection.

prepare(*H*, *K*, *L*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run(*H*, *K*, *L*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class **setor0**(**args*, ***kwargs*)
Set primary orientation reflection choosing hkl and angle values

prepare(*H*, *K*, *L*, *mu*, *theta*, *chi*, *phi*, *gamma*, *delta*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run(*H*, *K*, *L*, *mu*, *theta*, *chi*, *phi*, *gamma*, *delta*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class **setor1**(**args*, ***kwargs*)
Set secondary orientation reflection choosing hkl and angle values

prepare(*H*, *K*, *L*, *mu*, *theta*, *chi*, *phi*, *gamma*, *delta*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run(*H*, *K*, *L*, *mu*, *theta*, *chi*, *phi*, *gamma*, *delta*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class **setorn**(**args*, ***kwargs*)
Set orientation reflection indicated by the index.

prepare(*ref_id*, *H*, *K*, *L*, *mu*, *theta*, *chi*, *phi*, *gamma*, *delta*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run (*ref_id*, *H*, *K*, *L*, *mu*, *theta*, *chi*, *phi*, *gamma*, *delta*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class setaz (**args*, ***kwargs*)
Set hkl values of the psi reference vector

prepare (*PsiH*, *PsiK*, *PsiL*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run (*PsiH*, *PsiK*, *PsiL*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class computeub (**args*, ***kwargs*)
Compute UB matrix with reflections 0 and 1

prepare ()
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run ()
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class addreflection (**args*, ***kwargs*)
Add reflection at the botton of reflections list

prepare (*H*, *K*, *L*, *affinement*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run (*H*, *K*, *L*, *affinement*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class affine (**args*, ***kwargs*)
Affine current crystal. Fine tunning of lattice parameters and UB matrix based on current crystal reflections. Reflections with affinement set to 0 are not used. A new crystal with the post fix (affine) is created and set as current crystal

prepare ()
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run ()
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class orswap (**args*, ***kwargs*)
Swap values for primary and secondary vectors.

prepare ()
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run()
: **Macro API**. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

    Raises RuntimeError

class newcrystal(**args*, ***kwargs*)
Create a new crystal (if it does not exist) and select it.

prepare(*crystal_name*)
: **Macro API**. Prepare phase. Overwrite as necessary. Default implementation does nothing

run(*crystal_name*)
: **Macro API**. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

    Raises RuntimeError

class hscan(**args*, ***kwargs*)
Scan h axis

prepare(*start_pos*, *final_pos*, *nr_interv*, *integ_time*)
: **Macro API**. Prepare phase. Overwrite as necessary. Default implementation does nothing

class kscan(**args*, ***kwargs*)
Scan k axis

prepare(*start_pos*, *final_pos*, *nr_interv*, *integ_time*)
: **Macro API**. Prepare phase. Overwrite as necessary. Default implementation does nothing

class lscan(**args*, ***kwargs*)
Scan l axis

prepare(*start_pos*, *final_pos*, *nr_interv*, *integ_time*)
: **Macro API**. Prepare phase. Overwrite as necessary. Default implementation does nothing

class hklscan(**args*, ***kwargs*)
Scan h k l axes

prepare(*h_start_pos*, *h_final_pos*, *k_start_pos*, *k_final_pos*, *l_start_pos*, *l_final_pos*, *nr_interv*, *integ_time*)
: **Macro API**. Prepare phase. Overwrite as necessary. Default implementation does nothing

class th2th(**args*, ***kwargs*)
th2th - scan:

Relative scan around current position in del and th with d_th=2*d_delta

run(*rel_start_pos*, *rel_final_pos*, *nr_interv*, *integ_time*)
: **Macro API**. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

    Raises RuntimeError

class savecrystal(**args*, ***kwargs*)
Save crystal information to file

prepare()
: **Macro API**. Prepare phase. Overwrite as necessary. Default implementation does nothing

run()
: **Macro API**. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class loadcrystal(**args*, ***kwargs*)
Load crystal information from file

prepare()
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run()
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class latticecal(**args*, ***kwargs*)
Calibrate lattice parameters a, b or c to current 2theta value

prepare(*parameter*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run(*parameter*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

ioregister

IORegister related macros

class write_ioreg(**args*, ***kwargs*)
Writes a value to an input register

run(*ioreg*, *data*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class read_ioreg(**args*, ***kwargs*)
Reads an output register

run(*ioreg*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

lists

This is the lists macro module

class lsdef(**args*, ***kwargs*)
List all macro definitions

run(*filter*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class lsm(**args, **kwargs*)
Lists all motors

class lspm(**args, **kwargs*)
Lists all existing motors

class lscom(**args, **kwargs*)
Lists all communication channels

class lsior(**args, **kwargs*)
Lists all IORegisters

class lsexp(**args, **kwargs*)
Lists all experiment channels

class lsct(**args, **kwargs*)
Lists all Counter/Timers

class ls0d(**args, **kwargs*)
Lists all 0D experiment channels

class ls1d(**args, **kwargs*)
Lists all 1D experiment channels

class ls2d(**args, **kwargs*)
Lists all 2D experiment channels

class lspc(**args, **kwargs*)
Lists all pseudo counters

class lstg(**args, **kwargs*)
Lists all trigger/gate elements

class lsctrllib(**args, **kwargs*)
Lists all existing controller classes

class lsctrl(**args, **kwargs*)
Lists all existing controllers

class lsi(**args, **kwargs*)
Lists all existing instruments

class lsa(**args, **kwargs*)
Lists all existing objects

class lsmeas(**args, **kwargs*)
List existing measurement groups

prepare(*filter, **opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

class lsmac(**args, **kwargs*)
Lists existing macros

class lsmaclib(**args, **kwargs*)
Lists existing macro libraries.

mca

MCA related macros

class mca_start(**args*, ***kwargs*)
Starts an mca

run(*mca*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class mca_stop(**args*, ***kwargs*)
Stops an mca

run(*mca*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

scan

Macro library containning scan macros for the macros server Tango device server as part of the Sardana project.

class dNscan(**a*, ***kw*)
same as aNscan but it interprets the positions as being relative to the current positions and upon completion, it returns the motors to their original positions

class ascan(**args*, ***kwargs*)
Do an absolute scan of the specified motor. ascan scans one motor, as specified by motor. The motor starts at the position given by start_pos and ends at the position given by final_pos. The step size is (start_pos-final_pos)/nr_interv. The number of data points collected will be nr_interv+1. Count time is given by time which if positive, specifies seconds and if negative, specifies monitor counts.

prepare(*motor*, *start_pos*, *final_pos*, *nr_interv*, *integ_time*, ***opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

class a2scan(**args*, ***kwargs*)
two-motor scan. a2scan scans two motors, as specified by motor1 and motor2. Each motor moves the same number of intervals with starting and ending positions given by start_pos1 and final_pos1, start_pos2 and final_pos2, respectively. The step size for each motor is: (start_pos-final_pos)/nr_interv The number of data points collected will be nr_interv+1. Count time is given by time which if positive, specifies seconds and if negative, specifies monitor counts.

prepare(*motor1*, *start_pos1*, *final_pos1*, *motor2*, *start_pos2*, *final_pos2*, *nr_interv*, *integ_time*, ***opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

class a3scan(**args*, ***kwargs*)
three-motor scan . a3scan scans three motors, as specified by motor1, motor2 and motor3. Each motor moves the same number of intervals with starting and ending positions given by start_pos1 and final_pos1, start_pos2 and final_pos2, start_pos3 and final_pos3, respectively. The step size for each motor is (start_pos-final_pos)/nr_interv. The number of data points collected will be nr_interv+1. Count time is given by time which if positive, specifies seconds and if negative, specifies monitor counts.

prepare(*m1*, *s1*, *f1*, *m2*, *s2*, *f2*, *m3*, *s3*, *f3*, *nr_interv*, *integ_time*, ***opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

class a4scan(**args*, ***kwargs*)
four-motor scan . a4scan scans four motors, as specified by motor1, motor2, motor3 and motor4. Each

motor moves the same number of intervals with starting and ending positions given by start_posN and final_posN (for N=1,2,3,4). The step size for each motor is (start_pos-final_pos)/nr_interv. The number of data points collected will be nr_interv+1. Count time is given by time which if positive, specifies seconds and if negative, specifies monitor counts.

`prepare`(*m1, s1, f1, m2, s2, f2, m3, s3, f3, m4, s4, f4, nr_interv, integ_time, **opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

`class amultiscan`(**args, **kwargs*)
Multiple motor scan. amultiscan scans N motors, as specified by motor1, motor2,...,motorN. Each motor moves the same number of intervals with starting and ending positions given by start_posN and final_posN (for N=1,2,...). The step size for each motor is (start_pos-final_pos)/nr_interv. The number of data points collected will be nr_interv+1. Count time is given by time which if positive, specifies seconds and if negative, specifies monitor counts.

`prepare`(**args, **opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

`class dmultiscan`(**args, **kwargs*)
Multiple motor scan relative to the starting positions. dmultiscan scans N motors, as specified by motor1, motor2,...,motorN. Each motor moves the same number of intervals If each motor is at a position X before the scan begins, it will be scanned from X+start_posN to X+final_posN (where N is one of 1,2,...) The step size for each motor is (start_pos-final_pos)/nr_interv. The number of data points collected will be nr_interv+1. Count time is given by time which if positive, specifies seconds and if negative, specifies monitor counts.

`prepare`(**args, **opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

`class dscan`(**args, **kwargs*)
motor scan relative to the starting position. dscan scans one motor, as specified by motor. If motor motor is at a position X before the scan begins, it will be scanned from X+start_pos to X+final_pos. The step size is (start_pos-final_pos)/nr_interv. The number of data points collected will be nr_interv+1. Count time is given by time which if positive, specifies seconds and if negative, specifies monitor counts.

`prepare`(*motor, start_pos, final_pos, nr_interv, integ_time, **opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

`class d2scan`(**args, **kwargs*)
two-motor scan relative to the starting position. d2scan scans two motors, as specified by motor1 and motor2. Each motor moves the same number of intervals. If each motor is at a position X before the scan begins, it will be scanned from X+start_posN to X+final_posN (where N is one of 1,2). The step size for each motor is (start_pos-final_pos)/nr_interv. The number of data points collected will be nr_interv+1. Count time is given by time which if positive, specifies seconds and if negative, specifies monitor counts.

`prepare`(*motor1, start_pos1, final_pos1, motor2, start_pos2, final_pos2, nr_interv, integ_time, **opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

`class d3scan`(**args, **kwargs*)
three-motor scan . d3scan scans three motors, as specified by motor1, motor2 and motor3. Each motor moves the same number of intervals. If each motor is at a position X before the scan begins, it will be scanned from X+start_posN to X+final_posN (where N is one of 1,2,3) The step size for each motor is (start_pos-final_pos)/nr_interv. The number of data points collected will be nr_interv+1. Count time is given by time which if positive, specifies seconds and if negative, specifies monitor counts.

`prepare`(*m1, s1, f1, m2, s2, f2, m3, s3, f3, nr_interv, integ_time, **opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

class d4scan (**args, **kwargs*)

four-motor scan relative to the starting positions a4scan scans four motors, as specified by motor1, motor2, motor3 and motor4. Each motor moves the same number of intervals. If each motor is at a position X before the scan begins, it will be scanned from X+start_posN to X+final_posN (where N is one of 1,2,3,4). The step size for each motor is (start_pos-final_pos)/nr_interv. The number of data points collected will be nr_interv+1. Count time is given by time which if positive, specifies seconds and if negative, specifies monitor counts. Upon termination, the motors are returned to their starting positions.

prepare (*m1, s1, f1, m2, s2, f2, m3, s3, f3, m4, s4, f4, nr_interv, integ_time, **opts*)

Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

class mesh (**args, **kwargs*)

2d grid scan. The mesh scan traces out a grid using motor1 and motor2. The first motor scans from m1_start_pos to m1_final_pos using the specified number of intervals. The second motor similarly scans from m2_start_pos to m2_final_pos. Each point is counted for for integ_time seconds (or monitor counts, if integ_time is negative). The scan of motor1 is done at each point scanned by motor2. That is, the first motor scan is nested within the second motor scan.

prepare (*m1, m1_start_pos, m1_final_pos, m1_nr_interv, m2, m2_start_pos, m2_final_pos, m2_nr_interv, integ_time, bidirectional, **opts*)

Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run (**args*)

Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class fscan (**args, **kwargs*)

N-dimensional scan along user defined paths. The motion path for each motor is defined through the evaluation of a user-supplied function that is evaluated as a function of the independent variables. -independent variables are supplied through the indepvar string. The syntax for indepvar is "x=expresion1,y=expresion2,..." -If no indep vars need to be defined, write "!" or "*" or "None" -motion path for motor is generated by evaluating the corresponding function 'func' -Count time is given by integ_time. If integ_time is a scalar, then the same integ_time is used for all points. If it evaluates as an array (with same length as the paths), fscan will assign a different integration time to each acquisition point. -If integ_time is positive, it specifies seconds and if negative, specifies monitor counts.

IMPORTANT Notes: -no spaces are allowed in the indepvar string. -all funcs must evaluate to the same number of points

```
>>> fscan x=[1,3,5,7,9],y=arange(5) 0.1 motor1 x**2 motor2 sqrt(y*x+3)
>>> fscan x=[1,3,5,7,9],y=arange(5) [0.1,0.2,0.3,0.4,0.5] motor1 x**2 motor2␣
↪sqrt(y*x+3)
```

prepare (**args, **opts*)

Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run (**args*)

Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class scanhist (**args, **kwargs*)

Shows scan history information. Give optional parameter scan number to display details about a specific scan

run(*scan_number*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class ascanc(**args, **kwargs*)
Do an absolute continuous scan of the specified motor. ascanc scans one motor, as specified by motor.

prepare(*motor, start_pos, final_pos, integ_time, slow_down, **opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

class a2scanc(**args, **kwargs*)
two-motor continuous scan

prepare(*motor1, start_pos1, final_pos1, motor2, start_pos2, final_pos2, integ_time, slow_down, **opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

class a3scanc(**args, **kwargs*)
three-motor continuous scan

prepare(*m1, s1, f1, m2, s2, f2, m3, s3, f3, integ_time, slow_down, **opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

class a4scanc(**args, **kwargs*)
four-motor continuous scan

prepare(*m1, s1, f1, m2, s2, f2, m3, s3, f3, m4, s4, f4, integ_time, slow_down, **opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

class dscanc(**args, **kwargs*)
continuous motor scan relative to the starting position.

prepare(*motor, start_pos, final_pos, integ_time, slow_down, **opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

class d2scanc(**args, **kwargs*)
continuous two-motor scan relative to the starting positions

prepare(*motor1, start_pos1, final_pos1, motor2, start_pos2, final_pos2, integ_time, slow_down, **opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

class d3scanc(**args, **kwargs*)
continuous three-motor scan

prepare(*m1, s1, f1, m2, s2, f2, m3, s3, f3, integ_time, slow_down, **opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

class d4scanc(**args, **kwargs*)
continuous four-motor scan relative to the starting positions

prepare(*m1, s1, f1, m2, s2, f2, m3, s3, f3, m4, s4, f4, integ_time, slow_down, **opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

class meshc(**args, **kwargs*)
2d grid scan. scans continuous

prepare(*m1, m1_start_pos, m1_final_pos, slow_down, m2, m2_start_pos, m2_final_pos, m2_nr_interv, integ_time, bidirectional, **opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run (**args*)

Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class ascanct (**args, **kwargs*)

Do an absolute continuous scan of the specified motor. ascanct scans one motor, as specified by motor. The motor starts before the position given by start_pos in order to reach the constant velocity at the start_pos and finishes at the position after the final_pos in order to maintain the constant velocity until the final_pos.

prepare (*motor, start_pos, final_pos, nr_interv, integ_time, latency_time, **opts*)

Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

class a2scanct (**args, **kwargs*)

Two-motor continuous scan. a2scanct scans two motors, as specified by motor1 and motor2. Each motor starts before the position given by its start_pos in order to reach the constant velocity at its start_pos and finishes at the position after its final_pos in order to maintain the constant velocity until its final_pos.

prepare (*m1, s1, f1, m2, s2, f2, nr_interv, integ_time, latency_time, **opts*)

Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

class a3scanct (**args, **kwargs*)

Three-motor continuous scan. a2scanct scans three motors, as specified by motor1, motor2 and motor3. Each motor starts before the position given by its start_pos in order to reach the constant velocity at its start_pos and finishes at the position after its final_pos in order to maintain the constant velocity until its final_pos.

prepare (*m1, s1, f1, m2, s2, f2, m3, s3, f3, nr_interv, integ_time, latency_time, **opts*)

Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

class a4scanct (**args, **kwargs*)

Four-motor continuous scan. a2scanct scans four motors, as specified by motor1, motor2, motor3 and motor4. Each motor starts before the position given by its start_pos in order to reach the constant velocity at its start_pos and finishes at the position after its final_pos in order to maintain the constant velocity until its final_pos.

prepare (*m1, s1, f1, m2, s2, f2, m3, s3, f3, m4, s4, f4, nr_interv, integ_time, latency_time, **opts*)

Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

class meshct (**args, **kwargs*)

2d grid scan . The mesh scan traces out a grid using motor1 and motor2. The first motor scans in contiuous mode from m1_start_pos to m1_final_pos using the specified number of intervals. The second motor similarly scans from m2_start_pos to m2_final_pos but it does not move during the continuous scan. Each point is counted for integ_time seconds (or monitor counts, if integ_time is negative). The scan of motor1 is done at each point scanned by motor2. That is, the first motor scan is nested within the second motor scan.

prepare (*m1, m1_start_pos, m1_final_pos, m1_nr_interv, m2, m2_start_pos, m2_final_pos, m2_nr_interv, integ_time, bidirectional, latency_time, **opts*)

Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run (**args*)

Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class timescan(**args*, ***kwargs*)

Do a time scan over the specified time intervals. The scan starts immediately. The number of data points collected will be nr_interv + 1. Count time is given by integ_time. Latency time will be the longer one of latency_time and measurement group latency time.

prepare(*nr_interv*, *integ_time*, *latency_time*)

Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run(**args*)

Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

sequence

This is the sequence macro module

class sequence(**args*, ***kwargs*)

This macro executes a sequence of macros. As a parameter it receives a string which is a xml structure. These macros which allow hooks can nest another sequence (xml structure). In such a case, this macro is executed recursively.

run(**pars*)

Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

standard

This is the standard macro module

class wa(**args*, ***kwargs*)

Show all motor positions

prepare(*filter*, ***opts*)

Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run(*filter*)

Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class pwa(**args*, ***kwargs*)

Show all motor positions in a pretty table

run(*filter*)

Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class set_lim(**args*, ***kwargs*)

Sets the software limits on the specified motor hello

run(*motor*, *low*, *high*)

Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class set_lm(**args, **kwargs*)
Sets the dial limits on the specified motor

run(*motor, low, high*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class set_pos(**args, **kwargs*)
Sets the position of the motor to the specified value

run(*motor, pos*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class wm(**args, **kwargs*)
Show the position of the specified motors.

prepare(*motor_list, **opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run(*motor_list*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class pwm(**args, **kwargs*)
Show the position of the specified motors in a pretty table

run(*motor_list*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class mv(**args, **kwargs*)
Move motor(s) to the specified position(s)

run(*motor_pos_list*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class mstate(**args, **kwargs*)
Prints the state of a motor

run(*motor*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class umv(**args, **kwargs*)
Move motor(s) to the specified position(s) and update

prepare(*motor_pos_list, **opts*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run(*motor_pos_list*)
: **Macro API**. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

  Raises RuntimeError

class mvr(**args*, ***kwargs*)
: Move motor(s) relative to the current position(s)

  run(*motor_disp_list*)
  : **Macro API**. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

    Raises RuntimeError

class umvr(**args*, ***kwargs*)
: Move motor(s) relative to the current position(s) and update

  run(*motor_disp_list*)
  : **Macro API**. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

    Raises RuntimeError

class tw(**args*, ***kwargs*)
: Tweak motor by variable delta

  run(*motor*, *delta*)
  : **Macro API**. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

    Raises RuntimeError

class ct(**args*, ***kwargs*)
: Count for the specified time on the active measurement group

  prepare(*integ_time*, ***opts*)
  : **Macro API**. Prepare phase. Overwrite as necessary. Default implementation does nothing

  run(*integ_time*)
  : **Macro API**. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

    Raises RuntimeError

class uct(**args*, ***kwargs*)
: Count on the active measurement group and update

  prepare(*integ_time*, ***opts*)
  : **Macro API**. Prepare phase. Overwrite as necessary. Default implementation does nothing

  run(*integ_time*)
  : **Macro API**. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

    Raises RuntimeError

class settimer(**args*, ***kwargs*)
: Defines the timer channel for the active measurement group

  run(*timer*)
  : **Macro API**. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class logmacro(**args*, ***kwargs*)
Turn on/off logging of the spock output.

Note: The logmacro class has been included in Sardana on a provisional basis. Backwards incompatible changes (up to and including its removal) may occur if deemed necessary by the core developers

run(*offon*, *mode*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

class repeat(**args*, ***kwargs*)
This macro executes as many repetitions of it's body hook macros as specified by nr parameter. If nr parameter has negative value, repetitions will be executed until you stop repeat macro.

prepare(*nr*)
Macro API. Prepare phase. Overwrite as necessary. Default implementation does nothing

run(*nr*)
Macro API. Runs the macro. **Overwrite MANDATORY!** Default implementation raises RuntimeError.

Raises RuntimeError

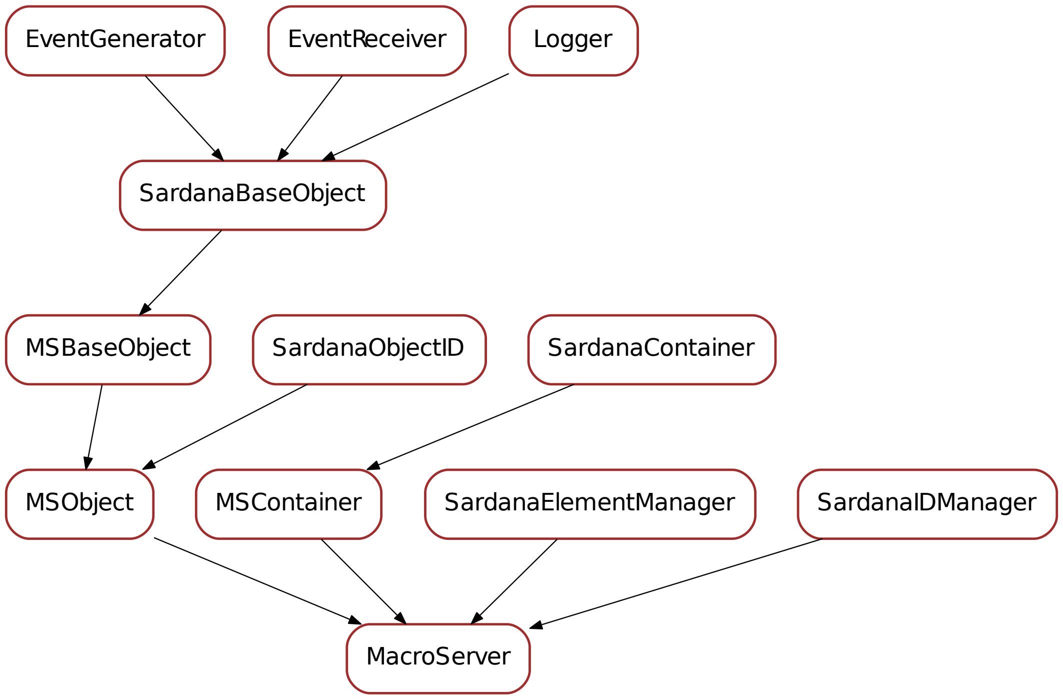
`macroserver`

Functions

Classes

- *MacroServer*

MacroServer



class MacroServer (*full_name*, *name=None*, *macro_path=None*, *environment_db=None*, *recorder_path=None*)

Bases: *sardana.macroserver.mscontainer.MSContainer*, *sardana.macroserver.msbase.MSObject*, *sardana.sardanamanager.SardanaElementManager*, `sardana.sardanamanager.SardanaIDManager`

msbase

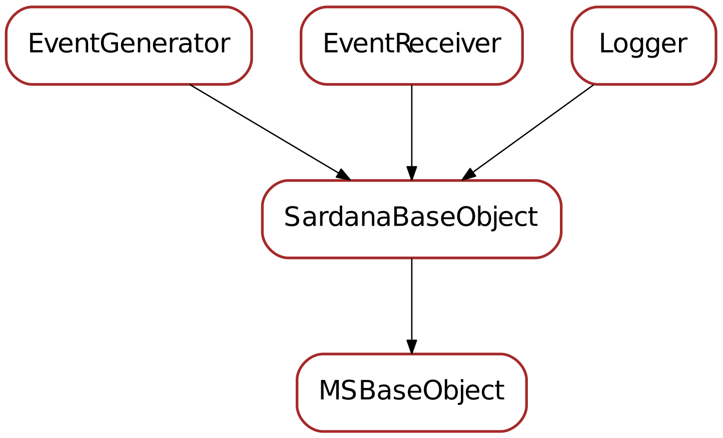
This module is part of the Python MacroServer libray. It defines the base classes for MacroServer object

Functions

Classes

- *MSBaseObject*
- *MSObject*

MSBaseObject

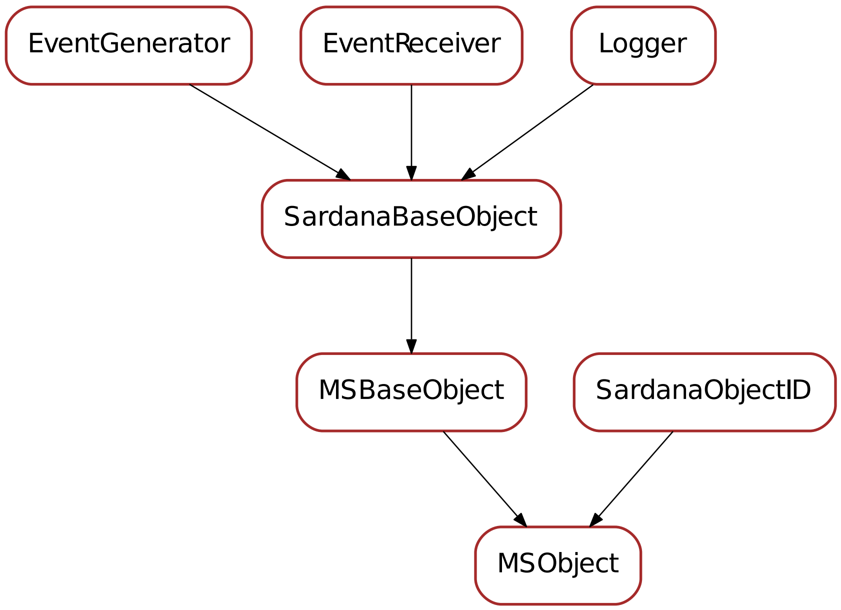


class MSBaseObject(***kwargs*)

Bases: `sardana.sardanabase.SardanaBaseObject`

The MacroServer most abstract object.

MSObject



class MSObject(***kwargs*)

Bases: *sardana.sardanabase.SardanaObjectID*, *sardana.macroserver.msbase.MSBaseObject*

A macro server object that besides the name and reference to the macro server base object has:

- _id : the internal identifier

mscontainer

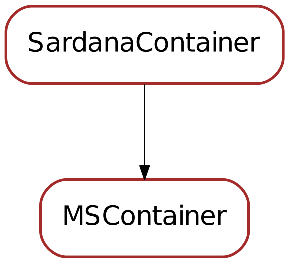
This module is part of the Python Macro Server libray. It defines the base classes for a macro server container element

Functions

Classes

- *MSContainer*

MacroServer



class MSContainer
    Bases: *sardana.sardanacontainer.SardanaContainer*

msdoor

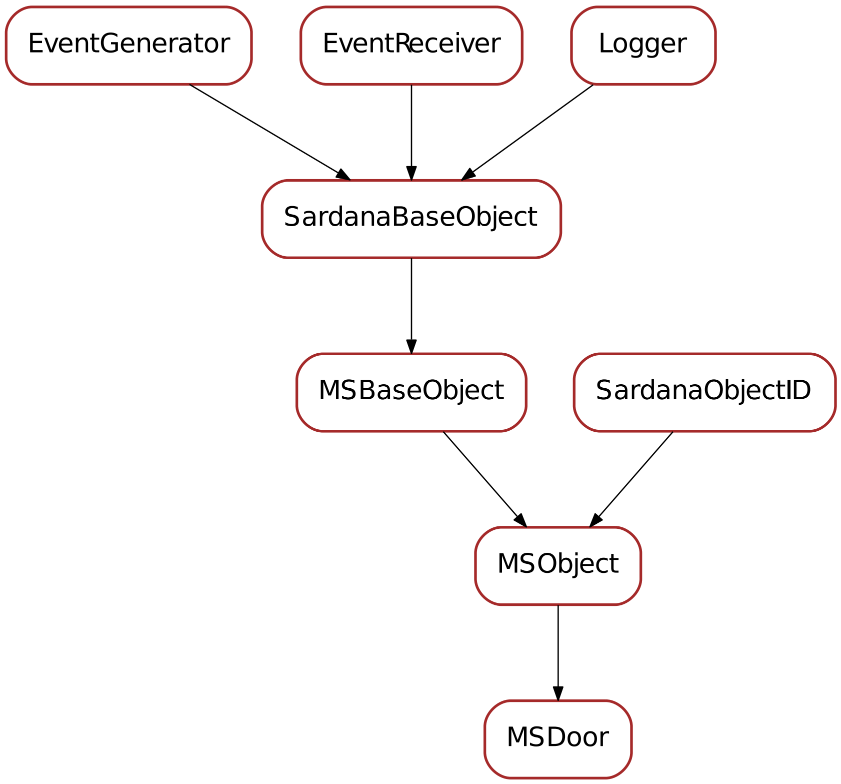
This module contains the class definition for the macro server door

Functions

Classes

- *MSDoor*

MSDoor



class MSDoor(***kwargs*)

Bases: *sardana.macroserver.msbase.MSObject*

Sardana door object

msenvmanager

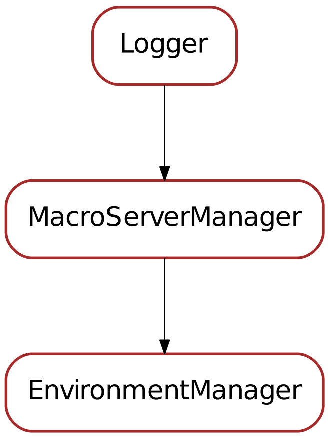
This module contains the class definition for the MacroServer environment manager

Functions

Classes

- *EnvironmentManager*

EnvironmentManager



class EnvironmentManager(*macro_server*, *environment_db=None*)
Bases: `sardana.macroserver.msmanager.MacroServerManager`

The MacroServer environment manager class. It is designed to be a singleton for the entire application.

`msexception`

This module contains the class definition for the MacroServer environment manager

Functions

Classes

- `MacroServerException`

MacroServerException

class MacroServerException(**args*, ***kwargs*)
Bases: `sardana.sardanaexception.SardanaException`

`msmacromanager`

This module contains the class definition for the MacroServer macro manager

Functions

Classes

- *MacroManager*
- *MacroExecutor*

MacroManager

class MacroManager(*macro_server, macro_path=None*)
    Bases: *sardana.macroserver.msmanager.MacroServerManager*

MacroExecutor

class MacroExecutor(*door*)
    Bases: `taurus.core.util.log.Logger`

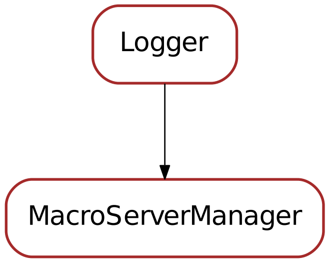
msmanager

Functions

Classes

- *MacroServerManager*

MacroServerManager



class MacroServerManager(*macro_server*)
    Bases: `taurus.core.util.log.Logger`

    Base Class for macro server managers

msmetamacro

This module contains the class definition for the MacroServer meta macro information

Functions

Classes

- *MacroLibrary*
- *MacroClass*
- *MacroFunction*

MacroLibrary

class MacroLibrary(***kwargs*)

Bases: *sardana.sardanameta.SardanaLibrary*

Object representing a python module containing macro classes and/or macro functions. Public members:

- module - reference to python module
- file_path - complete (absolute) path (with file name at the end)
- file_name - file name (including file extension)
- path - complete (absolute) path
- name - (=module name) module name (without file extension)
- macros - dict<str, MacroClass>
- **exc_info - exception information if an error occurred when loading** the module

serialize(**args, **kwargs*)

Returns a serializable object describing this object.

Returns a serializable dict

Return type dict[660]

get_macro(*meta_name*)

Returns a :class:~'sardana.sardanameta.SardanaCode' for the given meta name or None if the meta does not exist in this library.

Parameters meta_name (*str*[661]) – the meta name (class, function)

Returns a meta or None

Return type :class:~'sardana.sardanameta.SardanaCode'

get_macros()

Returns a sequence of the meta (class and functions) that belong to this library.

Returns a sequence of meta (class and functions) that belong to this library

Return type seq<:class:~'sardana.sardanameta.SardanaCode'>

[660] https://docs.python.org/dev/library/stdtypes.html#dict
[661] https://docs.python.org/dev/library/stdtypes.html#str

has_macro(*meta_name*)
Returns True if the given meta name belongs to this library or False otherwise.

Parameters **meta_name** (*str*[662]) – the meta name

Returns True if the given meta (class or function) name belongs to this library or False otherwise

Return type bool[663]

has_macros()
Returns True if any meta object exists in the library or False otherwise.

Returns True if any meta object (class or function) exists in the library or False otherwise

Return type bool[664]

add_macro_class(*meta_class*)
Adds a new :class:~'sardana.sardanameta.SardanaClass' to this library.

Parameters **meta_class** (*:class:~`sardana.sardanameta.SardanaClass`*) – the meta class to be added to this library

get_macro_class(*meta_class_name*)
Returns a :class:~'sardana.sardanameta.SardanaClass' for the given meta class name or None if the meta class does not exist in this library.

Parameters **meta_class_name** (*str*[665]) – the meta class name

Returns a meta class or None

Return type :class:~'sardana.sardanameta.SardanaClass'

get_macro_classes()
Returns a sequence of the meta classes that belong to this library.

Returns a sequence of meta classes that belong to this library

Return type seq<:class:~'sardana.sardanameta.SardanaClass'>

has_macro_class(*meta_class_name*)
Returns True if the given meta class name belongs to this library or False otherwise.

Parameters **meta_class_name** (*str*[666]) – the meta class name

Returns True if the given meta class name belongs to this library or False otherwise

Return type bool[667]

add_macro_function(*meta_function*)
Adds a new :class:~'sardana.sardanameta.SardanaFunction' to this library.

Parameters **meta_function** (*:class:~`sardana.sardanameta.SardanaFunction`*) – the meta function to be added to this library

get_macro_function(*meta_function_name*)
Returns a :class:~'sardana.sardanameta.SardanaFunction' for the given meta function name or None if the meta function does not exist in this library.

[662] https://docs.python.org/dev/library/stdtypes.html#str
[663] https://docs.python.org/dev/library/functions.html#bool
[664] https://docs.python.org/dev/library/functions.html#bool
[665] https://docs.python.org/dev/library/stdtypes.html#str
[666] https://docs.python.org/dev/library/stdtypes.html#str
[667] https://docs.python.org/dev/library/functions.html#bool

Parameters **meta_function_name** (*str*[668]) – the meta function name

Returns a meta function or None

Return type :class:~'sardana.sardanameta.SardanaFunction'

get_macro_functions()

Returns a sequence of the meta functions that belong to this library.

Returns a sequence of meta functions that belong to this library

Return type seq<:class:~'sardana.sardanameta.SardanaFunction'>

has_macro_function(*meta_function_name*)

Returns True if the given meta function name belongs to this library or False otherwise.

Parameters **meta_function_name** (*str*[669]) – the meta function name

Returns True if the given meta function name belongs to this library or False otherwise

Return type bool[670]

Parameterizable

class Parameterizable

Bases: object[671]

Helper class to handle parameter and result definition for a *MacroClass* or a *MacroFunction*

MacroClass

class MacroClass(***kwargs*)

Bases: *sardana.sardanameta.SardanaClass*, *sardana.macroserver.msmetamacro.Parameterizable*

serialize(**args*, ***kwargs*)

Returns a serializable object describing this object.

Returns a serializable dict

Return type dict[672]

MacroFunction

class MacroFunction(***kwargs*)

Bases: sardana.sardanameta.SardanaFunction, *sardana.macroserver.msmetamacro.Parameterizable*

serialize(**args*, ***kwargs*)

Returns a serializable object describing this object.

Returns a serializable dict

Return type dict[673]

668 https://docs.python.org/dev/library/stdtypes.html#str
669 https://docs.python.org/dev/library/stdtypes.html#str
670 https://docs.python.org/dev/library/functions.html#bool
671 https://docs.python.org/dev/library/functions.html#object
672 https://docs.python.org/dev/library/stdtypes.html#dict
673 https://docs.python.org/dev/library/stdtypes.html#dict

`msparameter`

This module contains the definition of the macroserver parameters for macros

Functions

Classes

- *`ParamType`*

ParamType

`class ParamType`(*macro_server*, *name*)
Bases: *`sardana.macroserver.msbase.MSBaseObject`*

`mstypemanager`

This module contains the definition of the macroserver data type manager

Functions

Classes

- *`TypeManager`*

TypeManager

`class TypeManager`(*macro_server*)
Bases: *`sardana.macroserver.msmanager.MacroServerManager`*

`tango`

Modules

`core`

Modules

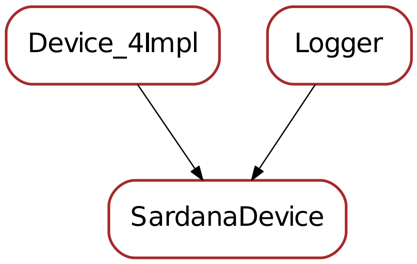
`SardanaDevice`

Generic Sardana Tango device module

Classes

- *SardanaDevice*
- *SardanaDeviceClass*

SardanaDevice



class SardanaDevice(*dclass*, *name*)

Bases: `PyTango.Device_4Impl`, `taurus.core.util.log.Logger`

SardanaDevice represents the base class for all Sardana `PyTango.DeviceImpl` classes

init(*name*)

initialize the device once in the object lifetime. Override when necessary but **always** call the method from your super class

Parameters name (*str*[674]) – device name

get_alias()

Returns this device alias name

Returns this device alias

Return type str[675]

alias

the device alias name

get_full_name()

Compose full name from the TANGO_HOST information and device name.

In case Sardana is used with Taurus 3 the full name is of format "dbhost:dbport/<domain>/<family>/<member>" where dbhost may be either FQDN or PQDN, depending on the TANGO_HOST configuration.

In case Sardana is used with Taurus 4 the full name is of format "tango://dbhost:dbport/<domain>/<family>/<member>" where dbhost is always FQDN.

Returns this device full name

[674] https://docs.python.org/dev/library/stdtypes.html#str
[675] https://docs.python.org/dev/library/stdtypes.html#str

Return type str[676]

init_device()
Initialize the device. Called during startup after `init()` and every time the tango `Init` command is executed. Override when necessary but **always** call the method from your super class

init_device_nodb()
Internal method. Initialize the device when tango database is not being used (example: in demos)

delete_device()
Clean the device. Called during shutdown and every time the tango `Init` command is executed. Override when necessary but **always** call the method from your super class

set_change_events(*evts_checked*, *evts_not_checked*)
Helper method to set change events on attributes

Parameters

- **evts_checked** (seq<`str`[677]>) – list of attribute names to activate change events programatically with tango filter active
- **evts_not_checked** (seq<`str`[678]>) – list of attribute names to activate change events programatically with tango filter inactive. Use this with care! Attributes configured with no change event filter may potentially generated a lot of events!

initialize_dynamic_attributes()
Initialize dynamic attributes. Default implementation does nothing. Override when necessary.

get_event_thread_pool()
Return the `ThreadPool`[679] used by sardana to send tango events.

Returns the sardana `ThreadPool`[680]

Return type `ThreadPool`[681]

get_attribute_by_name(*attr_name*)
Gets the attribute for the given name.

Parameters **attr_name** (`str`[682]) – attribute name

Returns the attribute object

Return type `Attribute`

get_wattribute_by_name(*attr_name*)
Gets the writable attribute for the given name.

Parameters **attr_name** (`str`[683]) – attribute name

Returns the attribute object

Return type `WAttribute`

get_database()
Helper method to return a reference to the current tango database

Returns the Tango database

[676] https://docs.python.org/dev/library/stdtypes.html#str
[677] https://docs.python.org/dev/library/stdtypes.html#str
[678] https://docs.python.org/dev/library/stdtypes.html#str
[679] http://taurus-scada.org/devel/api/taurus/core/util/_ThreadPool.html#taurus.core.util.ThreadPool
[680] http://taurus-scada.org/devel/api/taurus/core/util/_ThreadPool.html#taurus.core.util.ThreadPool
[681] http://taurus-scada.org/devel/api/taurus/core/util/_ThreadPool.html#taurus.core.util.ThreadPool
[682] https://docs.python.org/dev/library/stdtypes.html#str
[683] https://docs.python.org/dev/library/stdtypes.html#str

Return type `Database`

`set_write_attribute`(*attr*, *w_value*)

`set_attribute`(*attr*, *value=None*, *w_value=None*, *timestamp=None*, *quality=None*, *error=None*, *priority=1*, *synch=True*)

Sets the given attribute value. If timestamp is not given, *now* is used as timestamp. If quality is not given VALID is assigned. If error is given an error event is sent (with no value and quality INVALID). If priority is > 1, the event filter is temporarily disabled so the event is sent for sure. If synch is set to True, wait for fire event to finish

Parameters

- **`attr`** (`PyTango.Attribute`) – the tango attribute
- **`value`** (`object`[684]) – the value to be set (not mandatory if setting an error) [default: None]
- **`w_value`** – the write value to be set (not mandatory) [default: None, meaning maintain current write value]
- **`timestamp`** (float or `PyTango.TimeVal`) – the timestamp associated with the operation [default: None, meaning use *now* as timestamp]
- **`quality`** (`PyTango.AttrQuality`) – attribute quality [default: None, meaning VALID]
- **`error`** (`PyTango.DevFailed`) – a tango DevFailed error or None if not an error [default: None]
- **`priority`** (`int`[685]) – event priority [default: 1, meaning *normal* priority]. If priority is > 1, the event filter is temporarily disabled so the event is sent for sure. The event filter is restored to the previous value
- **`synch`** – If synch is set to True, wait for fire event to finish. If False, a job is sent to the sardana thread pool and the method returns immediately [default: True]

`set_attribute_push`(*attr*, *value=None*, *w_value=None*, *timestamp=None*, *quality=None*, *error=None*, *priority=1*, *synch=True*)

Synchronous internal implementation of `set_attribute()` (synch is passed to this method because it might need to know if it is being executed in a synchronous or asynchronous context).

`calculate_tango_state`(*ctrl_state*, *update=False*)

Calculate tango state based on the controller state.

Parameters

- **`ctrl_state`** (`State`) – the state returned by the controller
- **`update`** (`bool`[686]) – if True, set the state of this device with the calculated tango state [default: False:

Returns the corresponding tango state

Return type `PyTango.DevState`

`calculate_tango_status`(*ctrl_status*, *update=False*)

Calculate tango status based on the controller status.

Parameters

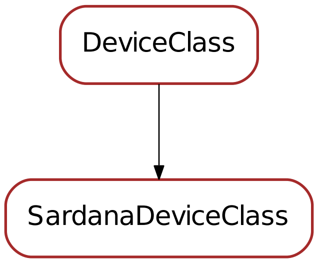
[684] https://docs.python.org/dev/library/functions.html#object
[685] https://docs.python.org/dev/library/functions.html#int
[686] https://docs.python.org/dev/library/functions.html#bool

- **ctrl_status** (*str*[687]) – the status returned by the controller
- **update** (*bool*[688]) – if True, set the state of this device with the calculated tango state [default: False:

Returns the corresponding tango state

Return type str[689]

SardanaDeviceClass



class SardanaDeviceClass(*name*)

Bases: `PyTango.DeviceClass`

SardanaDeviceClass represents the base class for all Sardana `PyTango.DeviceClass` classes

class_property_list = {}

Sardana device class properties definition

See also:

server

device_property_list = {}

Sardana device properties definition

See also:

server

cmd_list = {}

Sardana device command definition

See also:

server

attr_list = {}

Sardana device attribute definition

See also:

[687] https://docs.python.org/dev/library/stdtypes.html#str
[688] https://docs.python.org/dev/library/functions.html#bool
[689] https://docs.python.org/dev/library/stdtypes.html#str

server

write_class_property()
Write class properties `ProjectTitle`, `Description`, `doc_url`, `InheritedFrom` and `__icon`

dyn_attr(*dev_list*)
Invoked to create dynamic attributes for the given devices. Default implementation calls *`SardanaDevice.initialize_dynamic_attributes()`* for each device

Parameters dev_list (`PyTango.DeviceImpl`) – list of devices

device_name_factory(*dev_name_list*)
Builds list of device names to use when no Database is being used

Parameters dev_name_list (seq<obj:*list*>) – list to be filled with device names

pool

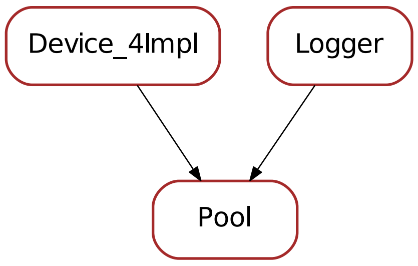
Modules

Pool

Classes

- *Pool*
- *PoolClass*

Pool



class Pool(*cl, name*)
Bases: `PyTango.Device_4Impl`, `taurus.core.util.log.Logger`

ElementsCache = None

init(*full_name*)

get_full_name()
Compose full name from the TANGO_HOST information and device name.

In case Sardana is used with Taurus 3 the full name is of format "dbhost:dbport/<domain>/<family>/<member>" where dbhost may be either FQDN or PQDN, depending on the TANGO_HOST configuration.

In case Sardana is used with Taurus 4 the full name is of format "tango://dbhost:dbport/<domain>/<family>/<member>" where dbhost is always FQDN.

Returns this device full name

Return type str[690]

`pool`

`delete_device`

`init_device`

`always_executed_hook`()

`read_attr_hardware`(*data*)

`read_ControllerLibList`(*attr*)

`read_ControllerClassList`(*attr*)

`read_ControllerList`(*attr*)

`read_InstrumentList`(*attr*)

`read_ExpChannelList`(*attr*)

`read_AcqChannelList`(*attr*)

`read_MotorGroupList`(*attr*)

`read_MotorList`(*attr*)

`read_TriggerGateList`(*attr*)

`read_MeasurementGroupList`(*attr*)

`read_IORegisterList`(*attr*)

`read_ComChannelList`(*attr*)

`getElements`(*cache=True*)

`read_Elements`(*attr*)

`is_Elements_allowed`(*req_type*)

`is_ControllerLibList_allowed`(*req_type*)

`is_ControllerClassList_allowed`(*req_type*)

`is_ControllerList_allowed`(*req_type*)

`is_InstrumentList_allowed`(*req_type*)

`is_ExpChannelList_allowed`(*req_type*)

`is_TriggerGateList_allowed`(*req_type*)

`is_AcqChannelList_allowed`(*req_type*)

`is_MotorGroupList_allowed`(*req_type*)

`is_MotorList_allowed`(*req_type*)

[690] https://docs.python.org/dev/library/stdtypes.html#str

is_MeasurementGroupList_allowed(*req_type*)

is_IORegisterList_allowed(*req_type*)

is_ComChannelList_allowed(*req_type*)

CreateController(*argin*)
Tango command to create controller.

Parameters **argin** (`list<str>`) – Must give either:

- **A JSON encoded dict as first string with:**
  - mandatory keys: 'type', 'library', 'klass' and 'name' (values are strings).
  - **optional keys:**
    - 'properties': a dict with keys being property names and values the property values
    - 'roles': a dict with keys being controller roles and values being element names. (example: { 'gap' : 'motor21', 'offset' : 'motor55' }). Only applicable of pseudo controllers
- a sequence of strings: <type>, <library>, <class>, <name> [, <role_name>'='<element name>] [, <property name>, <property value>]

Examples:

```
data = dict(type='Motor', library='DummyMotorController',
            klass='DummyMotorController',
            name='my_motor_ctrl_1')
pool.CreateController([json.dumps(data)])

pool.CreateController(['Motor', 'DummyMotorController',
↪'DummyMotorController',
                       'my_motor_ctrl_2'])
```

Returns None

CreateInstrument(*argin*)
Tango command to create instrument.

Parameters **argin** (`list<str>`) – Must give either:

- **A JSON encoded dict as first string with:**
  - mandatory keys: 'full_name', 'klass' (values are strings).
- a sequence of strings: <full_name>, <class>

Examples:

```
pool.CreateInstrument(['/OH', 'NXhutch'])
pool.CreateInstrument(['/OH/Mono', 'NXmonochromator'])
pool.CreateInstrument(['/EH', 'NXhutch'])
pool.CreateInstrument(['/EH/Pilatus', 'NXdetector'])
```

Returns None

CreateElement(*argin*)
Tango command to create element (motor, counter/timer, 0D, 1D, 2D, IORegister).

Parameters **argin** (`list<str>`) – Must give either:

- **A JSON encoded dict as first string with:**

  - mandatory keys: 'type', 'ctrl_name', 'axis', 'name' (values are strings).
  - **optional keys:**
    * 'full_name' : a string representing the full tango device name
- a sequence of strings: <type>, <ctrl_name>, <axis>, <name> [, <full_name>]

Examples:

```
data = dict(type='Motor', ctrl_name='my_motor_ctrl_1', axis='4
↪', name='theta',
            full_name='BL99/EH/THETA')
pool.CreateElement([json.dumps(data)])

pool.CreateElement(['Motor', 'my_motor_ctrl_1', '1', 'phi',
↪'BL99/EH/PHI'])
```

Returns None

`RenameElement`(*argin*)
Tango command to rename the element (rename Pool element and put new alias in the Tango Database).

Parameters `argin` –
Two elements sequence of strings: <old element name>, <new element name>
Returns None

`CreateMotorGroup`(*argin*)
Tango command to create motor group.
Parameters `argin` (`list<str>`) – Must give either:

- **A JSON encoded dict as first string with:**
  - mandatory keys: 'name', 'elements' (with value being a list of moveables)
  - **optional keys:**
    * 'full_name': with value being a full tango device name
- a sequence of strings: <motor group name> [, <element>]"

Examples:

```
data = dict(name='diffrac_motor_group', elements=['theta',
↪'theta2', 'phi'])
pool.CreateMotorGroup([json.dumps(data)])

pool.CreateMotorGroup(['diffrac_mg', 'theta', 'theta2' ])
```

Returns None

`CreateMeasurementGroup`(*argin*)
Tango command to create measurement group.
Parameters `argin` (`list<str>`) – Must give either:

- **A JSON encoded dict as first string with:**
  - mandatory keys: 'name', 'elements' (with value being a list of acquirables)"

- **optional keys:**
  - 'full_name': with value being a full tango device name
- a sequence of strings: <motor group name> [, <element>]"

An acquirable is either a sardana element (counter/timer, 0D, 1D, 2D, motor) or a tango attribute (ex: sys/tg_test/1/short_spectrum_ro)

Examples:

```
data = dict(name='my_exp_01', elements=['timer', 'C1', 'sys/tg_
↪test/1/double_scalar'])
pool.CreateMeasurementGroup([json.dumps(data)])

pool.CreateMeasurementGroup(['my_exp_02', 'timer', 'CCD1',
↪'sys/tg_test/1/short_spectrum_ro'])
```

Returns None

`on_pool_changed`(*evt_src*, *evt_type*, *evt_value*)

`DeleteElement`(*name*)
Tango command to delete element.
Parameters **`argin`** (`str`[691]) – name of element to be deleted

Returns None

`GetControllerClassInfo`(*names*)
Tango command to get detailed information about a controller class.
Parameters **`argin`** (`str`[692]) – Must give either:

- A JSON encoded list of controller class names
- a controller class name

Examples:

```
data = "DummyMotorController", "DummyCounterTimerController"
pool.GetControllerClassInfo(json.dumps(data))
pool.GetControllerClassInfo("DummyMotorController")
```

Returns
a JSON encoded string describing the controller class
Return type str[693]

`ReloadControllerLib`(*lib_name*)
Tango command to reload the controller library code.
Parameters **`argin`** (`str`[694]) – the controller library name (without extension)

Returns None

`ReloadControllerClass`(*class_name*)
Tango command to reload the controller class code (reloads the entire library where the class is described).
Parameters **`argin`** (`str`[695]) – the controller class name

Returns None

[691] https://docs.python.org/dev/library/stdtypes.html#str
[692] https://docs.python.org/dev/library/stdtypes.html#str
[693] https://docs.python.org/dev/library/stdtypes.html#str
[694] https://docs.python.org/dev/library/stdtypes.html#str
[695] https://docs.python.org/dev/library/stdtypes.html#str

Stop()
Stops all elements managed by this Pool

Parameters **argin** – None

Returns None

Abort()
Aborts all elements managed by this Pool

Parameters **argin** – None

Returns None

SendToController(*stream*)

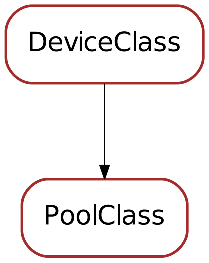
GetFile(*name*)

PutFile(*file_data*)

GetControllerCode(*argin*)

SetControllerCode(*argin*)

PoolClass



class PoolClass(*name*)
Bases: `PyTango.DeviceClass`

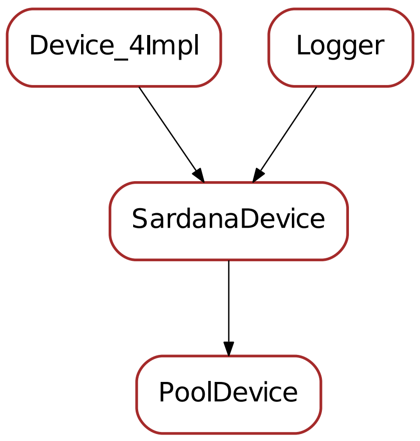
PoolDevice

Generic Tango Pool Device base classes

Classes

- *PoolDevice*
- *PoolDeviceClass*
- *PoolElementDevice*
- *PoolElementDeviceClass*
- *PoolGroupDevice*
- *PoolGroupDeviceClass*

PoolDevice



class PoolDevice(*dclass*, *name*)

Bases: `sardana.tango.core.SardanaDevice.SardanaDevice`

Base Tango Pool device class

ExtremeErrorStates = (<_mock._Mock object>, <_mock._Mock object>)
list of extreme error states

BusyStates = (<_mock._Mock object>, <_mock._Mock object>)
list of busy states

BusyRetries = 3
Maximum number of retries in a busy state

init(*name*)
initialize the device once in the object lifetime. Override when necessary but **always** call the method from your super class

Parameters **name** (`str`[696]) – device name

pool_device
The tango pool device

pool
The sardana pool object

get_element()
Returns the underlying pool element object

Returns the underlying pool element object

Return type `PoolElement`

set_element(*element*)
Associates this device with the sardana element

Parameters **element** (`PoolElement`) – the sardana element

[696] https://docs.python.org/dev/library/stdtypes.html#str

element
The underlying sardana element

init_device()
Initialize the device. Called during startup after *init()* and every time the tango `Init` command is executed. Override when necessary but **always** call the method from your super class

delete_device()
Clean the device. Called during shutdown and every time the tango `Init` command is executed. Override when necessary but **always** call the method from your super class

Abort()
The tango abort command. Aborts the active operation

is_Abort_allowed()
Returns True if it is allowed to execute the tango abort command

Returns True if it is allowed to execute the tango abort command or False otherwise

Return type bool[697]

Stop()
The tango stop command. Stops the active operation

is_Stop_allowed()
Returns True if it is allowed to execute the tango stop command

Returns True if it is allowed to execute the tango stop command or False otherwise

Return type bool[698]

get_dynamic_attributes()
Returns the standard dynamic and fully dynamic attributes for this device. The return is a tuple of two dictionaries:

- standard attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information
- dynamic attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information

tango information seq< `CmdArgType`, `AttrDataFormat`, `AttrWriteType` >
attribute information attribute information as returned by the sardana controller

Returns the standard dynamic and fully dynamic attributes

Return type seq< `CaselessDict`[699], `CaselessDict`[700]>

initialize_dynamic_attributes()
Initializes this device dynamic attributes

remove_unwanted_dynamic_attributes(*new_std_attrs*, *new_dyn_attrs*)
Removes unwanted dynamic attributes from previous device creation

add_dynamic_attribute(*attr_name*, *data_info*, *attr_info*, *read*, *write*, *is_allowed*)
Adds a single dynamic attribute

Parameters

- **attr_name** (`str`[701]) – the attribute name
- **data_info** (seq< `CmdArgType`, `AttrDataFormat`, `AttrWriteType` >) – tango attribute information

697 https://docs.python.org/dev/library/functions.html#bool
698 https://docs.python.org/dev/library/functions.html#bool
699 http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict
700 http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict
701 https://docs.python.org/dev/library/stdtypes.html#str

- **attr_info** – attribute information
- **read** – read method for the attribute
- **write** – write method for the attribute
- **is_allowed** – is allowed method

add_standard_attribute(*attr_name*, *data_info*, *attr_info*, *read*, *write*, *is_allowed*)
Adds a single standard dynamic attribute

Parameters

- **attr_name** (`str`[702]) – the attribute name
- **data_info** (seq< `CmdArgType`, `AttrDataFormat`, `AttrWriteType` >) – tango attribute information
- **attr_info** – attribute information
- **read** – read method for the attribute
- **write** – write method for the attribute
- **is_allowed** – is allowed method

read_DynamicAttribute(*attr*)
Generic read dynamic attribute. Default implementation raises `NotImplementedError`[703]

Parameters **attr** (`Attribute`) – attribute to be read

Raises `NotImplementedError`[704]

write_DynamicAttribute(*attr*)
Generic write dynamic attribute. Default implementation raises `NotImplementedError`[705]

Parameters **attr** (`Attribute`) – attribute to be written

Raises `NotImplementedError`[706]

is_DynamicAttribute_allowed(*req_type*)
Generic is dynamic attribute allowed. Default implementation calls `_is_allowed()`

Parameters **req_type** – request type

dev_state()
Calculates and returns the device state. Called by Tango on a read state request.

Returns the device state

Return type `DevState`

dev_status()
Calculates and returns the device status. Called by Tango on a read status request.

Returns the device status

Return type str[707]

wait_for_operation()
Waits for an operation to finish. It uses the maxumum number of retries. Sleeps 0.01s between retries.

Raises `Exception`[708] in case of a timeout

[702] https://docs.python.org/dev/library/stdtypes.html#str
[703] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[704] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[705] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[706] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[707] https://docs.python.org/dev/library/stdtypes.html#str
[708] https://docs.python.org/dev/library/exceptions.html#Exception

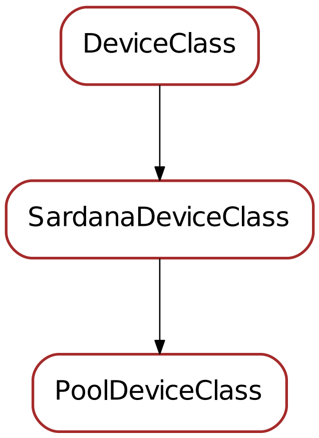
Restore()
Restore tango command. Restores the attributes to their former glory. This applies to memorized writable attributes which have a set point stored in the database

get_restore_data()

get_attributes_to_restore()

restore_attribute(*attribute, write_meth, db_value*)

PoolDeviceClass



class PoolDeviceClass(*name*)
Bases: `sardana.tango.core.SardanaDevice.SardanaDeviceClass`

Base Tango Pool Device Class class

class_property_list = {}
Sardana device class properties definition

See also:

server

device_property_list = {'Force_HW_Read': [<_mock._Mock object at 0x7f07dbbd1890>, 'Fo
Sardana device properties definition

See also:

server

cmd_list = {'Abort': [[<_mock._Mock object at 0x7f07dbbd1f10>, ''], [<_mock._Mock obj
Sardana device command definition

See also:

server

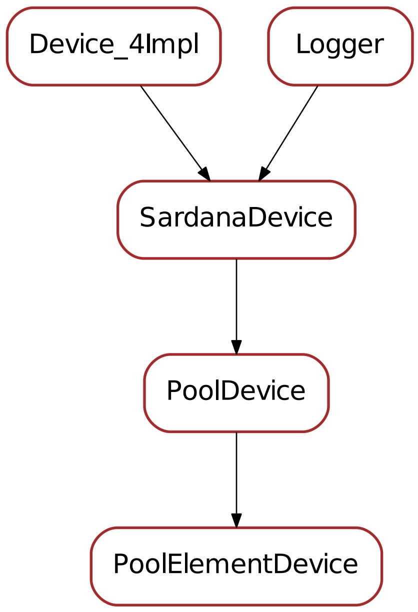
attr_list = {}
Sardana device attribute definition

See also:

server

standard_attr_list = {}

PoolElementDevice



class PoolElementDevice(*dclass, name*)
Bases: *sardana.tango.pool.PoolDevice.PoolDevice*

Base Tango Pool Element Device class

init_device()
Initialize the device. Called during startup after `init()` and every time the tango `Init` command is executed. Override when necessary but **always** call the method from your super class

read_Instrument(*attr*)
Read the value of the `Instrument` tango attribute. Returns the instrument full name or empty string if this element doesn't belong to any instrument
Parameters attr (`Attribute`) – tango instrument attribute

write_Instrument(*attr*)
Write the value of the `Instrument` tango attribute. Sets a new instrument full name or empty string if this element doesn't belong to any instrument. The instrument **must** have been previously created.
Parameters attr (`Attribute`) – tango instrument attribute

`get_dynamic_attributes`()

Override of *`PoolDevice.get_dynamic_attributes`*. Returns the standard dynamic and fully dynamic attributes for this device. The return is a tuple of two dictionaries:

- standard attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information
- dynamic attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information

tango information seq< `CmdArgType`, `AttrDataFormat`, `AttrWriteType` >
attribute information attribute information as returned by the sardana controller

Returns the standard dynamic and fully dynamic attributes

Return type seq< `CaselessDict`[709], `CaselessDict`[710]>

`read_DynamicAttribute`(*attr*)

Read a generic dynamic attribute. Calls the controller of this element to get the dynamic attribute value

Parameters **`attr`** (`Attribute`) – tango attribute

`write_DynamicAttribute`(*attr*)

Write a generic dynamic attribute. Calls the controller of this element to get the dynamic attribute value

Parameters **`attr`** (`Attribute`) – tango attribute

`read_SimulationMode`(*attr*)

Read the current simulation mode.

Parameters **`attr`** (`Attribute`) – tango attribute

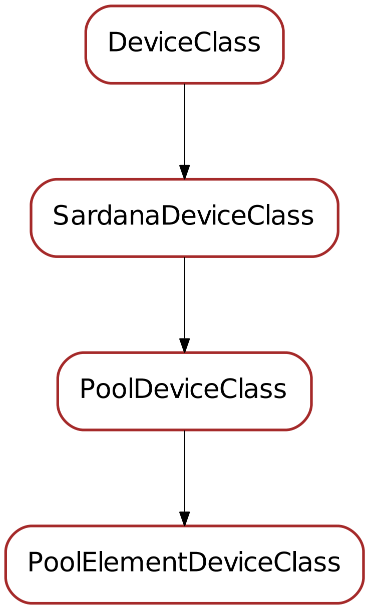
`write_SimulationMode`(*attr*)

Sets the simulation mode.

Parameters **`attr`** (`Attribute`) – tango attribute

[709] http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict
[710] http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict

PoolElementDeviceClass



class PoolElementDeviceClass(*name*)

Bases: *sardana.tango.pool.PoolDevice.PoolDeviceClass*

Base Tango Pool Element Device Class class

device_property_list = {'Axis': [<_mock._Mock object at 0x7f07dbbd1a10>, 'Axis in the

Sardana device properties definition

See also:

server

attr_list = {'Instrument': [[<_mock._Mock object at 0x7f07dbbd1a90>, <_mock._Mock obj

Sardana device attribute definition

See also:

server

cmd_list = {'Abort': [[<_mock._Mock object at 0x7f07dbbd1f10>, ''], [<_mock._Mock obj

get_standard_attr_info(*attr*)

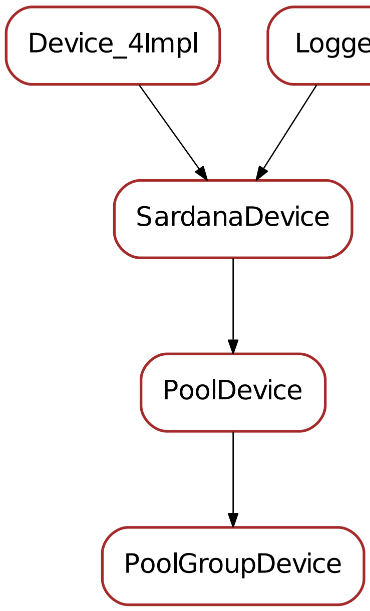
Returns information about the standard attribute

Parameters attr (*str*[711]) – attribute name

Returns a sequence of tango data_type, data format

[711] https://docs.python.org/dev/library/stdtypes.html#str

PoolGroupDevice



class PoolGroupDevice(*dclass, name*)

Bases: *sardana.tango.pool.PoolDevice.PoolDevice*

Base Tango Pool Group Device class

read_ElementList(*attr*)

Read the element list.

Parameters **attr** (`Attribute`) – tango attribute

get_element_names()

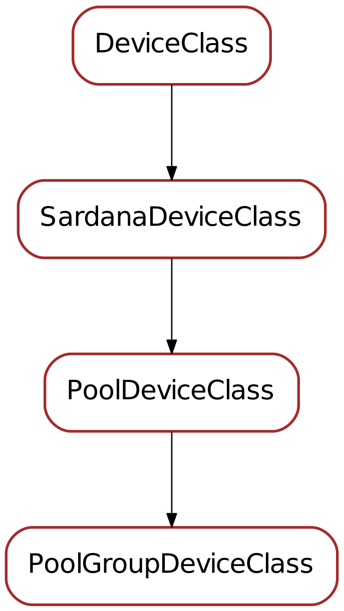
Returns the list of element names.

Returns a list of attribute names

elements_changed(*evt_src, evt_type, evt_value*)

Callback for when the elements of this group changed

PoolGroupDeviceClass



class PoolGroupDeviceClass(*name*)

Bases: *sardana.tango.pool.PoolDevice.PoolDeviceClass*

Base Tango Pool Group Device Class class

device_property_list = {'Elements': [<_mock._Mock object at 0x7f07dbbd1e10>, 'element

Sardana device properties definition

See also:

server

cmd_list = {'Abort': [[<_mock._Mock object at 0x7f07dbbd1f10>, ''], [<_mock._Mock obj

Sardana device command definition

See also:

server

attr_list = {'ElementList': [[<_mock._Mock object at 0x7f07dbbd1a90>, <_mock._Mock ob

Sardana device attribute definition

See also:

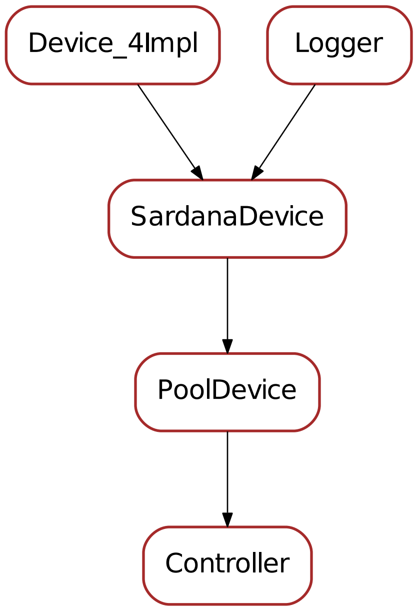
server

`Controller`

Classes

- *`Controller`*
- *`ControllerClass`*

Controller



class Controller(*dclass, name*)

Bases: *`sardana.tango.pool.PoolDevice.PoolDevice`*

init(*name*)

initialize the device once in the object lifetime. Override when necessary but **always** call the method from your super class

Parameters name (*str*[712]) – device name

get_ctrl()

set_ctrl(*ctrl*)

ctrl

delete_device

init_device

get_role_ids()

[712] https://docs.python.org/dev/library/stdtypes.html#str

always_executed_hook()

read_attr_hardware(*data*)

dev_state()
Calculates and returns the device state. Called by Tango on a read state request.

Returns the device state

Return type `DevState`

dev_status()
Calculates and returns the device status. Called by Tango on a read status request.

Returns the device status

Return type str[713]

read_ElementList(*attr*)

CreateElement(*argin*)

DeleteElement(*argin*)

get_element_names()

on_controller_changed(*event_src*, *event_type*, *event_value*)

get_dynamic_attributes()
Returns the standard dynamic and fully dynamic attributes for this device. The return is a tuple of two dictionaries:

- standard attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information
- dynamic attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information

tango information seq< `CmdArgType`, `AttrDataFormat`, `AttrWriteType` >
attribute information attribute information as returned by the sardana controller

Returns the standard dynamic and fully dynamic attributes

Return type seq< `CaselessDict`[714], `CaselessDict`[715]>

read_DynamicAttribute(*attr*)
Generic read dynamic attribute. Default implementation raises `NotImplementedError`[716]

Parameters **attr** (`Attribute`) – attribute to be read

Raises `NotImplementedError`[717]

write_DynamicAttribute(*attr*)
Generic write dynamic attribute. Default implementation raises `NotImplementedError`[718]

Parameters **attr** (`Attribute`) – attribute to be written

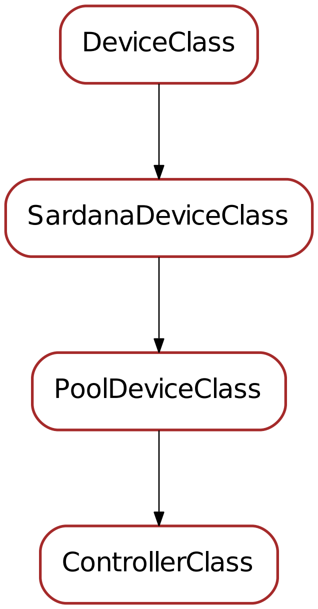
Raises `NotImplementedError`[719]

read_LogLevel(*attr*)

write_LogLevel(*attr*)

[713] https://docs.python.org/dev/library/stdtypes.html#str
[714] http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict
[715] http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict
[716] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[717] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[718] https://docs.python.org/dev/library/exceptions.html#NotImplementedError
[719] https://docs.python.org/dev/library/exceptions.html#NotImplementedError

ControllerClass



class ControllerClass(*name*)

Bases: *sardana.tango.pool.PoolDevice.PoolDeviceClass*

class_property_list = {}

device_property_list = {'Force_HW_Read': [<_mock._Mock object at 0x7f07dbbd1890>, 'Fo

cmd_list = {'Abort': [[<_mock._Mock object at 0x7f07dbbd1f10>, ''], [<_mock._Mock obj

attr_list = {'ElementList': [[<_mock._Mock object at 0x7f07dbbd1a90>, <_mock._Mock ob

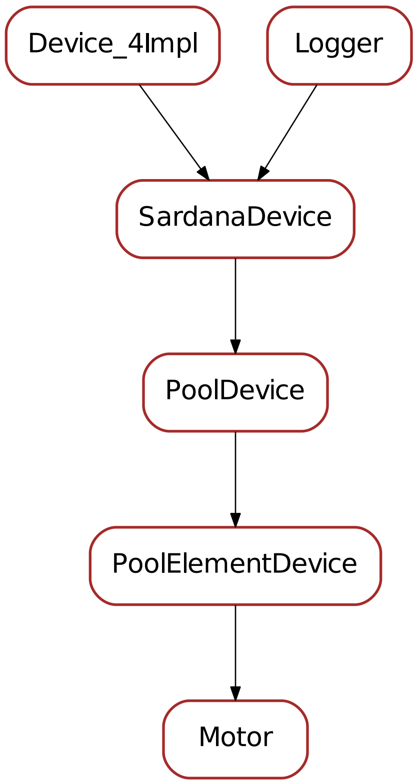
Motor

The sardana tango motor module

Classes

- *Motor*
- *MotorClass*

Motor



class Motor(*dclass, name*)

Bases: *sardana.tango.pool.PoolDevice.PoolElementDevice*

The tango motor device class. This class exposes through a tango device the sardana motor (*PoolMotor*).

The states

The motor interface knows five states which are ON, MOVING, ALARM, FAULT and UNKNOWN. A motor device is in MOVING state when it is moving! It is in ALARM state when it has reached one of the limit switches and is in FAULT if its controller software is not available (impossible to load it) or if a fault is reported from the hardware controller. The motor is in the UNKNOWN state if an exception occurs during the communication between the pool and the hardware controller. When the motor is in ALARM state, its status will indicate which limit switches is active.

The commands

The motor interface supports 3 commands on top of the Tango classical Init, State and Status commands. These commands are summarized in the following table:

Command name	Input data type	Output data type
Stop	void	void
Abort	void	void
DefinePosition	Tango::DevDouble	void
SaveConfig	void	void
MoveRelative	Tango::DevDouble	void

- **Stop** : It stops a running motion. This command does not have input or output argument.
- **Abort** : It aborts a running motion. This command does not have input or output argument.
- **DefinePosition** : Loads a position into controller. It has one input argument which is the new position value (a double). It is allowed only in the ON or ALARM states. The unit used for the command input value is the physical unit: millimeters or milli-radians. It is always an absolute position.
- **SaveConfig** : Write some of the motor parameters in database. Today, it writes the motor acceleration, deceleration, base_rate and velocity into database as motor device properties. It is allowed only in the ON or ALARM states
- **MoveRelative** : Moves the motor by a relative to the current position distance. It has one input argument which is the relative distance (a double). It is allowed only in the ON or ALARM states. The unit used for the command input value is the physical unit: millimeters or milli-radians.

The classical Tango Init command destroys the motor and re-create it.

The attributes

The motor interface supports several attributes which are summarized in the following table:

Name	Data type	Data format	Writable	Memorized	Operator/Expert
Position	Tango::DevDouble	Scalar	R/W	No *	Operator
DialPosition	Tango::DevDouble	Scalar	R	No	Expert
Offset	Tango::DevDouble	Scalar	R/W	Yes	Expert
Acceleration	Tango::DevDouble	Scalar	R/W	Yes	Expert
Base_rate	Tango::DevDouble	Scalar	R/W	Yes	Expert
Deceleration	Tango::DevDouble	Scalar	R/W	Yes	Expert
Velocity	Tango::DevDouble	Scalar	R/W	Yes	Expert
Limit_switches	Tango::DevBoolean	Spectrum	R	No	Expert
SimulationMode	Tango::DevBoolean	Scalar	R	No	Expert
Step_per_unit	Tango::DevDouble	Scalar	R/W	Yes	Expert
Backlash	Tango::DevLong	Scalar	R/W	Yes	Expert

- **Position** : This is read-write scalar double attribute. With the classical Tango min_value and max_value attribute properties, it is easy to define authorized limit for this attribute. See the definition of the DialPosition and Offset attributes to get a precise definition of the meaning of this attribute. It is not allowed to read or write this attribute when the motor is in FAULT or UNKNOWN state. It is also not possible to write this attribute when the motor is already MOVING. The unit used for this attribute is the physical unit e.g. millimeters or milli-radian. It is always an **absolute position** .

- **DialPosition** : This attribute is the motor dial position. The following formula links together the Position, DialPosition, Sign and Offset attributes:

  Position = Sign * DialPosition + Offset

  This allows to have the motor position centered around any position defined by the Offset attribute (classically the X ray beam position). It is a read only attribute. To set the motor position, the user has to use the Position attribute. It is not allowed to read this attribute when the motor is in FAULT or UNKNOWN mode. The unit used for this attribute is the physical unit: millimeters or milli-radian. It is also always an **absolute** position.
- **Offset** : The offset to be applied in the motor position computation. By default set to 0. It is a memorized attribute. It is not allowed to read or write this attribute when the motor is in FAULT, MOVING or UNKNOWN mode.
- **Acceleration** : This is an expert read-write scalar double attribute. This parameter value is written in database when the SaveConfig command is executed. It is not allowed to read or write this attribute when the motor is in FAULT or UNKNOWN state.
- **Deceleration** : This is an expert read-write scalar double attribute. This parameter value is written in database when the SaveConfig command is executed. It is not allowed to read or write this attribute when the motor is in FAULT or UNKNOWN state.
- **Base_rate** : This is an expert read-write scalar double attribute. This parameter value is written in database when the SaveConfig command is executed. It is not allowed to read or write this attribute when the motor is in FAULT or UNKNOWN state.
- **Velocity** : This is an expert read-write scalar double attribute. This parameter value is written in database when the SaveConfig command is executed. It is not allowed to read or write this attribute when the motor is in FAULT or UNKNOWN state.
- **Limit_switches** : Three limit switches are managed by this attribute. Each of the switch are represented by a boolean value: False means inactive while True means active. It is a read only attribute. It is not possible to read this attribute when the motor is in UNKNOWN mode. It is a spectrum attribute with 3 values which are:
  - Data[0] : The Home switch value
  - Data[1] : The Upper switch value
  - Data[2] : The Lower switch value
- **SimulationMode** : This is a read only scalar boolean attribute. When set, all motion requests are not forwarded to the software controller and then to the hardware. When set, the motor position is simulated and is immediately set to the value written by the user. To set this attribute, the user has to used the pool device Tango interface. The value of the position, acceleration, deceleration, base_rate, velocity and offset attributes are memorized at the moment this attribute is set. When this mode is turned off, if the value of any of the previously memorized attributes has changed, it is reapplied to the memorized value. It is not allowed to read this attribute when the motor is in FAULT or UNKNOWN states.
- **Step_per_unit** : This is the number of motor step per millimeter or per degree. It is a memorized attribute. It is not allowed to read or write this attribute when the motor is in FAULT or UNKNOWN mode. It is also not allowed to write this attribute when the motor is MOVING. The default value is 1.
- **Backlash** : If this attribute is defined to something different than 0, the motor will always stop the motion coming from the same mechanical direction. This means that it could be possible to ask the motor to go a little bit after the desired position and then to return to the desired position. The attribute value is the number of steps the motor will pass the desired position if it arrives from the "wrong" direction. This is a signed value. If the sign is positive, this means that the authorized direction to stop the motion is the increasing motor position direction. If the sign is negative, this means that the authorized direction to stop the motion is the decreasing motor position direction. It is a memorized attribute. It is not allowed to read or write this attribute when the motor is in FAULT or UNKNOWN mode. It is also not allowed to write this attribute when the motor is MOVING. Some hardware motor controllers are able to manage this backlash feature. If it is not the case, the motor interface will implement this behavior.

All the motor devices will have the already described attributes but some hardware motor controller supports other features which are not covered by this list of pre-defined attributes. Using Tango dynamic attribute creation, a motor device may have extra attributes used to get/set the motor hardware controller specific features. These are the attributes specified on the controller with `axis_attribues`.

The properties

- **Sleep_bef_last_read** : This property exposes the motor *instability time*. It defines the time in milli-second that the software managing a motor movement will wait between it detects the end of the motion and the last motor position reading.

Getting motor state and limit switches using event

The simplest way to know if a motor is moving is to survey its state. If the motor is moving, its state will be MOVING. When the motion is over, its state will be back to ON (or ALARM if a limit switch has been reached). The pool motor interface allows client interested by motor state or motor limit switches value to use the Tango event system subscribing to motor state change event. As soon as a motor starts a motion, its state is changed to MOVING and an event is sent. As soon as the motion is over, the motor state is updated and another event is sent. In the same way, as soon as a change in the limit switches value is detected, a change event is sent to client(s) which have subscribed to change event on the Limit_Switches attribute.

Reading the motor position attribute

For each motor, the key attribute is its position. Special care has been taken on this attribute management. When the motor is not moving, reading the Position attribute will generate calls to the controller and therefore hardware access. When the motor is moving, its position is automatically read every 100 milli-seconds and stored in the cache. This means that a client reading motor Position attribute while the motor is moving will get the position from the cache and will not generate extra controller calls. It is also possible to get a motor position using the Tango event system. When the motor is moving, an event is sent to the registered clients when the change event criterion is true. By default, this change event criterion is set to be a difference in position of 1. It is tunable on a motor basis using the classical motor Position attribute abs_change property or at the pool device basis using its DefaultMotPos_AbsChange property. Anyway, not more than 10 events could be sent by second. Once the motion is over, the motor position is made unavailable from the Tango polling buffer and is read a last time after a tunable waiting time (Sleep_bef_last_read property). A forced change event with this value is sent to clients using events.

`init`(*name*)
initialize the device once in the object lifetime. Override when necessary but **always** call the method from your super class

Parameters **name** (`str`[720]) – device name

`get_motor`()

`set_motor`(*motor*)

`motor`

`set_write_dial_position_to_db`()

[720] https://docs.python.org/dev/library/stdtypes.html#str

get_write_dial_position_from_db()

delete_device

init_device

on_motor_changed(*event_source*, *event_type*, *event_value*)

always_executed_hook()

read_attr_hardware(*data*)

get_dynamic_attributes()
Override of `PoolDevice.get_dynamic_attributes`. Returns the standard dynamic and fully dynamic attributes for this device. The return is a tuple of two dictionaries:

- standard attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information
- dynamic attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information

tango information seq< `CmdArgType`, `AttrDataFormat`, `AttrWriteType` >
attribute information attribute information as returned by the sardana controller

Returns the standard dynamic and fully dynamic attributes

Return type seq< `CaselessDict`[721], `CaselessDict`[722]>

initialize_dynamic_attributes()
Initializes this device dynamic attributes

read_Position(*attr*)

write_Position(*attr*)

read_Acceleration(*attr*)

write_Acceleration(*attribute*)

read_Deceleration(*attr*)

write_Deceleration(*attribute*)

read_Base_rate(*attr*)

write_Base_rate(*attribute*)

read_Velocity(*attr*)

write_Velocity(*attribute*)

read_Offset(*attr*)

write_Offset(*attribute*)

read_DialPosition(*attr*)

read_Step_per_unit(*attr*)

write_Step_per_unit(*attribute*)

read_Backlash(*attr*)

write_Backlash(*attribute*)

read_Sign(*attr*)

[721] http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict
[722] http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict

write_Sign(*attribute*)

read_Limit_switches(*attr*)

DefinePosition(*argin*)

is_DefinePosition_allowed()

SaveConfig()

is_SaveConfig_allowed()

MoveRelative(*argin*)

is_MoveRelative_allowed()

get_attributes_to_restore()
Make sure position is the last attribute to restore

is_Position_allowed(*req_type*)
Generic is_allowed

is_Acceleration_allowed(*req_type*)
Generic is_allowed

is_Deceleration_allowed(*req_type*)
Generic is_allowed

is_Base_rate_allowed(*req_type*)
Generic is_allowed

is_Velocity_allowed(*req_type*)
Generic is_allowed

is_Offset_allowed(*req_type*)
Generic is_allowed

is_DialPosition_allowed(*req_type*)
Generic is_allowed

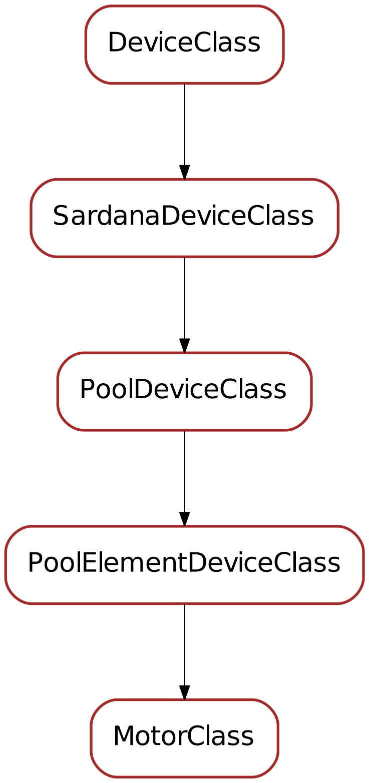
is_Step_per_unit_allowed(*req_type*)
Generic is_allowed

is_Backlash_allowed(*req_type*)
Generic is_allowed

is_Sign_allowed(*req_type*)
Generic is_allowed

is_Limit_switches_allowed(*req_type*)
Generic is_allowed

MotorClass



class MotorClass(*name*)

Bases: *sardana.tango.pool.PoolDevice.PoolElementDeviceClass*

class_property_list = {}

device_property_list = {'Axis': [<_mock._Mock object at 0x7f07dbbd1a10>, 'Axis in the

cmd_list = {'Abort': [[<_mock._Mock object at 0x7f07dbbd1f10>, ''], [<_mock._Mock obj

attr_list = {'Instrument': [[<_mock._Mock object at 0x7f07dbbd1a90>, <_mock._Mock obj

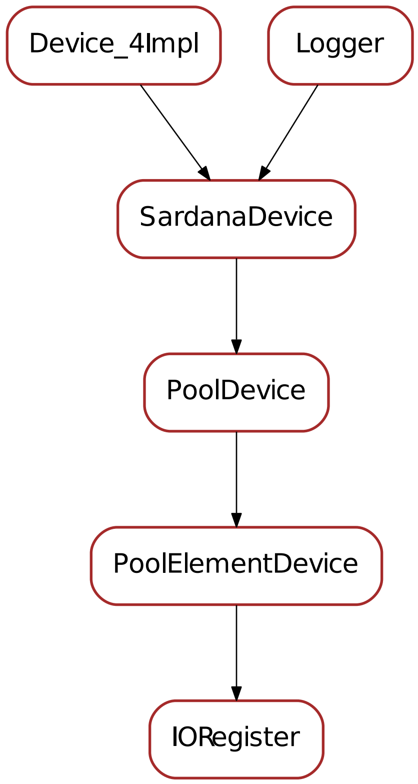
standard_attr_list = {'Acceleration': [[<_mock._Mock object at 0x7f07dbbd18d0>, <_moc

IORegister

Classes

- *IORegister*
- *IORegisterClass*

IORegister



class IORegister(*dclass, name*)

Bases: `sardana.tango.pool.PoolDevice.PoolElementDevice`

init(*name*)

initialize the device once in the object lifetime. Override when necessary but **always** call the method from your super class

Parameters name (`str`[723]) – device name

get_ior()

set_ior(*ior*)

ior

set_write_value_to_db()

get_write_value_from_db()

delete_device

init_device

on_ior_changed(*event_source, event_type, event_value*)

[723] https://docs.python.org/dev/library/stdtypes.html#str

always_executed_hook()

read_attr_hardware(*data*)

get_dynamic_attributes()

Override of `PoolDevice.get_dynamic_attributes`. Returns the standard dynamic and fully dynamic attributes for this device. The return is a tuple of two dictionaries:

- standard attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information
- dynamic attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information

tango information seq< `CmdArgType`, `AttrDataFormat`, `AttrWriteType` >
attribute information attribute information as returned by the sardana controller

Returns the standard dynamic and fully dynamic attributes

Return type seq< `CaselessDict`[724], `CaselessDict`[725]>

initialize_dynamic_attributes()

Initializes this device dynamic attributes

read_Value(*attr*)

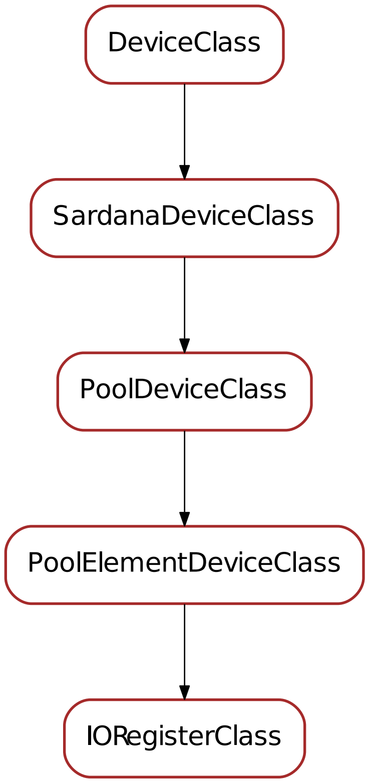
write_Value(*attr*)

is_Value_allowed(*req_type*)

Start()

[724] http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict
[725] http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict

IORegisterClass



class IORegisterClass(*name*)

Bases: *sardana.tango.pool.PoolDevice.PoolElementDeviceClass*

class_property_list = {}

device_property_list = {'Axis': [<_mock._Mock object at 0x7f07dbbd1a10>, 'Axis in the

cmd_list = {'Abort': [[<_mock._Mock object at 0x7f07dbbd1f10>, ''], [<_mock._Mock obj

attr_list = {'Instrument': [[<_mock._Mock object at 0x7f07dbbd1a90>, <_mock._Mock obj

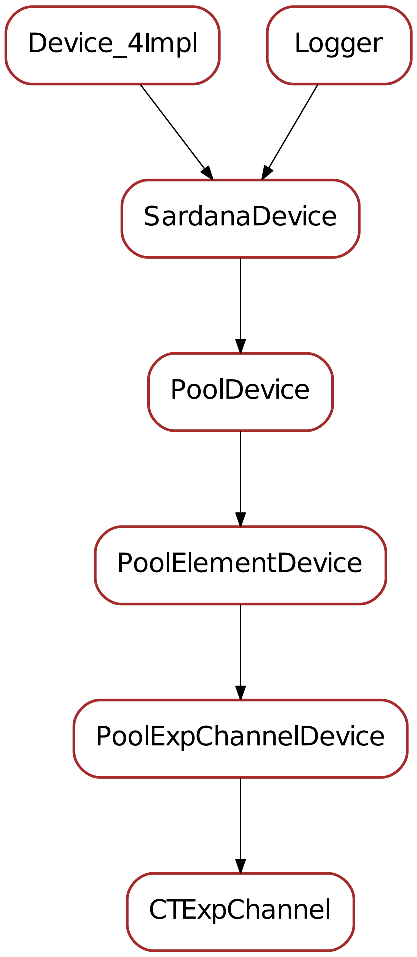
standard_attr_list = {'Value': [[<_mock._Mock object at 0x7f07dbbd18d0>, <_mock._Mock

CTExpChannel

Classes

- *CTExpChannel*
- *CTExpChannelClass*

CTExpChannel



class CTExpChannel (*dclass, name*)
Bases: `sardana.tango.pool.PoolDevice.PoolExpChannelDevice`

init (*name*)
initialize the device once in the object lifetime. Override when necessary but **always** call the method from your super class

Parameters name (`str`[726]) – device name

get_ct ()

set_ct (*ct*)

ct

delete_device

[726] https://docs.python.org/dev/library/stdtypes.html#str

init_device

on_ct_changed(*event_source*, *event_type*, *event_value*)

always_executed_hook()

read_attr_hardware(*data*)

get_dynamic_attributes()

Override of `PoolDevice.get_dynamic_attributes`. Returns the standard dynamic and fully dynamic attributes for this device. The return is a tuple of two dictionaries:

- standard attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information
- dynamic attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information

tango information seq< `CmdArgType`, `AttrDataFormat`, `AttrWriteType` >
attribute information attribute information as returned by the sardana controller

Returns the standard dynamic and fully dynamic attributes

Return type seq< `CaselessDict`[727], `CaselessDict`[728]>

initialize_dynamic_attributes()

Initializes this device dynamic attributes

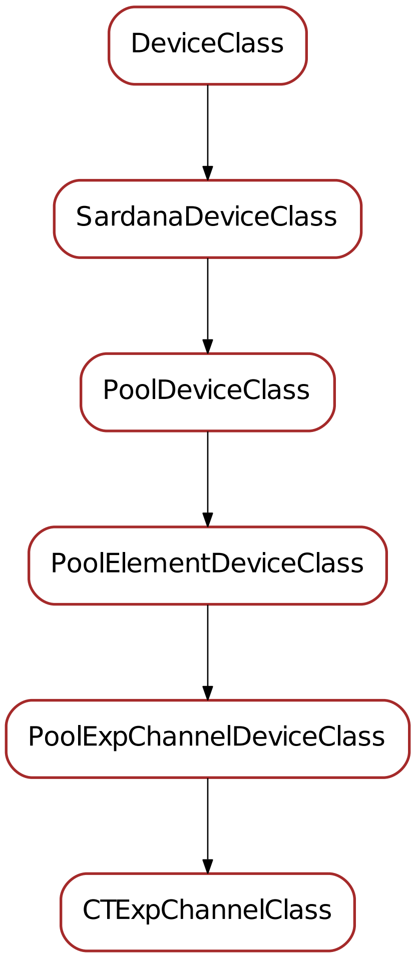
read_Value(*attr*)

is_Value_allowed(*req_type*)

Start()

[727] http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict
[728] http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict

CTExpChannelClass



class CTExpChannelClass(*name*)

Bases: `sardana.tango.pool.PoolDevice.PoolExpChannelDeviceClass`

class_property_list = {}

device_property_list = {'Axis': [<_mock._Mock object at 0x7f07dbbd1a10>, 'Axis in the

cmd_list = {'Abort': [[<_mock._Mock object at 0x7f07dbbd1f10>, ''], [<_mock._Mock obj

attr_list = {'Instrument': [[<_mock._Mock object at 0x7f07dbbd1a90>, <_mock._Mock obj

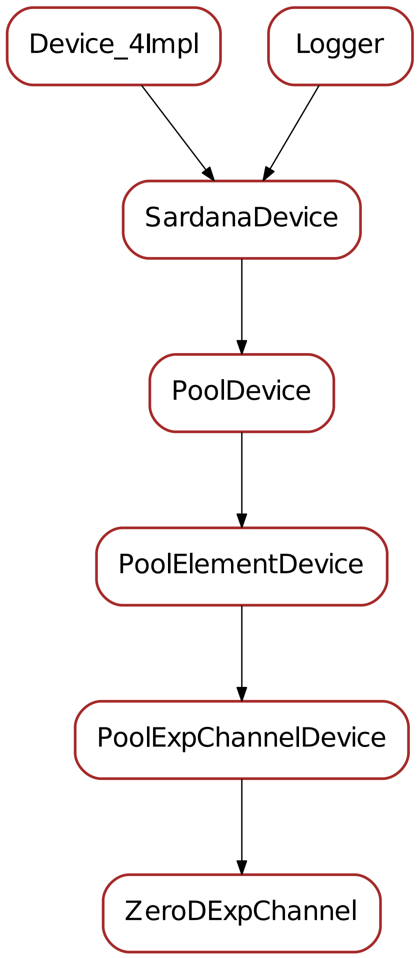
standard_attr_list = {'Data': [[<_mock._Mock object at 0x7f07dbbd1a90>, <_mock._Mock

ZeroDExpChannel

Classes

- *ZeroDExpChannel*
- *ZeroDExpChannelClass*

ZeroDExpChannel



class ZeroDExpChannel(*dclass, name*)
Bases: `sardana.tango.pool.PoolDevice.PoolExpChannelDevice`

init(*name*)
initialize the device once in the object lifetime. Override when necessary but **always** call the method from your super class

Parameters **name** (`str`[729]) – device name

get_zerod()

set_zerod(*zerod*)

zerod

delete_device

init_device

on_zerod_changed(*event_source*, *event_type*, *event_value*)

always_executed_hook()

read_attr_hardware(*data*)

get_dynamic_attributes()
Override of `PoolDevice.get_dynamic_attributes`. Returns the standard dynamic and fully dynamic attributes for this device. The return is a tuple of two dictionaries:

- standard attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information
- dynamic attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information

tango information seq< `CmdArgType`, `AttrDataFormat`, `AttrWriteType` >
attribute information attribute information as returned by the sardana controller

Returns the standard dynamic and fully dynamic attributes

Return type seq< `CaselessDict`[730], `CaselessDict`[731]>

initialize_dynamic_attributes()
Initializes this device dynamic attributes

read_Value(*attr*)

read_CurrentValue(*attr*)

Start()

read_ValueBuffer(*attr*)

read_AccumulationBuffer(*attr*)

read_TimeBuffer(*attr*)

read_AccumulationType(*attr*)

write_AccumulationType(*attr*)

is_Value_allowed(*req_type*)
Generic is_allowed

is_CurrentValue_allowed(*req_type*)
Generic is_allowed

is_AccumulationType_allowed(*req_type*)
Generic is_allowed

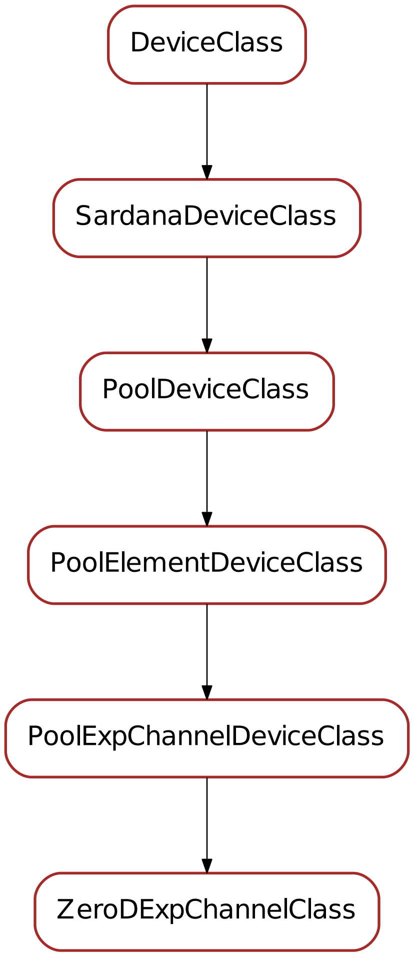
is_ValueBuffer_allowed(*req_type*)
Generic is_allowed

[729] https://docs.python.org/dev/library/stdtypes.html#str
[730] http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict
[731] http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict

is_AccumulationBuffer_allowed(*req_type*)
Generic is_allowed

is_TimeBuffer_allowed(*req_type*)
Generic is_allowed

ZeroDExpChannelClass



class ZeroDExpChannelClass(*name*)
Bases: `sardana.tango.pool.PoolDevice.PoolExpChannelDeviceClass`

class_property_list = {}

device_property_list = {'Axis': [<_mock._Mock object at 0x7f07dbbd1a10>, 'Axis in the

cmd_list = {'Abort': [[<_mock._Mock object at 0x7f07dbbd1f10>, ''], [<_mock._Mock obj

attr_list = {'AccumulationBuffer': [[<_mock._Mock object at 0x7f07dbbd18d0>, <_mock._

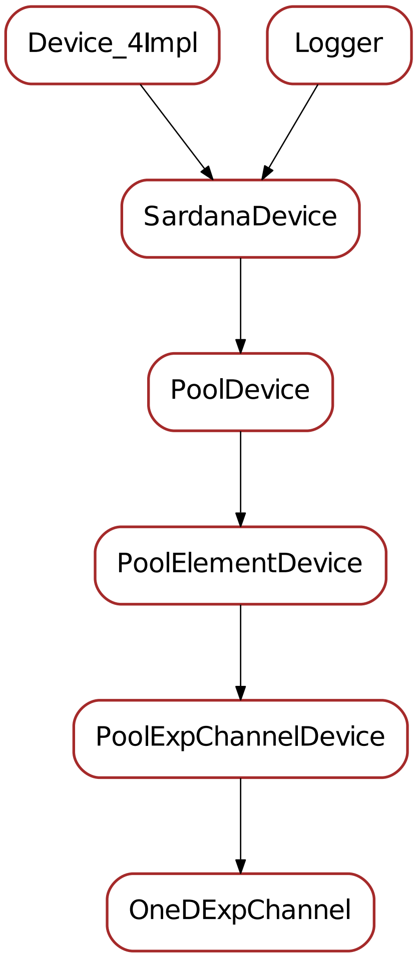
standard_attr_list = {'Data': [[<_mock._Mock object at 0x7f07dbbd1a90>, <_mock._Mock o

OneDExpChannel

Classes

- *OneDExpChannel*
- *OneDExpChannelClass*

OneDExpChannel



class OneDExpChannel(*dclass, name*)
Bases: `sardana.tango.pool.PoolDevice.PoolExpChannelDevice`

init(*name*)
initialize the device once in the object lifetime. Override when necessary but **always** call the method from your super class

Parameters **name** (str[732]) – device name

get_oned()

set_oned(*oned*)

oned

delete_device

init_device

on_oned_changed(*event_source*, *event_type*, *event_value*)

always_executed_hook()

read_attr_hardware(*data*)

get_dynamic_attributes()
Override of `PoolDevice.get_dynamic_attributes`. Returns the standard dynamic and fully dynamic attributes for this device. The return is a tuple of two dictionaries:

- standard attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information
- dynamic attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information

tango information seq< `CmdArgType`, `AttrDataFormat`, `AttrWriteType` >
attribute information attribute information as returned by the sardana controller

Returns the standard dynamic and fully dynamic attributes

Return type seq< CaselessDict[733], CaselessDict[734]>

initialize_dynamic_attributes()
Initializes this device dynamic attributes

read_Value(*attr*)

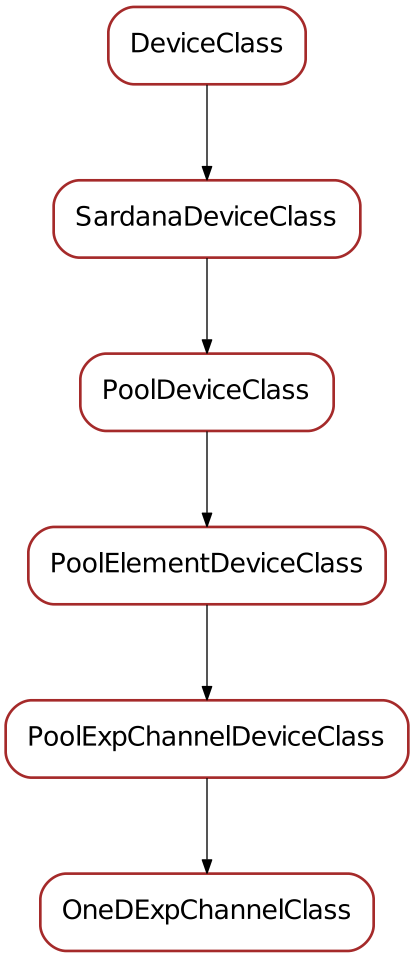
is_Value_allowed(*req_type*)

read_DataSource(*attr*)

Start()

[732] https://docs.python.org/dev/library/stdtypes.html#str
[733] http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict
[734] http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict

OneDExpChannelClass



class OneDExpChannelClass(*name*)

Bases: `sardana.tango.pool.PoolDevice.PoolExpChannelDeviceClass`

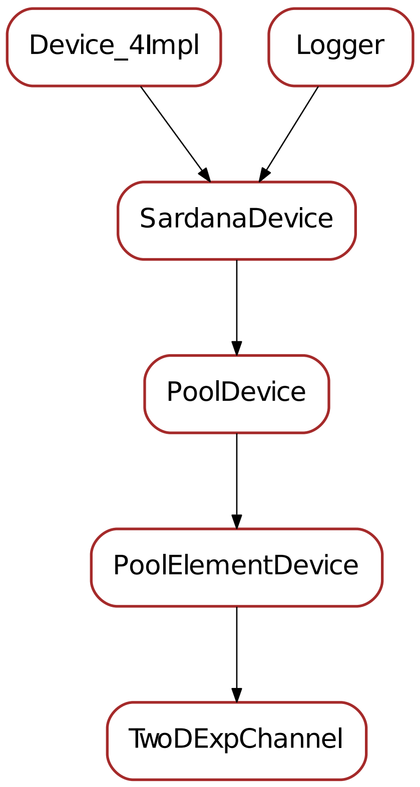
```
class_property_list = {}
device_property_list = {'Axis': [<_mock._Mock object at 0x7f07dbbd1a10>, 'Axis in the
cmd_list = {'Abort': [[<_mock._Mock object at 0x7f07dbbd1f10>, ''], [<_mock._Mock obj
attr_list = {'DataSource': [[<_mock._Mock object at 0x7f07dbbd1a90>, <_mock._Mock obj
standard_attr_list = {'Data': [[<_mock._Mock object at 0x7f07dbbd1a90>, <_mock._Mock
```

TwoDExpChannel

Classes

- *TwoDExpChannel*
- *TwoDExpChannelClass*

TwoDExpChannel



class TwoDExpChannel (*dclass, name*)

Bases: *sardana.tango.pool.PoolDevice.PoolElementDevice*

init (*name*)

initialize the device once in the object lifetime. Override when necessary but **always** call the method from your super class

Parameters name (*str*[735]) – device name

get_twod ()

set_twod (*twod*)

[735] https://docs.python.org/dev/library/stdtypes.html#str

twod

delete_device

init_device

on_twod_changed(*event_source*, *event_type*, *event_value*)

always_executed_hook()

read_attr_hardware(*data*)

get_dynamic_attributes()

Override of `PoolDevice.get_dynamic_attributes`. Returns the standard dynamic and fully dynamic attributes for this device. The return is a tuple of two dictionaries:

- standard attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information
- dynamic attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information

tango information seq< `CmdArgType`, `AttrDataFormat`, `AttrWriteType` >
attribute information attribute information as returned by the sardana controller

Returns the standard dynamic and fully dynamic attributes

Return type seq< `CaselessDict`[736], `CaselessDict`[737]>

read_Value(*attr*)

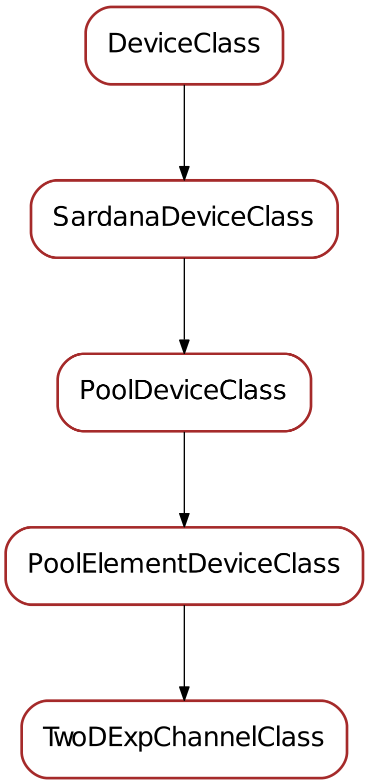
is_Value_allowed(*req_type*)

read_DataSource(*attr*)

Start()

[736] http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict
[737] http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict

TwoDExpChannelClass



class TwoDExpChannelClass(*name*)

Bases: *sardana.tango.pool.PoolDevice.PoolElementDeviceClass*

class_property_list = {}

device_property_list = {'Axis': [<_mock._Mock object at 0x7f07dbbd1a10>, 'Axis in the

cmd_list = {'Abort': [[<_mock._Mock object at 0x7f07dbbd1f10>, ''], [<_mock._Mock obj

attr_list = {'DataSource': [[<_mock._Mock object at 0x7f07dbbd1a90>, <_mock._Mock obj

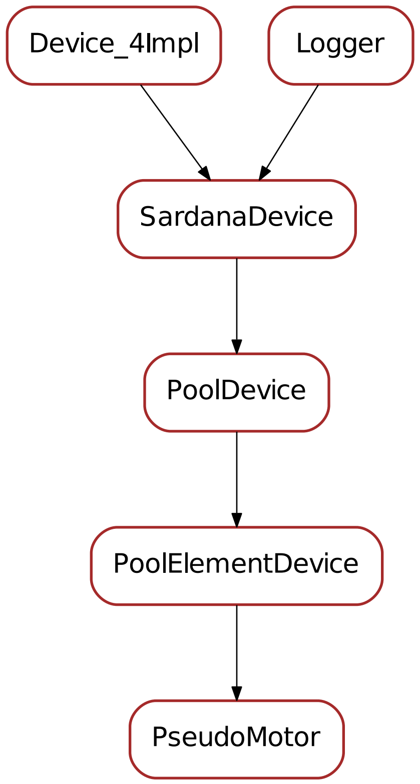
standard_attr_list = {'Value': [[<_mock._Mock object at 0x7f07dbbd1f10>, <_mock._Mock

PseudoMotor

Classes

- *PseudoMotor*
- *PseudoMotorClass*

PseudoMotor



class PseudoMotor(*dclass*, *name*)

Bases: `sardana.tango.pool.PoolDevice.PoolElementDevice`

init(*name*)

initialize the device once in the object lifetime. Override when necessary but **always** call the method from your super class

Parameters name (`str`[738]) – device name

get_pseudo_motor()

set_pseudo_motor(*pseudo_motor*)

pseudo_motor

delete_device

init_device

on_pseudo_motor_changed(*event_source*, *event_type*, *event_value*)

always_executed_hook()

read_attr_hardware(*data*)

[738] https://docs.python.org/dev/library/stdtypes.html#str

get_dynamic_attributes()
Override of `PoolDevice.get_dynamic_attributes`. Returns the standard dynamic and fully dynamic attributes for this device. The return is a tuple of two dictionaries:

- standard attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information
- dynamic attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information

tango information seq< `CmdArgType`, `AttrDataFormat`, `AttrWriteType` >
attribute information attribute information as returned by the sardana controller

Returns the standard dynamic and fully dynamic attributes

Return type seq< `CaselessDict`[739], `CaselessDict`[740]>

initialize_dynamic_attributes()
Initializes this device dynamic attributes

read_Position(*attr*)

write_Position(*attr*)

CalcPseudo(*physical_positions*)
Returns the pseudo motor position for the given physical positions

CalcPhysical(*pseudo_position*)
Returns the physical motor positions for the given pseudo motor position assuming the current pseudo motor write positions for all the other sibling pseudo motors

CalcAllPhysical(*pseudo_positions*)
Returns the physical motor positions for the given pseudo motor position(s)

CalcAllPseudo(*physical_positions*)
Returns the pseudo motor position(s) for the given physical positions

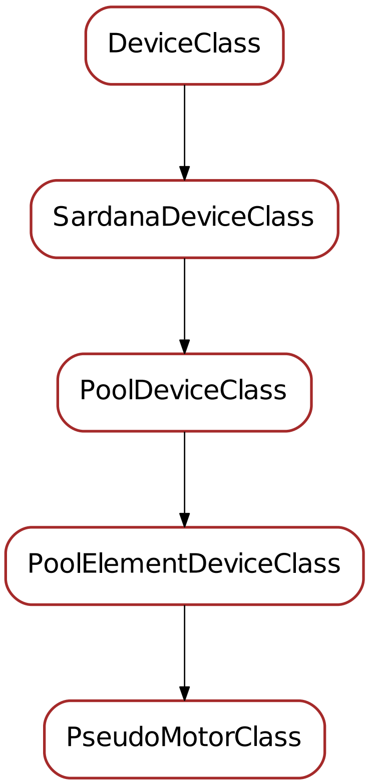
MoveRelative(*argin*)

is_MoveRelative_allowed()

is_Position_allowed(*req_type*)
Generic is_allowed

[739] http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict
[740] http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict

PseudoMotorClass



class PseudoMotorClass(*name*)

Bases: *sardana.tango.pool.PoolDevice.PoolElementDeviceClass*

class_property_list = {}

device_property_list = {'Axis': [<_mock._Mock object at 0x7f07dbbd1a10>, 'Axis in the

cmd_list = {'Abort': [[<_mock._Mock object at 0x7f07dbbd1f10>, ''], [<_mock._Mock obj

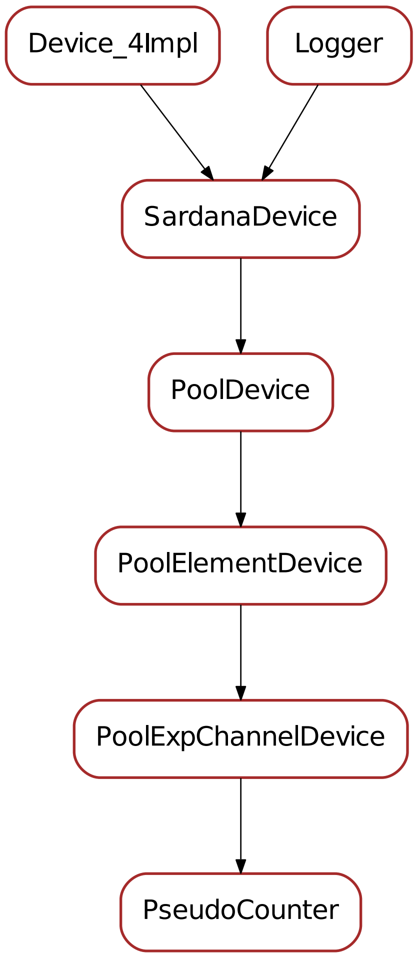
standard_attr_list = {'Position': [[<_mock._Mock object at 0x7f07dbbd18d0>, <_mock._M

PseudoCounter

Classes

- *PseudoCounter*
- *PseudoCounterClass*

PseudoCounter



class PseudoCounter(*dclass*, *name*)

Bases: `sardana.tango.pool.PoolDevice.PoolExpChannelDevice`

init(*name*)

initialize the device once in the object lifetime. Override when necessary but **always** call the method from your super class

Parameters name (`str`[741]) – device name

get_pseudo_counter()

set_pseudo_counter(*pseudo_counter*)

pseudo_counter

delete_device

[741] https://docs.python.org/dev/library/stdtypes.html#str

init_device

on_pseudo_counter_changed(*event_source*, *event_type*, *event_value*)

always_executed_hook()

read_attr_hardware(*data*)

get_dynamic_attributes()

Override of `PoolDevice.get_dynamic_attributes`. Returns the standard dynamic and fully dynamic attributes for this device. The return is a tuple of two dictionaries:

- standard attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information
- dynamic attributes: caseless dictionary with key being the attribute name and value is a tuple of attribute name(str), tango information, attribute information

tango information seq< `CmdArgType`, `AttrDataFormat`, `AttrWriteType` >
attribute information attribute information as returned by the sardana controller

Returns the standard dynamic and fully dynamic attributes

Return type seq< `CaselessDict`[742], `CaselessDict`[743]>

initialize_dynamic_attributes()

Initializes this device dynamic attributes

read_Value(*attr*)

is_Value_allowed(*req_type*)

Generic is_allowed

CalcPseudo(*physical_values*)

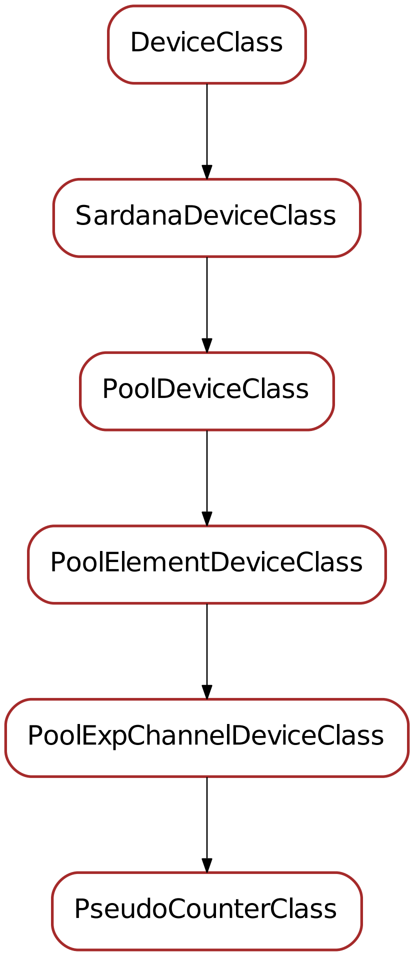
Returns the pseudo counter value for the given physical counters

CalcAllPseudo(*physical_values*)

Returns the pseudo counter values for the given physical counters

[742] http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict
[743] http://taurus-scada.org/devel/api/taurus/core/util/_CaselessDict.html#taurus.core.util.CaselessDict

PseudoCounterClass



class PseudoCounterClass(*name*)

Bases: `sardana.tango.pool.PoolDevice.PoolExpChannelDeviceClass`

class_property_list = {}

device_property_list = {'Axis': [<_mock._Mock object at 0x7f07dbbd1a10>, 'Axis in the

cmd_list = {'Abort': [[<_mock._Mock object at 0x7f07dbbd1f10>, ''], [<_mock._Mock obj

standard_attr_list = {'Data': [[<_mock._Mock object at 0x7f07dbbd1a90>, <_mock._Mock o

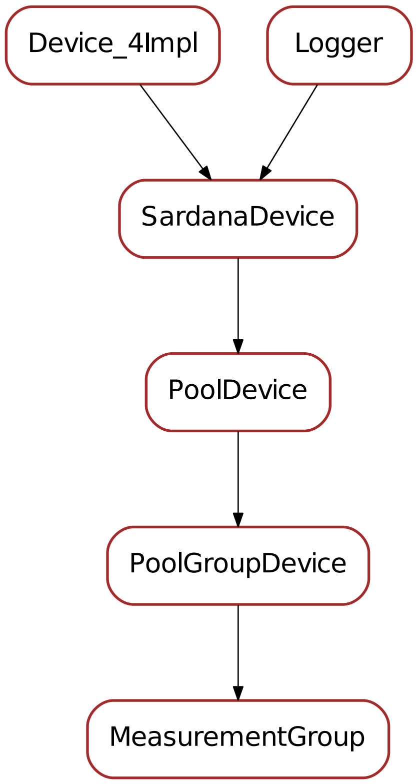
MeasurementGroup

The sardana tango measurement group module

Classes

- *MeasurementGroup*
- *MeasurementGroupClass*

MeasurementGroup



class MeasurementGroup(*dclass*, *name*)

Bases: *sardana.tango.pool.PoolDevice.PoolGroupDevice*

init(*name*)

initialize the device once in the object lifetime. Override when necessary but **always** call the method from your super class

Parameters name (*str*[744]) – device name

get_measurement_group()

set_measurement_group(*measurement_group*)

measurement_group

delete_device

[744] https://docs.python.org/dev/library/stdtypes.html#str

init_device

on_measurement_group_changed(*event_source, event_type, event_value*)

always_executed_hook()

read_attr_hardware(*data*)

read_IntegrationTime(*attr*)

write_IntegrationTime(*attr*)

read_MonitorCount(*attr*)

write_MonitorCount(*attr*)

read_AcquisitionMode(*attr*)

write_AcquisitionMode(*attr*)

read_Configuration(*attr*)

write_Configuration(*attr*)

read_Repetitions(*attr*)

write_Repetitions(*attr*)

read_Moveable(*attr*)

write_Moveable(*attr*)

read_Synchronization(*attr*)

write_Synchronization(*attr*)

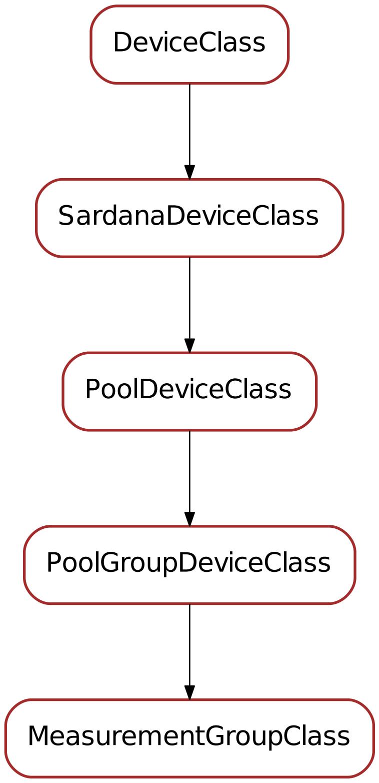
read_LatencyTime(*attr*)

Start()

Stop()
The tango stop command. Stops the active operation

StartMultiple(*n*)

MeasurementGroupClass



class MeasurementGroupClass(*name*)

Bases: *sardana.tango.pool.PoolDevice.PoolGroupDeviceClass*

class_property_list = {}

device_property_list = {'Elements': [<_mock._Mock object at 0x7f07dbbd1e10>, 'element

cmd_list = {'Abort': [[<_mock._Mock object at 0x7f07dbbd1f10>, ''], [<_mock._Mock obj

attr_list = {'AcquisitionMode': [[<_mock._Mock object at 0x7f07dbbd1a90>, <_mock._Moc

`macroserver`

Modules

`macroexecutor`

Functions

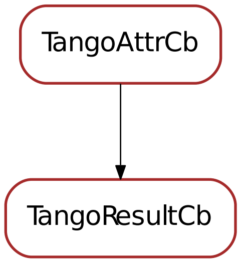
Classes

- `BaseMacroExecutor`
- `MacroExecutorFactory`

TangoAttrCb

class TangoAttrCb(*tango_macro_executor*)
  An abstract callback class for Tango events

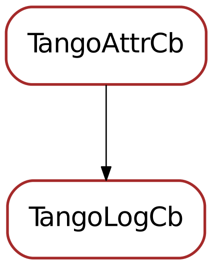
TangoResultCb



class TangoResultCb(*tango_macro_executor*)
  Callback class for Tango events of the Result attribute

  push_event(**args*, ***kwargs*)
    callback method receiving the event

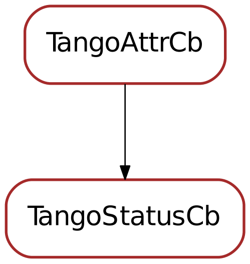
TangoLogCb



class TangoLogCb(*tango_macro_executor*, *log_name*)
Callback class for Tango events of the log attributes e.g. Output, Error, Critical

push_event(**args*, ***kwargs*)
callback method receiving the event

TangoStatusCb



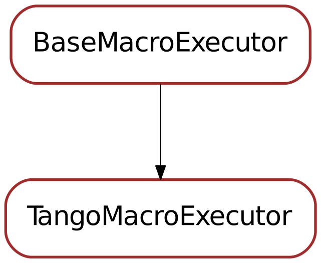
class TangoStatusCb(*tango_macro_executor*)
Callback class for Tango events of the MacroStatus attribute

START_STATES = ['start']

DONE_STATES = ['finish', 'stop', 'exception']

push_event(**args*, ***kwargs*)
callback method receiving the event

TangoMacroExecutor



class TangoMacroExecutor(*door_name=None*)

Macro executor implemented using Tango communication with the Door device

getData()

Returns the data object for the last executed macro

Returns (obj)

createCommonBuffer()

Create a common buffer, where all the registered logs will be stored.

getCommonBuffer()

Get common buffer. Method getCommonBuffer can only be used if at least one buffer exists.

Returns

(seq<str>) list of strings with messages from all log levels

See also:

createCommonBuffer()

getExceptionStr()

Get macro exception type representation (None if the macro state is not exception).

Returns (str)

getLog(*log_level*)

Get log messages.

Parameters **log_level** – (str) string indicating the log level

Returns (seq<str>) list of strings with log messages

getResult()

Get macro result.

Returns (seq<str>) list of strings with Result messages

getState()

Get macro execution state.

Returns (str)

getStateBuffer()

Get buffer (history) of macro execution states.

Returns (seq<str>)

`log_levels = ['debug', 'output', 'info', 'warning', 'critical', 'error']`

`registerAll`()
Register for macro result, all log levels and common buffer.

`registerLog`(*log_level*)
Start registering log messages.
Parameters **`log_level`** – (str) string indicating the log level

`registerResult`()
Register for macro result

`run`(*macro_name*, *macro_params=None*, *sync=True*, *timeout=inf*)
Execute macro.
Parameters

- **`macro_name`** – (string) name of macro to be executed
- **`macro_params`** – (list<string>) macro parameters (default is macro_params=None for macros without parameters or with the default values)
- **`sync`** – (bool) whether synchronous or asynchronous call (default is sync=True)
- **`timeout`** –

  (float) timeout (in s) that will be passed to the wait method, in case of synchronous execution

  In asyncrhonous execution method *`wait()`* has to be explicitly called.

`stop`(*started_event_timeout=3.0*)
Stop macro execution. Execute macro in synchronous way before using this method.
Parameters **`started_event_timeout`** – (float) waiting timeout for started event

`unregisterAll`()
Unregister macro result, all log levels and common buffer.

`unregisterLog`(*log_level*)
Stop registering log messages.
Parameters **`log_level`** – (str) string indicating the log level

`unregisterResult`()
Unregister macro result.

`wait`(*timeout=inf*)
Wait until macro is done. Use it in asynchronous executions.
Parameters **`timeout`** – (float) waiting timeout (in s)

Modules

`sardanadefs`

This module contains the most generic sardana constants and enumerations

Constants

`EpsilonError = 1e-16`
maximum difference between two floats so that they are considered equal

`InvalidId = 0`
A constant representing an invalid ID

`InvalidAxis = 0`
A constant representing an invalid axis

`TYPE_ELEMENTS = set([<_mock._Mock object>, <_mock._Mock object>, <_mock._Mock object>, <_mo`
a set containning all "controllable" element types. Constant values belong to *`ElementType`*

`TYPE_GROUP_ELEMENTS = set([<_mock._Mock object>, <_mock._Mock object>])`
a set containing all group element types. Constant values belong to *`ElementType`*

`TYPE_MOVEABLE_ELEMENTS = set([<_mock._Mock object>, <_mock._Mock object>, <_mock._Mock obje`
a set containing the type of elements which are moveable. Constant values belong to *`ElementType`*

`TYPE_PHYSICAL_ELEMENTS = set([<_mock._Mock object>, <_mock._Mock object>, <_mock._Mock obje`
a set containing the possible types of physical elements. Constant values belong to *`ElementType`*

`TYPE_ACQUIRABLE_ELEMENTS = set([<_mock._Mock object>, <_mock._Mock object>, <_mock._Mock ob`
a set containing the possible types of acquirable elements. Constant values belong to *`ElementType`*

`TYPE_PSEUDO_ELEMENTS = set([<_mock._Mock object>, <_mock._Mock object>])`
a set containing the possible types of pseudo elements. Constant values belong to *`ElementType`*

`SardanaServer = SardanaServer()`
the global object containing the SardanaServer information

Enumerations

`ServerRunMode = <taurus.core.util.enumeration.Enumeration object>`

`State = <taurus.core.util.enumeration.Enumeration object>`

`DataType = <taurus.core.util.enumeration.Enumeration object>`

`DataFormat = <taurus.core.util.enumeration.Enumeration object>`

`DataAccess = <taurus.core.util.enumeration.Enumeration object>`

`ElementType = <taurus.core.util.enumeration.Enumeration object>`

`Interface = <taurus.core.util.enumeration.Enumeration object>`

`Interfaces = {<_mock._Mock object at 0x7f07db79c0d0>: set([]), <_mock._Mock object at 0x7`
a dictionary containing the direct interfaces supported by each type (`dict`[745] <*`sardana.sardanadefs.Interface`*, `set`[746] < *`sardana.sardanadefs.Interface`*> >)

`InterfacesExpanded = {<_mock._Mock object at 0x7f07db79c0d0>: set([<_mock._Mock object at`
a dictionary containing the *all* interfaces supported by each type. (`dict`[747] <*`sardana.sardanadefs.Interface`*, `set`[748] < *`sardana.sardanadefs.Interface`*> >)

[745] https://docs.python.org/dev/library/stdtypes.html#dict
[746] https://docs.python.org/dev/library/stdtypes.html#set
[747] https://docs.python.org/dev/library/stdtypes.html#dict
[748] https://docs.python.org/dev/library/stdtypes.html#set

INTERFACES = {'Acquirable': (set(['PoolElement']), 'An acquirable element'), 'CTExpChanne
a dictionary containing the direct interfaces supported by each type (`dict`[749]<`str`[750], `tuple`[751]<`set`[752]<`str`[753], `str`[754]>>>)

INTERFACES_EXPANDED = {'Acquirable': (set(['PoolElement', 'Object', 'Acquirable', 'PoolOb
a dictionary containing the *all* interfaces supported by each type (`dict`[755] <`str`[756], `set`[757] < `str`[758]> >)

Functions

from_dtype_str(*dtype*)
Transforms the given dtype parameter (string/*DataType* or None) into a tuple of two elements (str, *DataFormat*) where the first element is a string with a simplified data type.

- If None is given, it returns ('float', `DataFormat.Scalar`)
- If *DataType* is given, it returns (*DataType*, `DataFormat.Scalar`)

Parameters **dtype** (str or None or *DataType*) – the data type to be transformed
Returns a tuple <str, *DataFormat*> for the given dtype
Return type tuple<str, *DataFormat*>

from_access_str(*access*)
Transforms the given access parameter (string or *DataAccess*) into a simplified data access string.
Parameters **dtype** (*str*[759]) – the access to be transformed
Returns a simple string for the given access
Return type str[760]

to_dtype_dformat(*data*)
Transforms the given data parameter (string/ or sequence of string or sequence of sequence of string/*DataType*) into a tuple of two elements (*DataType*, *DataFormat*).
Parameters **data** (*str*[761] *or seq<str> or seq<seq<str>>*) – the data information to be transformed
Returns a tuple <*DataType*, *DataFormat*> for the given data
Return type tuple<*DataType*, *DataFormat*>

to_daccess(*daccess*)
Transforms the given access parameter (string or None) into a *DataAccess*. If None is given returns `DataAccess.ReadWrite`
Parameters **dtype** (*str*[762]) – the access to be transformed
Returns a *DataAccess* for the given access
Return type *DataAccess*

[749] https://docs.python.org/dev/library/stdtypes.html#dict
[750] https://docs.python.org/dev/library/stdtypes.html#str
[751] https://docs.python.org/dev/library/stdtypes.html#tuple
[752] https://docs.python.org/dev/library/stdtypes.html#set
[753] https://docs.python.org/dev/library/stdtypes.html#str
[754] https://docs.python.org/dev/library/stdtypes.html#str
[755] https://docs.python.org/dev/library/stdtypes.html#dict
[756] https://docs.python.org/dev/library/stdtypes.html#str
[757] https://docs.python.org/dev/library/stdtypes.html#set
[758] https://docs.python.org/dev/library/stdtypes.html#str
[759] https://docs.python.org/dev/library/stdtypes.html#str
[760] https://docs.python.org/dev/library/stdtypes.html#str
[761] https://docs.python.org/dev/library/stdtypes.html#str
[762] https://docs.python.org/dev/library/stdtypes.html#str

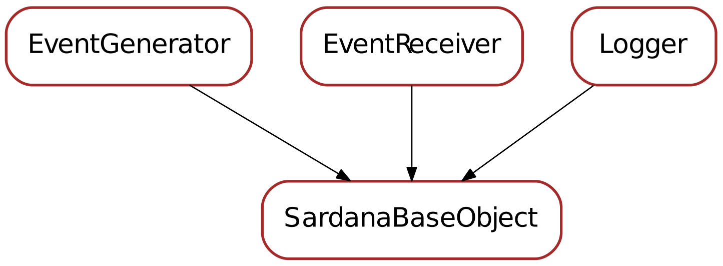
sardanabase

This module is part of the Python Sardana library. It defines the base classes for Sardana object

Classes

- *SardanaBaseObject*
- *SardanaObjectID*

SardanaBaseObject



class SardanaBaseObject(***kwargs*)

The Sardana most abstract object. It contains only two members:

- _manager : a weak reference to the manager (pool or ms) where it belongs
- _name : the name
- _full_name : the name (usually a tango device name, but can be anything else.)

get_manager()

Return the `sardana.Manager` which *owns* this sardana object.

Returns the manager which *owns* this pool object.

Return type `sardana.Manager`

get_name()

Returns this sardana object name

Returns this sardana object name

Return type str[763]

set_name(*name*)

Sets sardana object name

Param sardana object name

Type str

get_full_name()

Returns this sardana object full name

Returns this sardana object full name

[763] https://docs.python.org/dev/library/stdtypes.html#str

Return type str[764]

get_type()
Returns this sardana object type.
Returns this sardana object type

Return type *ElementType*

get_parent()
Returns this pool object parent.
Returns this objects parent

Return type *SardanaBaseObject*

get_parent_name()
Returns this sardana object parent's name.
Returns this objects parent

Return type str[765]

get_frontend()
Returns this sardana frontend object or None if no frontend is registered
Returns this objects frontend

Return type object[766]

fire_event(*event_type*, *event_value*, *listeners=None*, *protected=True*)

get_interfaces()
Returns the set of interfaces this object implements.
Returns The set of interfaces this object implements.

Return type class:*set* <*sardana.sardanadefs.Interface*>

get_interface()
Returns the interface this object implements.
Returns The interface this object implements.

Return type *sardana.sardanadefs.Interface*

get_interface_names()
Returns a sequence of interface names this object implements.
Returns The sequence of interfaces this object implements.

Return type sequence<*str*>

serialize(**args*, ***kwargs*)

serialized(**args*, ***kwargs*)

str(**args*, ***kwargs*)

manager
reference to the `sardana.Manager`

name
object name

full_name
object full name

[764] https://docs.python.org/dev/library/stdtypes.html#str
[765] https://docs.python.org/dev/library/stdtypes.html#str
[766] https://docs.python.org/dev/library/functions.html#object

frontend
  the object frontend

Critical = 50

Debug = 10

DftLogLevel = 20

DftLogMessageFormat = '%(threadName)-14s %(levelname)-8s %(asctime)s %(name)s: %(mess

Error = 40

Fatal = 50

Info = 20

Trace = 5

Warning = 30

add_listener(*listener*)
  Adds a new listener for this object.
    Parameters **listener** – a listener

are_events_blocked()

block_events()

flush_queue()

has_listeners()
  Returns True if anybody is listening to events from this object
    Returns True is at least one listener is listening or False otherwise

log_level = 20

queue_event(*event_type*, *event_value*, *listeners=None*)

remove_listener(*listener*)
  Removes an existing listener for this object.
    Parameters **listener** – the listener to be removed
    Returns True is succeeded or False otherwise

root_inited = True

unblock_events()

SardanaObjectID

class SardanaObjectID(*id=0*)
  To be used by sardana objects which have an ID associated to them.

get_id()
Returns this sardana object ID
Returns this sardana object ID
Return type int[767]

serialize(**args*, ***kwargs*)

id
object ID

sardanacontainer

This module is part of the Python Pool libray. It defines the base classes for a pool container element

Classes

- *SardanaContainer*

SardanaContainer

class SardanaContainer
A container class for sardana elements

add_element(*e*)
Adds a new `pool.PoolObject` to this container
Parameters **e** (`pool.PoolObject`) – the pool element to be added

remove_element(*e*)
Removes the `pool.PoolObject` from this container
Parameters **e** (`pool.PoolObject`) – the pool object to be removed
Throw KeyError

get_element_id_map()
Returns a reference to the internal pool object ID map
Returns the internal pool object ID map
Return type dict<id, pool.PoolObject>

get_element_name_map()
Returns a reference to the internal pool object name map
Returns the internal pool object name map

[767] https://docs.python.org/dev/library/functions.html#int

Return type dict<str, pool.PoolObject>

get_element_type_map()
Returns a reference to the internal pool object type map

Returns the internal pool object type map

Return type dict<pool.ElementType, dict<id, pool.PoolObject>>

get_element(***kwargs*)
Returns a reference to the requested pool object

Parameters **kwargs** – if key 'id' given: search by ID else if key 'full_name' given: search by full name else if key 'name' given: search by name

Returns the pool object

Return type pool.PoolObject

Throw KeyError

get_element_by_name(*name*, ***kwargs*)
Returns a reference to the requested pool object

Parameters **name** (*str*[768]) – pool object name

Returns the pool object

Return type pool.PoolObject

Throw KeyError

get_element_by_full_name(*full_name*, ***kwargs*)
Returns a reference to the requested pool object

Parameters **name** (*str*[769]) – pool object full name

Returns the pool object

Return type pool.PoolObject

Throw KeyError

get_element_by_id(*id*, ***kwargs*)
Returns a reference to the requested pool object

Parameters **id** (*int*[770]) – pool object ID

Returns the pool object

Return type pool.PoolObject

Throw KeyError

get_elements_by_type(*t*)
Returns a list of all pool objects of the given type

Parameters **t** (*pool.ElementType*) – element type

Returns list of pool objects

Return type seq<pool.PoolObject>

get_element_names_by_type(*t*)
Returns a list of all pool object names of the given type

Parameters **t** (*pool.ElementType*) – element type

Returns list of pool object names

[768] https://docs.python.org/dev/library/stdtypes.html#str
[769] https://docs.python.org/dev/library/stdtypes.html#str
[770] https://docs.python.org/dev/library/functions.html#int

Return type seq<str>

rename_element (*old_name, new_name*)
Rename an object
Parameters
- **old_name** (*str*[771]) – old object name
- **new_name** (*str*[772]) – new object name

check_element (*name, full_name*)

sardanaevent

This module is part of the Python Pool libray. It defines the base classes for pool event mechanism

Classes

- *EventGenerator*
- *EventReceiver*
- *EventType*

EventGenerator

class EventGenerator (*max_queue_len=10, listeners=None*)
A class capable of generating events to their listeners

add_listener (*listener*)
Adds a new listener for this object.
Parameters listener – a listener

remove_listener (*listener*)
Removes an existing listener for this object.
Parameters listener – the listener to be removed
Returns True is succeeded or False otherwise

has_listeners ()
Returns True if anybody is listening to events from this object
Returns True is at least one listener is listening or False otherwise

fire_event (*event_type, event_value, listeners=None*)

queue_event (*event_type, event_value, listeners=None*)

[771] https://docs.python.org/dev/library/stdtypes.html#str
[772] https://docs.python.org/dev/library/stdtypes.html#str

flush_queue()

EventReceiver

class EventReceiver

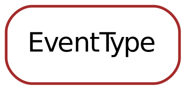
A simple class that implements useful features for a class which is an event receiver. The actual class may inherit from this EventReceiver class and may choose to use just a part of the API provided by this class, the whole API or none of the API.

block_events()

unblock_events()

are_events_blocked()

EventType



class EventType(*name*, *priority=0*)

Definition of an event type

get_name()

Returns this event name

Returns this event name

Return type str[773]

get_priority()

Returns this event priority

Returns this event priority

Return type str[774]

[773] https://docs.python.org/dev/library/stdtypes.html#str
[774] https://docs.python.org/dev/library/stdtypes.html#str

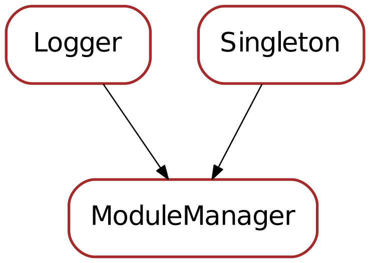
sardanamodulemanager

This module is part of the Python Sardana library. It defines the base classes for module manager

Classes

- *ModuleManager*

ModuleManager



class ModuleManager
This class handles python module loading/reloading and unloading.

init (**args, **kwargs*)
Singleton instance initialization.

reInit ()

cleanUp ()

reset_python_path ()

remove_python_path (*path_id*)

add_python_path (*path*)

findFullModuleName (*module_name, path=None*)

isValidModule (*module_name, path=None*)
Method to verify is a module is loadable.

reloadModule (*module_name, path=None, reload=True*)
Loads/reloads the given module name

deep_reload_module (*module_name, path=None, exclude=None*)

loadModule (*module_name, path=None*)
Loads the given module name. If the module has been already loaded into this python interpreter, nothing is done.

Parameters

- **module_name** (*str*[775]) – the module to be loaded.
- **path** (*seq<str> or None*[776]) – list of paths to look for modules [default: None]

Returns python module

Raises ImportError

unloadModule(*module_name*)
Unloads the given module name

unloadModules(*module_list=None*)
Unloads the given module name

getModule(*module_name*)
Returns the module object for the given module name

getModuleNames()

Critical = 50

Debug = 10

DftLogLevel = 20

DftLogMessageFormat = '%(threadName)-14s %(levelname)-8s %(asctime)s %(name)s: %(mess

Error = 40

Fatal = 50

Info = 20

Trace = 5

Warning = 30

log_level = 20

root_inited = True

sardanameta

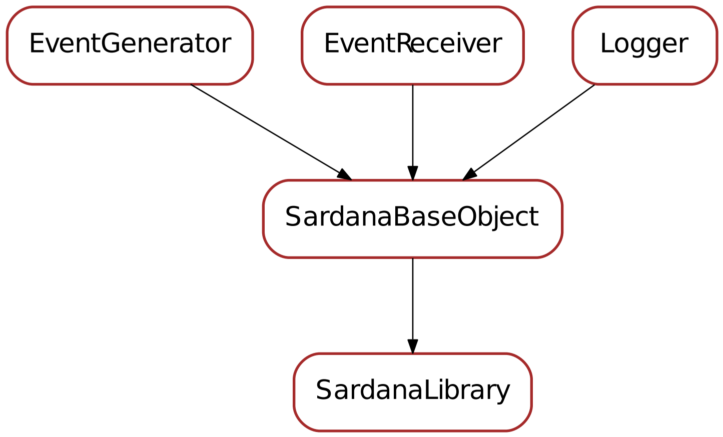
This module is part of the Python Sardana libray. It defines the base classes for MetaLibrary and MetaClass

Classes

- *SardanaLibrary*
- *SardanaClass*

[775] https://docs.python.org/dev/library/stdtypes.html#str
[776] https://docs.python.org/dev/library/constants.html#None

SardanaLibrary



class SardanaLibrary(***kwargs*)

Object representing a python module containing sardana classes. Public members:

- module - reference to python module
- file_path - complete (absolute) path (with file name at the end)
- file_name - file name (including file extension)
- path - complete (absolute) path
- name - (=module name) module name (without file extension)
- meta_classes - dict<str, SardanMetaClass>
- **exc_info - exception information if an error occurred when loading** the module

description = '<Undocumented>'

module_name

Returns the module name for this library.

Returns the module name

Return type str[777]

code

Returns a sequence of sourcelines corresponding to the module code.

Returns list of source code lines

Return type list<str>

add_meta_class(*meta_class*)

Adds a new :class:~'sardana.sardanameta.SardanaClass' to this library.

Parameters meta_class (`:class:~`sardana.sardanameta.SardanaClass``) – the meta class to be added to this library

get_meta_class(*meta_class_name*)

Returns a :class:~'sardana.sardanameta.SardanaClass' for the given meta class name or None if the meta class does not exist in this library.

[777] https://docs.python.org/dev/library/stdtypes.html#str

Parameters **meta_class_name** (*str*[778]) – the meta class name

Returns a meta class or None

Return type :class:~'sardana.sardanameta.SardanaClass'

get_meta_classes()
Returns a sequence of the meta classes that belong to this library.

Returns a sequence of meta classes that belong to this library

Return type seq<:class:~'sardana.sardanameta.SardanaClass'>

has_meta_class(*meta_class_name*)
Returns True if the given meta class name belongs to this library or False otherwise.

Parameters **meta_class_name** (*str*[779]) – the meta class name

Returns True if the given meta class name belongs to this library or False otherwise

Return type bool[780]

add_meta_function(*meta_function*)
Adds a new :class:~'sardana.sardanameta.SardanaFunction' to this library.

Parameters **meta_function** (*:class:~`sardana.sardanameta.SardanaFunction`*) – the meta function to be added to this library

get_meta_function(*meta_function_name*)
Returns a :class:~'sardana.sardanameta.SardanaFunction' for the given meta function name or None if the meta function does not exist in this library.

Parameters **meta_function_name** (*str*[781]) – the meta function name

Returns a meta function or None

Return type :class:~'sardana.sardanameta.SardanaFunction'

get_meta_functions()
Returns a sequence of the meta functions that belong to this library.

Returns a sequence of meta functions that belong to this library

Return type seq<:class:~'sardana.sardanameta.SardanaFunction'>

has_meta_function(*meta_function_name*)
Returns True if the given meta function name belongs to this library or False otherwise.

Parameters **meta_function_name** (*str*[782]) – the meta function name

Returns True if the given meta function name belongs to this library or False otherwise

Return type bool[783]

get_meta(*meta_name*)
Returns a :class:~'sardana.sardanameta.SardanaCode' for the given meta name or None if the meta does not exist in this library.

Parameters **meta_name** (*str*[784]) – the meta name (class, function)

Returns a meta or None

Return type :class:~'sardana.sardanameta.SardanaCode'

[778] https://docs.python.org/dev/library/stdtypes.html#str
[779] https://docs.python.org/dev/library/stdtypes.html#str
[780] https://docs.python.org/dev/library/functions.html#bool
[781] https://docs.python.org/dev/library/stdtypes.html#str
[782] https://docs.python.org/dev/library/stdtypes.html#str
[783] https://docs.python.org/dev/library/functions.html#bool
[784] https://docs.python.org/dev/library/stdtypes.html#str

has_meta(*meta_name*)
Returns True if the given meta name belongs to this library or False otherwise.

Parameters **meta_name** (`str`[785]) – the meta name

Returns True if the given meta (class or function) name belongs to this library or False otherwise

Return type bool[786]

has_metas()
Returns True if any meta object exists in the library or False otherwise.

Returns True if any meta object (class or function) exists in the library or False otherwise

Return type bool[787]

get_metas()
Returns a sequence of the meta (class and functions) that belong to this library.

Returns a sequence of meta (class and functions) that belong to this library

Return type seq<:class:~'sardana.sardanameta.SardanaCode'>

get_name()
Returns the module name for this library (same as :meth:~sardana.sardanameta.SardanaLibrary.get_module_nan

Returns the module name

Return type str[788]

get_module_name()
Returns the module name for this library (same as :meth:~sardana.sardanameta.SardanaLibrary.get_name).

Returns the module name

Return type str[789]

get_module()
Returns the python module for this library.

Returns the python module

Return type object[790]

get_description()
Returns the this library documentation or "<Undocumented>" if no documentation exists.

Returns this library documentation or None

Return type str[791]

get_code()
Returns a sequence of sourcelines corresponding to the module code.

Returns list of source code lines

Return type list<str>

get_file_path()
Returns the file path for this library. On posix systems is something like: /abs/path/filename.py

Returns this library file path

[785] https://docs.python.org/dev/library/stdtypes.html#str
[786] https://docs.python.org/dev/library/functions.html#bool
[787] https://docs.python.org/dev/library/functions.html#bool
[788] https://docs.python.org/dev/library/stdtypes.html#str
[789] https://docs.python.org/dev/library/stdtypes.html#str
[790] https://docs.python.org/dev/library/functions.html#object
[791] https://docs.python.org/dev/library/stdtypes.html#str

Return type str[792]

get_file_name()
Returns the file name for this library. On posix systems is something like: filename.py

Returns this library file name

Return type str[793]

has_errors()
Returns True if this library has syntax errors or False otherwise.

Returns True if this library has syntax errors or False otherwise

Return type bool[794]

set_error(*exc_info*)
Sets the error information for this library

Parameters **exc_info** (*tuple<type, value, traceback>*) – error information. It must be an object similar to the one returned by `sys.exc_info()`[795]

get_error()
Gets the error information for this library or None if no error exists

Returns error information. An object similar to the one returned by `sys.exc_info()`[796]

Return type tuple<type, value, traceback>

serialize(**args*, ***kwargs*)
Returns a serializable object describing this object.

Returns a serializable dict

Return type dict[797]

Critical = 50

Debug = 10

DftLogLevel = 20

DftLogMessageFormat = '%(threadName)-14s %(levelname)-8s %(asctime)s %(name)s: %(mess

Error = 40

Fatal = 50

Info = 20

Trace = 5

Warning = 30

add_listener(*listener*)
Adds a new listener for this object.

Parameters **listener** – a listener

are_events_blocked()

block_events()

fire_event(*event_type*, *event_value*, *listeners=None*, *protected=True*)

[792] https://docs.python.org/dev/library/stdtypes.html#str
[793] https://docs.python.org/dev/library/stdtypes.html#str
[794] https://docs.python.org/dev/library/functions.html#bool
[795] https://docs.python.org/dev/library/sys.html#sys.exc_info
[796] https://docs.python.org/dev/library/sys.html#sys.exc_info
[797] https://docs.python.org/dev/library/stdtypes.html#dict

flush_queue()

frontend
the object frontend

full_name
object full name

get_frontend()
Returns this sardana frontend object or None if no frontend is registered

Returns this objects frontend

Return type `object`[798]

get_full_name()
Returns this sardana object full name

Returns this sardana object full name

Return type str[799]

get_interface()
Returns the interface this object implements.

Returns The interface this object implements.

Return type *`sardana.sardanadefs.Interface`*

get_interface_names()
Returns a sequence of interface names this object implements.

Returns The sequence of interfaces this object implements.

Return type sequence<*`str`*>

get_interfaces()
Returns the set of interfaces this object implements.

Returns The set of interfaces this object implements.

Return type class:*set* <*`sardana.sardanadefs.Interface`*>

get_manager()
Return the `sardana.Manager` which *owns* this sardana object.

Returns the manager which *owns* this pool object.

Return type `sardana.Manager`

get_parent()
Returns this pool object parent.

Returns this objects parent

Return type *`SardanaBaseObject`*

get_parent_name()
Returns this sardana object parent's name.

Returns this objects parent

Return type str[800]

get_type()
Returns this sardana object type.

Returns this sardana object type

Return type *`ElementType`*

[798] https://docs.python.org/dev/library/functions.html#object
[799] https://docs.python.org/dev/library/stdtypes.html#str
[800] https://docs.python.org/dev/library/stdtypes.html#str

`has_listeners`()
  Returns True if anybody is listening to events from this object
    Returns True is at least one listener is listening or False otherwise

`log_level = 20`

`manager`
  reference to the `sardana.Manager`

`name`
  object name

`queue_event`(*event_type*, *event_value*, *listeners=None*)

`remove_listener`(*listener*)
  Removes an existing listener for this object.
    Parameters **`listener`** – the listener to be removed

    Returns True is succeeded or False otherwise

`root_inited = True`

`serialized`(**args*, ***kwargs*)

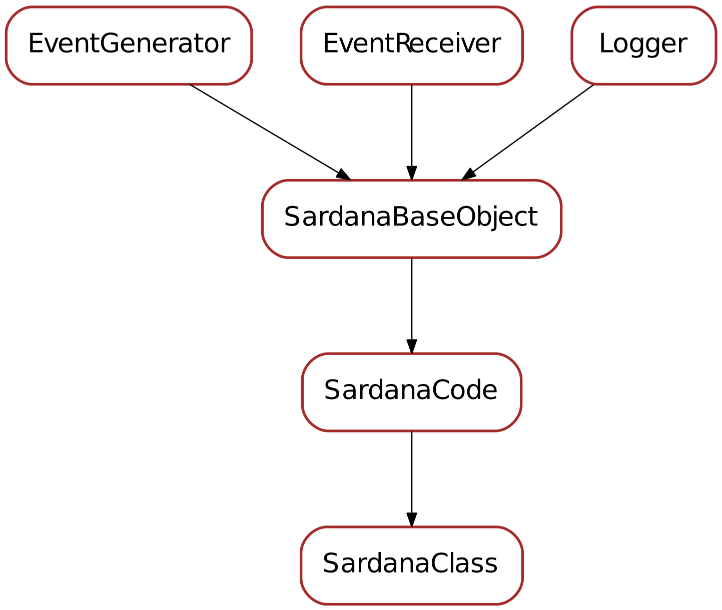
`set_name`(*name*)
  Sets sardana object name
    Param sardana object name

    Type str

`str`(**args*, ***kwargs*)

`unblock_events`()

SardanaClass



class SardanaClass(***kwargs*)

Object representing a python class.

Critical = 50

Debug = 10

DftLogLevel = 20

DftLogMessageFormat = '%(threadName)-14s %(levelname)-8s %(asctime)s %(name)s: %(mess

Error = 40

Fatal = 50

Info = 20

Trace = 5

Warning = 30

add_listener(*listener*)

Adds a new listener for this object.

Parameters **listener** – a listener

are_events_blocked()

block_events()

code

Returns a tuple (sourcelines, firstline) corresponding to the definition of this code object. sourcelines is a list of source code lines. firstline is the line number of the first source code line.

code_object

description = '<Undocumented>'

file_name

Returns the file name for the library where this class is. On posix systems is something like: filename.py

Returns the file name for the library where this class is

Return type str[801]

file_path

Returns the file path for for the library where this class is. On posix systems is something like: /abs/path/filename.py

Returns the file path for for the library where this class is

Return type str[802]

fire_event(*event_type*, *event_value*, *listeners=None*, *protected=True*)

flush_queue()

frontend

the object frontend

full_name

object full name

get_brief_description(*max_chars=60*)

get_code()

Returns a tuple (sourcelines, firstline) corresponding to the definition of the controller class. sourcelines is a list of source code lines. firstline is the line number of the first source code line.

get_frontend()

Returns this sardana frontend object or None if no frontend is registered

Returns this objects frontend

Return type `object`[803]

get_full_name()

Returns this sardana object full name

Returns this sardana object full name

Return type str[804]

get_interface()

Returns the interface this object implements.

Returns The interface this object implements.

Return type *`sardana.sardanadefs.Interface`*

get_interface_names()

Returns a sequence of interface names this object implements.

Returns The sequence of interfaces this object implements.

Return type sequence<*`str`*>

get_interfaces()

Returns the set of interfaces this object implements.

801 https://docs.python.org/dev/library/stdtypes.html#str
802 https://docs.python.org/dev/library/stdtypes.html#str
803 https://docs.python.org/dev/library/functions.html#object
804 https://docs.python.org/dev/library/stdtypes.html#str

Returns The set of interfaces this object implements.

Return type class:*set* <*sardana.sardanadefs.Interface*>

get_manager()
Return the `sardana.Manager` which *owns* this sardana object.

Returns the manager which *owns* this pool object.

Return type `sardana.Manager`

get_name()
Returns this sardana object name

Returns this sardana object name

Return type str[805]

get_parent()
Returns this pool object parent.

Returns this objects parent

Return type *SardanaBaseObject*

get_parent_name()
Returns this sardana object parent's name.

Returns this objects parent

Return type str[806]

get_type()
Returns this sardana object type.

Returns this sardana object type

Return type *ElementType*

has_listeners()
Returns True if anybody is listening to events from this object

Returns True is at least one listener is listening or False otherwise

lib
Returns the library :class:~'sardana.sardanameta.SardanaLibrary' for this class.

Returns a reference to the library where this class is located

Return type :class:~'sardana.sardanameta.SardanaLibrary'

log_level = 20

manager
reference to the `sardana.Manager`

module
Returns the python module for this class.

Returns the python module

Return type object[807]

module_name
Returns the module name for this class.

Returns the module name

Return type str[808]

[805] https://docs.python.org/dev/library/stdtypes.html#str
[806] https://docs.python.org/dev/library/stdtypes.html#str
[807] https://docs.python.org/dev/library/functions.html#object
[808] https://docs.python.org/dev/library/stdtypes.html#str

name
object name

path
Returns the absolute path for the library where this class is. On posix systems is something like: /abs/path

Returns the absolute path for the library where this class is

Return type str[809]

queue_event(*event_type*, *event_value*, *listeners=None*)

remove_listener(*listener*)
Removes an existing listener for this object.

Parameters **listener** – the listener to be removed

Returns True is succeeded or False otherwise

root_inited = True

serialize(**args*, ***kwargs*)
Returns a serializable object describing this object.

Returns a serializable dict

Return type dict[810]

serialized(**args*, ***kwargs*)

set_name(*name*)
Sets sardana object name

Param sardana object name

Type str

str(**args*, ***kwargs*)

unblock_events()

klass

sardanamanager

This module is part of the Python Sardana libray. It defines the base class for Sardana manager

Classes

- *SardanaElementManager*

[809] https://docs.python.org/dev/library/stdtypes.html#str
[810] https://docs.python.org/dev/library/stdtypes.html#dict

SardanaElementManager

class SardanaElementManager
A class capable of manage elements

SerializationProtocol = 'json'

get_serialization_protocol()

set_serialization_protocol(*protocol*)

serialization_protocol
the serialization protocol

serialize_element(*element*, **args*, ***kwargs*)

serialize_object(*obj*, **args*, ***kwargs*)

str_element(*element*, **args*, ***kwargs*)

str_object(*obj*, **args*, ***kwargs*)

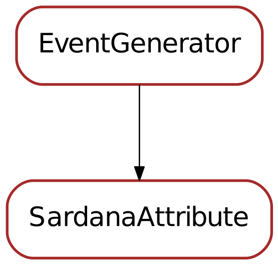
sardanaattribute

This module is part of the Python Sardana libray. It defines the base classes for Sardana attributes

Classes

- *SardanaAttribute*
- *SardanaSoftwareAttribute*
- *ScalarNumberAttribute*
- *SardanaAttributeConfiguration*

SardanaAttribute



class SardanaAttribute(*obj, name=None, initial_value=None, **kwargs*)

Class representing an atomic attribute like position of a motor or a counter value

has_value()

Determines if the attribute's read value has been read at least once in the lifetime of the attribute.

Returns True if the attribute has a read value stored or False otherwise

Return type bool[811]

has_write_value()

Determines if the attribute's write value has been read at least once in the lifetime of the attribute.

Returns True if the attribute has a write value stored or False otherwise

Return type bool[812]

get_obj()

Returns the object which *owns* this attribute

Returns the object which *owns* this attribute

Return type obj

in_error()

Determines if this attribute is in error state.

Returns True if the attribute is in error state or False otherwise

Return type bool[813]

set_value(*value, exc_info=None, timestamp=None, propagate=1*)

Sets the current read value and propagates the event (if propagate > 0).

Parameters

- **value** (`obj or SardanaValue`) – the new read value for this attribute. If a SardanaValue is given, exc_info and timestamp are ignored (if given)
- **exc_info** (`tuple<3> or None`[814]) – exception information as returned by `sys.exc_info()`[815] [default: None, meaning no exception]

[811] https://docs.python.org/dev/library/functions.html#bool
[812] https://docs.python.org/dev/library/functions.html#bool
[813] https://docs.python.org/dev/library/functions.html#bool
[814] https://docs.python.org/dev/library/constants.html#None
[815] https://docs.python.org/dev/library/sys.html#sys.exc_info

- **timestamp** (*float*[816] *or None*[817]) – timestamp of attribute readout [default: None, meaning create a 'now' timestamp]
- **propagate** (*int*[818]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

get_value()

Returns the last read value for this attribute.

Returns the last read value for this attribute

Return type obj

Raises Exception[819] if no read value has been set yet

get_value_obj()

Returns the last read value for this attribute.

Returns the last read value for this attribute

Return type *SardanaValue*

Raises Exception[820] if no read value has been set yet

set_write_value(*w_value*, *timestamp=None*, *propagate=1*)

Sets the current write value.

Parameters

- **value** (*obj or SardanaValue*) – the new write value for this attribute. If a SardanaValue is given, timestamp is ignored (if given)
- **timestamp** (*float*[821] *or None*[822]) – timestamp of attribute write [default: None, meaning create a 'now' timestamp]
- **propagate** (*int*[823]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

get_write_value()

Returns the last write value for this attribute.

Returns the last write value for this attribute or None if value has not been written yet

Return type obj

get_write_value_obj()

Returns the last write value object for this attribute.

Returns the last write value for this attribute or None if value has not been written yet

Return type *SardanaValue*

get_exc_info()

Returns the exception information (like `sys.exc_info()`[824]) about last attribute readout or None if last read did not generate an exception.

Returns exception information or None

[816] https://docs.python.org/dev/library/functions.html#float
[817] https://docs.python.org/dev/library/constants.html#None
[818] https://docs.python.org/dev/library/functions.html#int
[819] https://docs.python.org/dev/library/exceptions.html#Exception
[820] https://docs.python.org/dev/library/exceptions.html#Exception
[821] https://docs.python.org/dev/library/functions.html#float
[822] https://docs.python.org/dev/library/constants.html#None
[823] https://docs.python.org/dev/library/functions.html#int
[824] https://docs.python.org/dev/library/sys.html#sys.exc_info

Return type tuple<3> or None

accepts(*propagate*)

get_timestamp()
Returns the timestamp of the last readout or None if the attribute has never been read before

Returns timestamp of the last readout or None

Return type float[825] or None

get_write_timestamp()
Returns the timestamp of the last write or None if the attribute has never been written before

Returns timestamp of the last write or None

Return type float[826] or None

fire_write_event(*propagate=1*)
Fires an event to the listeners of the object which owns this attribute.

Parameters **propagate** (*int*[827]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

fire_read_event(*propagate=1*)
Fires an event to the listeners of the object which owns this attribute.

Parameters **propagate** (*int*[828]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

obj
Returns the object which *owns* this attribute

Returns the object which *owns* this attribute

Return type obj

value_obj
Returns the last read value for this attribute.

Returns the last read value for this attribute

Return type *SardanaValue*

Raises Exception[829] if no read value has been set yet

write_value_obj
Returns the last write value object for this attribute.

Returns the last write value for this attribute or None if value has not been written yet

Return type *SardanaValue*

value
Returns the last read value for this attribute.

Returns the last read value for this attribute

Return type obj

Raises Exception[830] if no read value has been set yet

w_value
Returns the last write value for this attribute.

[825] https://docs.python.org/dev/library/functions.html#float
[826] https://docs.python.org/dev/library/functions.html#float
[827] https://docs.python.org/dev/library/functions.html#int
[828] https://docs.python.org/dev/library/functions.html#int
[829] https://docs.python.org/dev/library/exceptions.html#Exception
[830] https://docs.python.org/dev/library/exceptions.html#Exception

Returns the last write value for this attribute or None if value has not been written yet

Return type obj

timestamp
the read timestamp

w_timestamp
the write timestamp

error
Determines if this attribute is in error state.

Returns True if the attribute is in error state or False otherwise

Return type bool[831]

exc_info
Returns the exception information (like `sys.exc_info()`[832]) about last attribute readout or None if last read did not generate an exception.

Returns exception information or None

Return type tuple<3> or None

add_listener(*listener*)
Adds a new listener for this object.

Parameters **listener** – a listener

fire_event(*event_type*, *event_value*, *listeners=None*)

flush_queue()

has_listeners()
Returns True if anybody is listening to events from this object

Returns True is at least one listener is listening or False otherwise

queue_event(*event_type*, *event_value*, *listeners=None*)

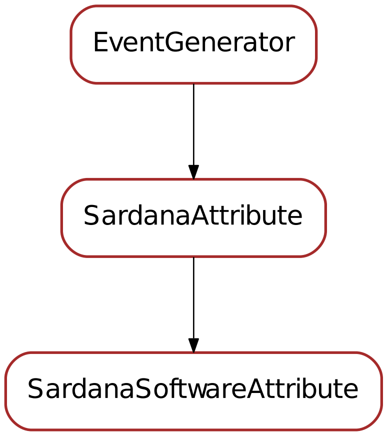
remove_listener(*listener*)
Removes an existing listener for this object.

Parameters **listener** – the listener to be removed

Returns True is succeeded or False otherwise

[831] https://docs.python.org/dev/library/functions.html#bool
[832] https://docs.python.org/dev/library/sys.html#sys.exc_info

SardanaSoftwareAttribute



class SardanaSoftwareAttribute(*obj*, *name=None*, *initial_value=None*, ***kwargs*)

Class representing a software attribute. The difference between this and `SardanaAttribute` is that, because it is a pure software attribute, there is no difference ever between the read and write values.

get_value()

Returns the last read value for this attribute.

Returns the last read value for this attribute

Return type obj

Raises `Exception`[833] if no read value has been set yet

set_value(*value*, *exc_info=None*, *timestamp=None*, *propagate=1*)

Sets the current read value and propagates the event (if propagate > 0).

Parameters

- **value** (*obj*) – the new read value for this attribute
- **exc_info** (*tuple<3> or None*[834]) – exception information as returned by `sys.exc_info()`[835] [default: None, meaning no exception]
- **timestamp** (*float*[836] *or None*[837]) – timestamp of attribute readout [default: None, meaning create a 'now' timestamp]
- **propagate** (*int*[838]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

value

Returns the last read value for this attribute.

Returns the last read value for this attribute

[833] https://docs.python.org/dev/library/exceptions.html#Exception
[834] https://docs.python.org/dev/library/constants.html#None
[835] https://docs.python.org/dev/library/sys.html#sys.exc_info
[836] https://docs.python.org/dev/library/functions.html#float
[837] https://docs.python.org/dev/library/constants.html#None
[838] https://docs.python.org/dev/library/functions.html#int

Return type obj

Raises `Exception`[839] if no read value has been set yet

accepts(*propagate*)

add_listener(*listener*)
Adds a new listener for this object.

Parameters **listener** – a listener

error
Determines if this attribute is in error state.

Returns True if the attribute is in error state or False otherwise

Return type bool[840]

exc_info
Returns the exception information (like `sys.exc_info()`[841]) about last attribute readout or None if last read did not generate an exception.

Returns exception information or None

Return type tuple<3> or None

fire_event(*event_type*, *event_value*, *listeners=None*)

fire_read_event(*propagate=1*)
Fires an event to the listeners of the object which owns this attribute.

Parameters **propagate** (*int*[842]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

fire_write_event(*propagate=1*)
Fires an event to the listeners of the object which owns this attribute.

Parameters **propagate** (*int*[843]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

flush_queue()

get_exc_info()
Returns the exception information (like `sys.exc_info()`[844]) about last attribute readout or None if last read did not generate an exception.

Returns exception information or None

Return type tuple<3> or None

get_obj()
Returns the object which *owns* this attribute

Returns the object which *owns* this attribute

Return type obj

get_timestamp()
Returns the timestamp of the last readout or None if the attribute has never been read before

Returns timestamp of the last readout or None

Return type float[845] or None

[839] https://docs.python.org/dev/library/exceptions.html#Exception
[840] https://docs.python.org/dev/library/functions.html#bool
[841] https://docs.python.org/dev/library/sys.html#sys.exc_info
[842] https://docs.python.org/dev/library/functions.html#int
[843] https://docs.python.org/dev/library/functions.html#int
[844] https://docs.python.org/dev/library/sys.html#sys.exc_info
[845] https://docs.python.org/dev/library/functions.html#float

get_value_obj()
Returns the last read value for this attribute.

Returns the last read value for this attribute

Return type *SardanaValue*

Raises `Exception`[846] if no read value has been set yet

get_write_timestamp()
Returns the timestamp of the last write or None if the attribute has never been written before

Returns timestamp of the last write or None

Return type float[847] or None

get_write_value()
Returns the last write value for this attribute.

Returns the last write value for this attribute or None if value has not been written yet

Return type obj

get_write_value_obj()
Returns the last write value object for this attribute.

Returns the last write value for this attribute or None if value has not been written yet

Return type *SardanaValue*

has_listeners()
Returns True if anybody is listening to events from this object

Returns True is at least one listener is listening or False otherwise

has_value()
Determines if the attribute's read value has been read at least once in the lifetime of the attribute.

Returns True if the attribute has a read value stored or False otherwise

Return type bool[848]

has_write_value()
Determines if the attribute's write value has been read at least once in the lifetime of the attribute.

Returns True if the attribute has a write value stored or False otherwise

Return type bool[849]

in_error()
Determines if this attribute is in error state.

Returns True if the attribute is in error state or False otherwise

Return type bool[850]

obj
Returns the object which *owns* this attribute

Returns the object which *owns* this attribute

Return type obj

queue_event(*event_type*, *event_value*, *listeners=None*)

846 https://docs.python.org/dev/library/exceptions.html#Exception
847 https://docs.python.org/dev/library/functions.html#float
848 https://docs.python.org/dev/library/functions.html#bool
849 https://docs.python.org/dev/library/functions.html#bool
850 https://docs.python.org/dev/library/functions.html#bool

`remove_listener`(*listener*)

Removes an existing listener for this object.

Parameters **`listener`** – the listener to be removed

Returns True is succeeded or False otherwise

`set_write_value`(*w_value*, *timestamp=None*, *propagate=1*)

Sets the current write value.

Parameters

- **`value`** (*`obj or SardanaValue`*) – the new write value for this attribute. If a SardanaValue is given, timestamp is ignored (if given)
- **`timestamp`** (*`float`*[851] *`or None`*[852]) – timestamp of attribute write [default: None, meaning create a 'now' timestamp]
- **`propagate`** (*`int`*[853]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

`timestamp`

the read timestamp

`value_obj`

Returns the last read value for this attribute.

Returns the last read value for this attribute

Return type *`SardanaValue`*

Raises `Exception`[854] if no read value has been set yet

`w_timestamp`

the write timestamp

`w_value`

Returns the last write value for this attribute.

Returns the last write value for this attribute or None if value has not been written yet

Return type obj

`write_value_obj`

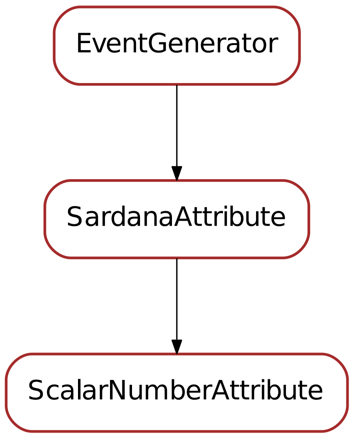
Returns the last write value object for this attribute.

Returns the last write value for this attribute or None if value has not been written yet

Return type *`SardanaValue`*

[851] https://docs.python.org/dev/library/functions.html#float
[852] https://docs.python.org/dev/library/constants.html#None
[853] https://docs.python.org/dev/library/functions.html#int
[854] https://docs.python.org/dev/library/exceptions.html#Exception

ScalarNumberAttribute



class ScalarNumberAttribute(**args, **kwargs*)

A `SardanaAttribute` specialized for numbers

accepts(*propagate*)

add_listener(*listener*)

Adds a new listener for this object.

Parameters listener – a listener

error

Determines if this attribute is in error state.

Returns True if the attribute is in error state or False otherwise

Return type bool[855]

exc_info

Returns the exception information (like `sys.exc_info()`[856]) about last attribute readout or None if last read did not generate an exception.

Returns exception information or None

Return type tuple<3> or None

fire_event(*event_type, event_value, listeners=None*)

fire_read_event(*propagate=1*)

Fires an event to the listeners of the object which owns this attribute.

Parameters propagate (*int*[857]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

fire_write_event(*propagate=1*)

Fires an event to the listeners of the object which owns this attribute.

[855] https://docs.python.org/dev/library/functions.html#bool
[856] https://docs.python.org/dev/library/sys.html#sys.exc_info
[857] https://docs.python.org/dev/library/functions.html#int

Parameters **propagate** (*int*[858]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

flush_queue()

get_exc_info()
Returns the exception information (like `sys.exc_info()`[859]) about last attribute readout or None if last read did not generate an exception.

Returns exception information or None

Return type tuple<3> or None

get_obj()
Returns the object which *owns* this attribute

Returns the object which *owns* this attribute

Return type obj

get_timestamp()
Returns the timestamp of the last readout or None if the attribute has never been read before

Returns timestamp of the last readout or None

Return type float[860] or None

get_value()
Returns the last read value for this attribute.

Returns the last read value for this attribute

Return type obj

Raises `Exception`[861] if no read value has been set yet

get_value_obj()
Returns the last read value for this attribute.

Returns the last read value for this attribute

Return type *SardanaValue*

Raises `Exception`[862] if no read value has been set yet

get_write_timestamp()
Returns the timestamp of the last write or None if the attribute has never been written before

Returns timestamp of the last write or None

Return type float[863] or None

get_write_value()
Returns the last write value for this attribute.

Returns the last write value for this attribute or None if value has not been written yet

Return type obj

get_write_value_obj()
Returns the last write value object for this attribute.

Returns the last write value for this attribute or None if value has not been written yet

[858] https://docs.python.org/dev/library/functions.html#int
[859] https://docs.python.org/dev/library/sys.html#sys.exc_info
[860] https://docs.python.org/dev/library/functions.html#float
[861] https://docs.python.org/dev/library/exceptions.html#Exception
[862] https://docs.python.org/dev/library/exceptions.html#Exception
[863] https://docs.python.org/dev/library/functions.html#float

Return type `SardanaValue`

has_listeners()
Returns True if anybody is listening to events from this object

Returns True is at least one listener is listening or False otherwise

has_value()
Determines if the attribute's read value has been read at least once in the lifetime of the attribute.

Returns True if the attribute has a read value stored or False otherwise

Return type bool[864]

has_write_value()
Determines if the attribute's write value has been read at least once in the lifetime of the attribute.

Returns True if the attribute has a write value stored or False otherwise

Return type bool[865]

in_error()
Determines if this attribute is in error state.

Returns True if the attribute is in error state or False otherwise

Return type bool[866]

obj
Returns the object which *owns* this attribute

Returns the object which *owns* this attribute

Return type obj

queue_event(*event_type*, *event_value*, *listeners=None*)

remove_listener(*listener*)
Removes an existing listener for this object.

Parameters **listener** – the listener to be removed

Returns True is succeeded or False otherwise

set_value(*value*, *exc_info=None*, *timestamp=None*, *propagate=1*)
Sets the current read value and propagates the event (if propagate > 0).

Parameters

- **value** (*obj or* `SardanaValue`) – the new read value for this attribute. If a SardanaValue is given, exc_info and timestamp are ignored (if given)
- **exc_info** (*tuple<3> or None*[867]) – exception information as returned by `sys.exc_info()`[868] [default: None, meaning no exception]
- **timestamp** (*float*[869] *or None*[870]) – timestamp of attribute readout [default: None, meaning create a 'now' timestamp]
- **propagate** (*int*[871]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

[864] https://docs.python.org/dev/library/functions.html#bool
[865] https://docs.python.org/dev/library/functions.html#bool
[866] https://docs.python.org/dev/library/functions.html#bool
[867] https://docs.python.org/dev/library/constants.html#None
[868] https://docs.python.org/dev/library/sys.html#sys.exc_info
[869] https://docs.python.org/dev/library/functions.html#float
[870] https://docs.python.org/dev/library/constants.html#None
[871] https://docs.python.org/dev/library/functions.html#int

set_write_value(*w_value*, *timestamp=None*, *propagate=1*)
Sets the current write value.

Parameters

- **value** (*obj or SardanaValue*) – the new write value for this attribute. If a SardanaValue is given, timestamp is ignored (if given)
- **timestamp** (*float*[872] *or None*[873]) – timestamp of attribute write [default: None, meaning create a 'now' timestamp]
- **propagate** (*int*[874]) – 0 for not propagating, 1 to propagate, 2 propagate with priority

timestamp
the read timestamp

value
Returns the last read value for this attribute.

Returns the last read value for this attribute

Return type obj

Raises Exception[875] if no read value has been set yet

value_obj
Returns the last read value for this attribute.

Returns the last read value for this attribute

Return type *SardanaValue*

Raises Exception[876] if no read value has been set yet

w_timestamp
the write timestamp

w_value
Returns the last write value for this attribute.

Returns the last write value for this attribute or None if value has not been written yet

Return type obj

write_value_obj
Returns the last write value object for this attribute.

Returns the last write value for this attribute or None if value has not been written yet

Return type *SardanaValue*

[872] https://docs.python.org/dev/library/functions.html#float
[873] https://docs.python.org/dev/library/constants.html#None
[874] https://docs.python.org/dev/library/functions.html#int
[875] https://docs.python.org/dev/library/exceptions.html#Exception
[876] https://docs.python.org/dev/library/exceptions.html#Exception

SardanaAttributeConfiguration

class SardanaAttributeConfiguration

Storage class for *SardanaAttribute* information (like ranges)

NoRange = (-inf, inf)

sardanavalue

This module is part of the Python Sardana libray. It defines the base classes for Sardana values

Classes

- *SardanaValue*

SardanaValue

class SardanaValue(*value=None, exc_info=None, timestamp=None, dtype=None, dformat=None*)

Sardana test API

Macro test API

Classes

- *BaseMacroExecutor*
- *MacroExecutorFactory*
- *BaseMacroTestCase*
- *RunMacroTestCase*

- *RunStopMacroTestCase*
- *SarDemoEnv*

Decorator

@macroTest

macroTest(*klass=None*, *helper_name=None*, *test_method_name=None*, *test_method_doc=None*, ***helper_kwargs*)

This decorator is an specialization of :function::*taurus.test.insertTest* for macro testing. It inserts test methods from a helper method that may accept arguments.

macroTest provides a very economic API for creating new tests for a given macro based on a helper method.

macroTest accepts the following arguments:

- **helper_name (str): the name of the helper method. macroTest will** insert a test method which calls the helper with any the helper_kwargs (see below).
- **test_method_name (str): Optional. Name of the test method to be used.** If None given, one will be generated from the macro and helper names.
- **test_method_doc (str): The docstring for the inserted test method** (this shows in the unit test output). If None given, a default one is generated which includes the input parameters and the helper name.
- ****helper_kwargs: All remaining keyword arguments are passed to the** helper.

macroTest can work with the *macro_name* class member

This decorator can be considered a "base" decorator. It is often used to create other decorators in which the helper method is pre-set. Some of them are already provided in this module:

- `testRun()` is equivalent to macroTest with helper_name='macro_runs'
- `testStop()` is equivalent to macroTest with helper_name='macro_stops'
- `testFail()` is equivalent to macroTest with helper_name='macro_fails'

The advantage of using the decorators compared to writing the test methods directly is that the helper method can get keyword arguments and therefore avoid duplication of code for very similar tests (think, e.g. on writing similar tests for various sets of macro input parameters):

Consider the following code written using the *RunMacroTestCase.macro_runs()* helper:

```
class FooTest(RunMacroTestCase, unittest.TestCase)
    macro_name = twice

    def test_foo_runs_with_input_2(self):
        '''test that twice(2) runs'''
        self.macro_runs(macro_params=['2'])

    def test_foo_runs_with_input_minus_1(self):
        '''test that twice(2) runs'''
        self.macro_runs(macro_params=['-1'])
```

The equivalent code could be written as:

```
@macroTest(helper_name='macro_runs', macro_params=['2'])
@macroTest(helper_name='macro_runs', macro_params=['-1'])
class FooTest(RunMacroTestCase, unittest.TestCase):
    macro_name = 'twice'
```

Or, even better, using the specialized testRun decorator:

```
@testRun(macro_params=['2'])
@testRun(macro_params=['-1'])
class FooTest(RunMacroTestCase, unittest.TestCase):
    macro_name = 'twice'
```

See also:

:function::*taurus.test.insertTest*

BaseMacroExecutor

class BaseMacroExecutor

Abstract MacroExecutor class. Inherit from it if you want to create your own macro executor.

log_levels = ['debug', 'output', 'info', 'warning', 'critical', 'error']

run (*macro_name, macro_params=None, sync=True, timeout=inf*)

Execute macro.

Parameters

- **macro_name** – (string) name of macro to be executed
- **macro_params** – (list<string>) macro parameters (default is macro_params=None for macros without parameters or with the default values)
- **sync** – (bool) whether synchronous or asynchronous call (default is sync=True)
- **timeout** –

  (float) timeout (in s) that will be passed to the wait method, in case of synchronous execution

  In asyncrhonous execution method *wait()* has to be explicitly called.

wait (*timeout=inf*)

Wait until macro is done. Use it in asynchronous executions.

Parameters timeout – (float) waiting timeout (in s)

stop (*started_event_timeout=3.0*)

Stop macro execution. Execute macro in synchronous way before using this method.

Parameters started_event_timeout – (float) waiting timeout for started event

registerLog (*log_level*)

Start registering log messages.

Parameters log_level – (str) string indicating the log level

unregisterLog(*log_level*)
Stop registering log messages.
Parameters **log_level** – (str) string indicating the log level

getLog(*log_level*)
Get log messages.
Parameters **log_level** – (str) string indicating the log level

Returns (seq<str>) list of strings with log messages

registerAll()
Register for macro result, all log levels and common buffer.

unregisterAll()
Unregister macro result, all log levels and common buffer.

registerResult()
Register for macro result

unregisterResult()
Unregister macro result.

getResult()
Get macro result.
Returns (seq<str>) list of strings with Result messages

createCommonBuffer()
Create a common buffer, where all the registered logs will be stored.

getCommonBuffer()
Get common buffer. Method getCommonBuffer can only be used if at least one buffer exists.

Returns

(seq<str>) list of strings with messages from all log levels

See also:

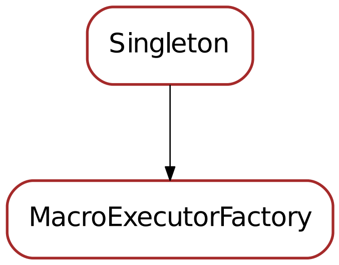
createCommonBuffer()

getState()
Get macro execution state.
Returns (str)

getStateBuffer()
Get buffer (history) of macro execution states.
Returns (seq<str>)

getExceptionStr()
Get macro exception type representation (None if the macro state is not exception).
Returns (str)

MacroExecutorFactory



class MacroExecutorFactory(**a, **kw*)

A scheme-agnostic factory for MacroExecutor instances

Example:

```
f =  MacroExecutorFactory()
f.getMacroExecutor('tango://my/door/name') #returns a TangoMacroExecutor
```

Note: For the moment, only TangoMacroExecutor is supported

getMacroExecutor(*door_name=None*)

Returns a macro executor instance (a subclass of `BaseMacroExecutor`) depending on the door being used.

BaseMacroTestCase

class BaseMacroTestCase

An abstract class for macro testing. BaseMacroTestCase will provide a *macro_executor* member which is an instance of BaseMacroExecutor and which can be used to run a macro.

To use it, simply inherit from BaseMacroTestCase *and* unittest.TestCase and provide the following class members:

- macro_name (string) name of the macro to be tested
- **door_name (string) name of the door where the macro will be executed.** This is optional. If not set, *sardanacustomsettings.UNITTEST_DOOR_NAME* is used

Then you may define test methods.

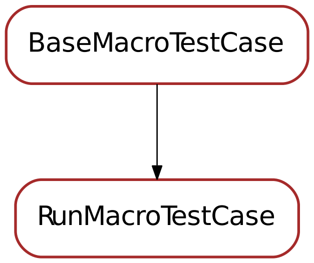
macro_name = None

door_name = **'door/demo1/1'**

setUp()
A macro_executor instance must be created

tearDown()
The macro_executor instance must be removed

RunMacroTestCase



class RunMacroTestCase
A base class for testing execution of arbitrary Sardana macros. See `BaseMacroTestCase` for requirements.

It provides the following helper methods:

- `macro_runs()`
- `macro_fails()`

assertFinished(*msg*)
Asserts that macro has finished.

setUp()
Preconditions: - Those from `BaseMacroTestCase` - the macro executor registers to all the log levels

macro_runs(*macro_name=None, macro_params=None, wait_timeout=inf, data=0*)
A helper method to create tests that check if the macro can be successfully executed for the given input parameters. It may also optionally perform checks on the outputs from the execution.

Parameters

- **macro_name** – (str) macro name (takes precedence over macro_name class member)
- **macro_params** – (seq<str>): parameters for running the macro. If passed, they must be given as a sequence of their string representations.
- **wait_timeout** – (float) maximum allowed time (in s) for the macro to finish. By default infinite timeout is used.
- **data** – (obj) Optional. If passed, the macro data after the execution is tested to be equal to this.

macro_fails(*macro_name=None, macro_params=None, wait_timeout=inf, exception=None*)
Check that the macro fails to run for the given input parameters

Parameters

- **macro_name** – (str) macro name (takes precedence over macro_name class member)
- **macro_params** – (seq<str>) input parameters for the macro
- **wait_timeout** – maximum allowed time for the macro to fail. By default infinite timeout is used.
- **exception** – (str or Exception) if given, an additional check of the type of the exception is done. (IMPORTANT: this is just a comparison of str representations of exception objects)

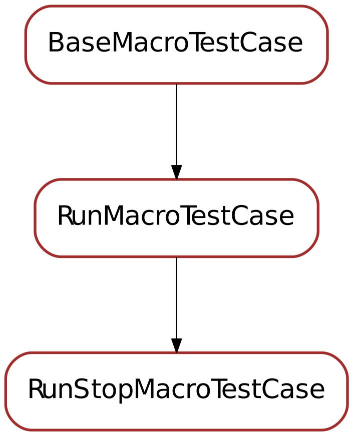
door_name = 'door/demo1/1'

macro_name = None

tearDown()

The macro_executor instance must be removed

RunStopMacroTestCase



class RunStopMacroTestCase

This is an extension of *RunMacroTestCase* to include helpers for testing the abort process of a macro. Useful for Runnable and Stopable macros.

It provides the *macro_stops()* helper

assertStopped(*msg*)

Asserts that macro was stopped

macro_stops(*macro_name=None*, *macro_params=None*, *stop_delay=0.1*, *wait_timeout=inf*)

A helper method to create tests that check if the macro can be successfully stoped (a.k.a. aborted) after it has been launched.

Parameters

- **macro_name** – (str) macro name (takes precedence over macro_name class member)
- **macro_params** – (seq<str>): parameters for running the macro. If passed, they must be given as a sequence of their string representations.
- **stop_delay** – (float) Time (in s) to wait between launching the macro and sending the stop command. default=0.1
- **wait_timeout** – (float) maximum allowed time (in s) for the macro to finish. By default infinite timeout is used.

assertFinished(*msg*)
Asserts that macro has finished.

door_name = 'door/demo1/1'

macro_fails(*macro_name=None*, *macro_params=None*, *wait_timeout=inf*, *exception=None*)
Check that the macro fails to run for the given input parameters

Parameters

- **macro_name** – (str) macro name (takes precedence over macro_name class member)
- **macro_params** – (seq<str>) input parameters for the macro
- **wait_timeout** – maximum allowed time for the macro to fail. By default infinite timeout is used.
- **exception** – (str or Exception) if given, an additional check of the type of the exception is done. (IMPORTANT: this is just a comparison of str representations of exception objects)

macro_name = None

macro_runs(*macro_name=None*, *macro_params=None*, *wait_timeout=inf*, *data=0*)
A helper method to create tests that check if the macro can be successfully executed for the given input parameters. It may also optionally perform checks on the outputs from the execution.

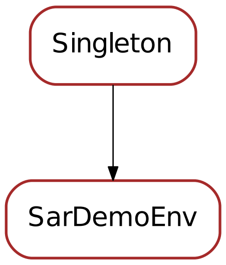
Parameters

- **macro_name** – (str) macro name (takes precedence over macro_name class member)
- **macro_params** – (seq<str>): parameters for running the macro. If passed, they must be given as a sequence of their string representations.
- **wait_timeout** – (float) maximum allowed time (in s) for the macro to finish. By default infinite timeout is used.
- **data** – (obj) Optional. If passed, the macro data after the execution is tested to be equal to this.

setUp()
Preconditions: - Those from `BaseMacroTestCase` - the macro executor registers to all the log levels

tearDown()
The macro_executor instance must be removed

SarDemoEnv



class SarDemoEnv(**a, **kw*)

Class to get _SAR_DEMO environment variable with cross checking with the MacroServer (given by `UNITTEST_DOOR_NAME`)

ready = False

init(*door_name=None*)

getElements(*elem_type='all'*)

Return the name of sardemo element(s) of given elem type

Parameters **elem_type** – (str) type of elemnts to return (all by default)

Returns (list<str>)

getMoveables()

Return the name of moveable(s) defined by SarDemo

Returns (list<str>)

getControllers()

Return the name of controllers(s) defined by SarDemo

Returns (list<str>)

getCTs()

Return the name of counter timer exp channel(s) defined by SarDemo

Returns (list<str>)

getMotors()

Return the name of motor(s) defined by SarDemo

Returns (list<str>)

getPseudoMotors()

Return the name of pseudomotor(s) defined by SarDemo

Returns (list<str>)

getZerods()

Return the name of zerod exp channel(s) defined by SarDemo

Returns (list<str>)

getOneds()

Return the name of one exp channel(s) defined by SarDemo

Returns (list<str>)

getTwods()
Return the name of two exp channel(s) defined by SarDemo

Returns (list<str>)

changeDoor(*door_name*)
Change the door name and reset all lists

Sardana migration guide

This chapter describes how to migrate different sardana components between the different API versions.

How to migrate your macro code

API v0 -> v1

This chapter describes the necessary steps to fully migrate your macros from *API v0* (sardana 0.x) to *API v1* (sardana 1.x)

Mandatory changes

The following are the 2 necessary changes to make your macros work in sardana *API v1*:

1. from:

   ```
   from macro import Macro, Type, Table, List
   ```

   to:

   ```
   from sardana.macroserver.macro import Macro, Type, Table, List
   ```

2. Parameter type `Type.Motor` should be changed `Type.Moveable`. In **v0** the *Motor* meant any motor (including physical motor, pseudo motor). In **v1**, for consistency, *Motor* means only physical motor and *Moveable* means all moveable elements (including physical motor, pseudo motor).

New features in API v1

This chapter is a summary of all new features in *API v1*.

1. Macros can now be functions(see *Writing macros*).

How to migrate your controller code

API v0 -> v1

This chapter describes the necessary steps to fully migrate your controller from *API v0* (sardana 0.x) to *API v1* (sardana 1.x)

Mandatory changes

The following are the 2 necessary changes to make your controller work in sardana *API v1*:

1. from:

   ```
   import pool
   from pool import <ControllerClass>/PoolUtil
   ```

   to:

   ```
   from sardana import pool
   from sardana.pool import PoolUtil
   from sardana.pool.controller import <ControllerClass>
   ```

2. change contructor from:

   ```
   def __init__(self, inst, props):
       code
   ```

   to:

   ```
   def __init__(self, inst, props, *args, **kwargs):
       MotorController.__init__(self, inst, props, *args, **kwargs)
       code
   ```

   (and don't forget to call the super class constructor also with args and kwargs).

The following change is not mandatory but is necessary in order for your controller to be recognized by the pool to be a *API v1* controller:

3. _log member changed from `logging.Logger`[877] to `taurus.core.util.Logger`[878]. This means that you need to change code from:

   ```
   self._log.setLevel(logging.INFO)
   ```

   to:

   ```
   self._log.setLogLevel(logging.INFO)
   ```

   or:

   ```
   self._log.setLogLevel(taurus.Info)
   ```

   since taurus.Info == logging.INFO.

Optional changes

The following changes are not necessary to make your controller work. The *API v1* supports the *API v0* on these matters.

1. **class members**:

1. from: *`class_prop`* to: *`ctrl_properties`*

2. from: *`ctrl_extra_attributes`* to: *`axis_attributes`*

[877] https://docs.python.org/dev/library/logging.html#logging.Logger
[878] http://taurus-scada.org/devel/api/taurus/core/util/_Logger.html#taurus.core.util.Logger

3. new feature in *API v1*: `ctrl_attributes`
3. **data types**:
   (a) `StateOne()` **return type**: Previously `StateOne()` had to return a member of `PyTango.DevState`. Now it **can** instead return a member of `State`. This eliminates the need to import `PyTango`.
   (b) In *API v0* class member (like `ctrl_extra_attributes`) value for key *type* had to be a string (like 'PyTango.DevString' or 'PyTango.DevDouble'). Now they can be a python type (like str or float). Please check *Data Type definition* for more information.
4. **generic controller method names**:
   (a) from: `GetPar()` to: `GetAxisPar()`
   (b) from: `SetPar()` to: `SetAxisPar()`
   (c) from: `GetExtraAttributePar()` to: `GetAxisExtraPar()`
   (d) from: `SetExtraAttributePar()` to: `SetAxisExtraPar()`
   (e) new feature in *API v1*: `GetCtrlPar()`, `SetCtrlPar()`
   (f) new feature in *API v1*: `AbortAll()` (has default implementation which calls `AbortOne()` for each axis)
5. **pseudo motor controller method names**:
   (a) from: `calc_pseudo()` to: `CalcPseudo()`
   (b) from: `calc_physical()` to: `CalcPhysical()`
   (c) from: `calc_all_pseudo()` to: `CalcAllPseudo()`
   (d) from: `calc_all_physical()` to: `CalcAllPhysical()`
   (e) new feature in *API v1*: `GetMotor()`
   (f) new feature in *API v1*: `GetPseudoMotor()`

New features in API v1

This chapter is a summary of all new features in *API v1*.

New controller features:

1. All Controllers now have a `ctrl_attributes` class member to define extra controller attributes (and new methods: `GetCtrlPar()`, `SetCtrlPar()`)
2. For `ctrl_properties`, `axis_attributes` and `ctrl_extra_attributes`:
   - **new (more pythonic) syntax. Old syntax is still supported:**
     - can replace data type strings for python type ('PyTango.DevDouble' -> float)
     - Default behavior. Example: before data access needed to be described explicitly. Now it is read-write by default.
     - support for 2D
     - new keys 'fget' and 'fset' override default method calls
3. no need to import `PyTango` (`StateOne()` can return sardana.State.On instead of PyTango.DevState.ON)

4. PseudoMotorController has new `GetMotor()` and `GetPseudoMotor()`
5. new `AbortAll()` (with default implementation which calls `AbortOne()` for each axis)
6. new `StopOne()` (with default implementation which calls `AbortOne()`)
7. new `StopAll()` (with default implementation which calls `StoptOne()` for each axis)
8. **new `GetAxisAttributes()` allows features like:**
    (a) per axis customized dynamic attributes
    (b) Basic interface (example: motor without velocity or acceleration)
    (c) Discrete motor (declare position has an integer instead of a float). No need for IORegisters anymore
9. **New `MotorController` constants:**
    - `HomeLimitSwitch`;
    - `UpperLimitSwitch`;
    - `LowerLimitSwitch`

New acquisition features:

1. Measurement group has a new *Configuration* attribute which contains the full description of the experiment in JSON format

New Tango API features:

1. Controllers are now Tango devices
2. Pool has a default PoolPath (points to <pool install dir>/poolcontrollers)
3. Create* commands can receive JSON object or an old style list of parameters
4. new CreateElement command (can replace CreateMotor, CreateExpChannel, etc)
5. Pool Abort command: aborts all elements (non pseudo elements)
6. Pool Stop command: stops all elements (non pseudo elements)
7. Controller Abort command: aborts all controller elements
8. Controller Stop command: stops all controller elements
9. Controllers have a LogLevel attribute which allows remote python logging management

Others:

1. Pool device is a python device :-)
2. many command line parameters help logging, debugging

Examples

Macro examples

Macro parameter examples

This chapter consists of a series of examples demonstrating how to declare macros which receive parameter(s).

```python
##############################################################################
##
# This file is part of Sardana
##
# http://www.sardana-controls.org/
##
# Copyright 2011 CELLS / ALBA Synchrotron, Bellaterra, Spain
##
# Sardana is free software: you can redistribute it and/or modify
# it under the terms of the GNU Lesser General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
##
# Sardana is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU Lesser General Public License for more details.
##
# You should have received a copy of the GNU Lesser General Public License
# along with Sardana.  If not, see <http://www.gnu.org/licenses/>.
##
##############################################################################

"""This module contains macros that demonstrate the usage of macro parameters"""

from sardana.macroserver.macro import *

__all__ = ["pt0", "pt1", "pt2", "pt3", "pt3d", "pt4", "pt5", "pt6", "pt7",
           "pt7d1", "pt7d2", "pt8", "pt9", "pt10", "pt11", "pt12", "pt13",
           "pt14", "pt14d", "twice"]


class pt0(Macro):
    """Macro without parameters. Pretty dull.
       Usage from Spock, ex.:
       pt0
       """

    param_def = []

    def run(self):
        pass


class pt1(Macro):
    """Macro with one float parameter: Each parameter is described in the
    param_def sequence as being a sequence of four elements: name, type,
    default value and description.
    Usage from Spock, ex.:
    pt1 1
    """

    param_def = [['value', Type.Float, None, 'some bloody float']]

    def run(self, f):
        pass

```

(continues on next page)

(continued from previous page)

```

class pt2(Macro):
    """Macro with one Motor parameter: Each parameter is described in the
    param_def sequence as being a sequence of four elements: name, type,
    default value and description.
    Usage from Spock, ex.
    pt2 mot1
    """

    param_def = [['motor', Type.Motor, None, 'some bloody motor']]

    def run(self, m):
        pass


class pt3(Macro):
    """Macro with a list of numbers as parameter: the type is a sequence of
    parameter types which is repeated. In this case it is a repetition of a
    float so only one parameter is defined.
    By default the repetition as a semantics of 'at least one'
    Usages from Spock, ex.:
    pt3 [1 34 15]
    pt3 1 34 15
    """

    param_def = [
        ['numb_list', [['pos', Type.Float, None, 'value']], None, 'List of values'],
    ]

    def run(self, *args, **kwargs):
        pass


class pt3d(Macro):
    """Macro with a list of numbers as parameter: the type is a sequence of
    parameter types which is repeated. In this case it is a repetition of a
    float so only one parameter is defined. The parameter has a default value.
    By default the repetition as a semantics of 'at least one'
    Usages from Spock, ex.:
    pt3d [1 34 15]
    pt3d 1 34 15
    Usage taken the default value, ex.:
    pt3d [1 [] 15]
    """

    param_def = [
        ['numb_list', [['pos', Type.Float, 21, 'value']], None, 'List of values'],
    ]

    def run(self, *args, **kwargs):
        pass


class pt4(Macro):
    """Macro with a list of motors as parameter: the type is a sequence of
    parameter types which is repeated. In this case it is a repetition of a
```

(continues on next page)

(continued from previous page)

```
    motor so only one parameter is defined.
    By default the repetition as a semantics of 'at least one'.
    Usages from Spock, ex.:
    pt4 [mot1 mot2 mot3]
    pt4 mot1 mot2 mot3
    """

    param_def = [
        ['motor_list', [['motor', Type.Motor, None, 'motor name']],
            None, 'List of motors'],
    ]

    def run(self, *args, **kwargs):
        pass


class pt5(Macro):
    """Macro with a motor parameter followed by a list of numbers.
    Usages from Spock, ex.:
    pt5 mot1 [1 3]
    pt5 mot1 1 3
    """

    param_def = [
        ['motor', Type.Motor, None, 'Motor to move'],
        ['numb_list', [['pos', Type.Float, None, 'value']], None, 'List of values'],
    ]

    def run(self, *args, **kwargs):
        pass


class pt6(Macro):
    """Macro with a motor parameter followed by a list of numbers. The list as
    explicitly stated an optional last element which is a dictionary that defines the
    min and max values for repetitions.
    Usages from Spock, ex.:
    pt6 mot1 [1 34 1]
    pt6 mot1 1 34 1
    """

    param_def = [
        ['motor', Type.Motor, None, 'Motor to move'],
        ['numb_list', [['pos', Type.Float, None, 'value'], {
            'min': 1, 'max': None}], None, 'List of values'],
    ]

    def run(self, *args, **kwargs):
        pass


class pt7(Macro):
    """Macro with a list of pair Motor,Float.
    Usages from Spock, ex.:
    pt7 [[mot1 1] [mot2 3]]
    pt7 mot1 1 mot2 3
```

(continues on next page)

(continued from previous page)

```
    """

    param_def = [
        ['m_p_pair', [['motor', Type.Motor, None, 'Motor to move'],
                      ['pos',   Type.Float, None, 'Position to move to']],
         None, 'List of motor/position pairs']
    ]

    def run(self, *args, **kwargs):
        pass


class pt7d1(Macro):
    """Macro with a list of pair Motor,Float. Default value for last ParamRepeat␣
↪element.
    Usages from Spock, ex.:
    pt7d1 [[mot1 1] [mot2 3]]
    pt7d1 mot1 1 mot2 3
    Using default value, ex.:
    pt7d1 [[mot1] [mot2 3]] # at any repetition

    """

    param_def = [
        ['m_p_pair', [['motor', Type.Motor, None, 'Motor to move'],
                      ['pos',   Type.Float, 2, 'Position to move to']],
         None, 'List of motor/position pairs']
    ]

    def run(self, *args, **kwargs):
        pass


class pt7d2(Macro):
    """Macro with a list of pair Motor,Float. Default value for both ParamRepeat␣
↪elements.
    Usages from Spock, ex.:
    pt7d2 [[mot1 1] [mot2 3]]
    pt7d2 mot1 1 mot2 3
    Using both default values, ex.:
    pt7d2 [[] [mot2 3] []] # at any repetition
    """

    param_def = [
        ['m_p_pair', [['motor', Type.Motor, 'mot1', 'Motor to move'],
                      ['pos',   Type.Float, 2, 'Position to move to']],
         None, 'List of motor/position pairs']
    ]

    def run(self, *args, **kwargs):
        pass


class pt8(Macro):
    """Macro with a list of pair Motor,Float. The min and max elements have been
    explicitly stated.
```

(continues on next page)

(continued from previous page)

```
    Usages from Spock, ex.:
    pt8 [[mot1 1] [mot2 3]]
    pt8 mot1 1 mot2 3
    """

    param_def = [
        ['m_p_pair', [['motor', Type.Motor, None, 'Motor to move'],
                      ['pos',   Type.Float, None, 'Position to move to'],
                      {'min': 1, 'max': 2}],
         None, 'List of motor/position pairs']
    ]

    def run(self, *args, **kwargs):
        pass


class pt9(Macro):
    """Same as macro pt7 but with old style ParamRepeat. If you are writing
    a macro with variable number of parameters for the first time don't even
    bother to look at this example since it is DEPRECATED.
    Usages from Spock, ex.:
    pt9 [[mot1 1][mot2 3]]
    pt9 mot1 1 mot2 3
    """

    param_def = [
        ['m_p_pair',
         ParamRepeat(['motor', Type.Motor, None, 'Motor to move'],
                     ['pos',  Type.Float, None, 'Position to move to'], min=1, max=2),
         None, 'List of motor/position pairs'],
    ]

    def run(self, *args, **kwargs):
        pass


class pt10(Macro):
    """Macro with list of numbers followed by a motor parameter. The repeat
    parameter may be defined as first one.
    Usage from Spock, ex.:
    pt10 [1 3] mot1
    """

    param_def = [
        ['numb_list', [['pos', Type.Float, None, 'value']], None, 'List of values'],
        ['motor', Type.Motor, None, 'Motor to move']
    ]

    def run(self, *args, **kwargs):
        pass


class pt11(Macro):
    """Macro with counter parameter followed by a list of numbers, followed by
    a motor parameter. The repeat parameter may be defined in between other
    parameters.
```

(continues on next page)

(continued from previous page)

```
    Usages from Spock, ex.:
    pt11 ct1 [1 3] mot1
    """

    param_def = [
        ['counter', Type.ExpChannel, None, 'Counter to count'],
        ['numb_list', [['pos', Type.Float, None, 'value']], None, 'List of values'],
        ['motor', Type.Motor, None, 'Motor to move']
    ]

    def run(self, *args, **kwargs):
        pass


class pt12(Macro):
    """Macro with list of motors followed by list of numbers. Two repeat
    parameters may defined.
    Usage from Spock, ex.:
    pt12 [1 3 4] [mot1 mot2]
    """

    param_def = [
        ['numb_list', [['pos', Type.Float, None, 'value']], None, 'List of values'],
        ['motor_list', [['motor', Type.Motor, None, 'Motor to move']],
         None, 'List of motors']
    ]

    def run(self, *args, **kwargs):
        pass


class pt13(Macro):
    """Macro with list of motors groups, where each motor group is a list of
    motors. Repeat parameters may be defined as nested.
    Usage from Spock, ex.:
    pt13 [[mot1 mot2] [mot3 mot4]]
"""

    param_def = [
        ['motor_group_list',
         [['motor_list', [['motor', Type.Motor, None, 'Motor to move']],
           None, 'List of motors']],
         None, 'Motor groups']
    ]

    def run(self, *args, **kwargs):
        pass


class pt14(Macro):
    """Macro with list of motors groups, where each motor group is a list of
    motors and a float. Repeat parameters may be defined as nested.
    Usage from Spock, ex.:
    pt14 [[[mot1 mot2] 3] [[mot3] 5]]
    """

```

(continues on next page)

(continued from previous page)

```
    param_def = [
        ['motor_group_list',
         [['motor_list', [['motor', Type.Motor, None, 'Motor to move']], None, 'List␣
↪of motors'],
          ['float', Type.Float, None, 'Number']],
         None, 'Motor groups']
    ]

    def run(self, *args, **kwargs):
        pass


class pt14d(Macro):
    """Macro with list of motors groups, where each motor group is a list of
    motors and a float. Repeat parameters may be defined as nested.
    Default values can be used.
    Usages taken default values, ex.:
    pt14d [[[mot1 mot2] 3] [[mot3] []]]
    pt14d [[[mot1 []] 3] [[mot3] []]]
    pt14d [[[[]] 3] [[mot3] []]]
    """

    param_def = [
        ['motor_group_list',
         [['motor_list', [['motor', Type.Motor, 'mot1', 'Motor to move']], None,
↪'List of motors'],
          ['float', Type.Float, 33, 'Number']],
         None, 'Motor groups']
    ]

    def run(self, *args, **kwargs):
        pass


class twice(Macro):
    """A macro that returns a float that is twice its input. It also sets its
    data to be a dictionary with 'in','out' as keys and value,result
    as values, respectively"""

    # uncomment the following lines as necessary. Otherwise you may delete them
    param_def = [["value", Type.Float, 23, "value to be doubled"]]
    result_def = [["result", Type.Float, None,
                   "the double of the given value"]]
    #hints = {}
    # env = (,)

    # uncomment the following lines if need prepare. Otherwise you may delete them
    # def prepare(self):
    #    pass

    def run(self, n):
        ret = 2 * n
        self.setData({'in': n, 'out': ret})
        return ret
```

Macro call examples

This chapter consists of a series of examples demonstrating how to call macros from inside a macro

```
##############################################################################
##
# This file is part of Sardana
##
# http://www.sardana-controls.org/
##
# Copyright 2011 CELLS / ALBA Synchrotron, Bellaterra, Spain
##
# Sardana is free software: you can redistribute it and/or modify
# it under the terms of the GNU Lesser General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
##
# Sardana is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU Lesser General Public License for more details.
##
# You should have received a copy of the GNU Lesser General Public License
# along with Sardana.  If not, see <http://www.gnu.org/licenses/>.
##
##############################################################################

"""
A macro package to show examples on how to run a macro from inside another macro
"""

__all__ = ["call_wa", "call_wm", "subsubm", "subm", "mainmacro", "runsubs"]

__docformat__ = 'restructuredtext'

from sardana.macroserver.macro import Macro, Type, ParamRepeat

#-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-
# First example:
# A 'mainmacro' that executes a 'subm' that in turn executes a 'subsubm'.
# The 'subsubm' macro itself calls a short ascan macro
#-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~


class call_wa(Macro):

    def run(self):
        self.macros.wa()


class call_wm(Macro):

    param_def = [
        ['motor_list',
         ParamRepeat(['motor', Type.Motor, None, 'Motor to move']),
         None, 'List of motor to show'],
    ]
```

(continues on next page)

(continued from previous page)

```

    def run(self, m):
        self.macros.wm(m)


class subsubm(Macro):
    """this macro just calls the 'subm' macro
    This macro is part of the examples package. It was written for demonstration␣
↪purposes"""

    def run(self):
        self.output("Starting %s" % self.getName())
        m = self.macros
        motors = self.getObjs('.*', type_class=Type.Motor)
        m.ascan(motors[0], 0, 100, 10, 0.2)
        self.output("Finished %s" % self.getName())


class subm(Macro):
    """this macro just calls the 'subsubm' macro
    This macro is part of the examples package. It was written for demonstration␣
↪purposes"""

    def run(self):
        self.output("Starting %s" % self.getName())
        self.macros.subsubm()
        self.output("Finished %s" % self.getName())


class mainmacro(Macro):
    """this macro just calls the 'subm' macro
    This macro is part of the examples package. It was written for demonstration␣
↪purposes"""

    def run(self):
        self.output("Starting %s" % self.getName())
        self.macros.subm()
        self.output("Finished %s" % self.getName())

#-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-
# Second example:
# a 'runsubs' macro that shows the different ways to call a macro from inside
# another macro
#-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~-~


class runsubs(Macro):
    """ A macro that calls a ascan macro using the motor given as first parameter.

    This macro is part of the examples package. It was written for demonstration␣
↪purposes

    Call type will allow to choose to format in which the ascan macro is called
    from this macro:
    1 - m.ascan(motor.getName(), '0', '10', '4', '0.2')
    2 - m.ascan(motor, 0, 10, 4, 0.2)
```

(continues on next page)

(continued from previous page)

```
    3 - self.execMacro('ascan', motor.getName(), '0', '10', '4', '0.2')
    4 - self.execMacro(['ascan', motor, 0, 10, 4, 0.2])
    5 - params = 'ascan', motor, 0, 10, 4, 0.2
        self.execMacro(params)
    6 - self.execMacro("ascan %s 0 10 4 0.2" % motor.getName())
    7 - macro, prep = self.createMacro("ascan %s 0 10 4 0.2" % motor.getName())
        macro.hooks = [ self.hook ]
        self.runMacro(macro)
    8 - macro, prep = self.createMacro('ascan', motor, 0, 10, 4, 0.2)
        macro.hooks = [ self.hook ]
        self.runMacro(macro)
    9 - params = 'ascan', motor, 0, 10, 4, 0.2
        macro, prep = self.createMacro(params)
        macro.hooks = [ self.hook ]
        self.runMacro(macro)

        Options 7,8 and 9 use the lower level macro API in order to be able to
        attach hooks to the ascan macro."""
    param_def = [
        ['motor',      Type.Motor,   None, 'Motor to move'],
        ['call_type',  Type.Integer, 2, 'type of run to execute internally'],
    ]

    def hook(self):
        self.info("executing hook in a step of a scan...")

    def run(self, motor, call_type):
        m = self.macros
        self.output("Using type %d" % call_type)
        if call_type == 1:
            m.ascan(motor.getName(), '0', '10', '4', '0.2')
        elif call_type == 2:
            m.ascan(motor, 0, 10, 4, 0.2)
        elif call_type == 3:
            self.execMacro('ascan', motor.getName(), '0', '10', '4', '0.2')
        elif call_type == 4:
            self.execMacro('ascan', motor, 0, 10, 4, 0.2)
        elif call_type == 5:
            params = 'ascan', motor, 0, 10, 4, 0.2
            self.execMacro(params)
        elif call_type == 6:
            self.execMacro("ascan %s 0 10 4 0.2" % motor.getName())
        elif call_type == 7:
            macro, prep = self.createMacro("ascan %s 0 10 4 0.2" %
                                           motor.getName())
            macro.hooks = [self.hook]
            self.runMacro(macro)
        elif call_type == 8:
            macro, prep = self.createMacro('ascan', motor, 0, 10, 4, 0.2)
            macro.hooks = [self.hook]
            self.runMacro(macro)
        elif call_type == 9:
            params = 'ascan', motor, 0, 10, 4, 0.2
            macro, prep = self.createMacro(params)
            macro.hooks = [self.hook]
            self.runMacro(macro)
```

(continues on next page)

(continued from previous page)

```python


class get_data(Macro):
    """A macro that executes another macro from within it, get its data,
    and calculates a result using this data.

    This macro is part of the examples package. It was written for
    demonstration purposes"""

    param_def = [["mot", Type.Moveable, None, "moveable to be moved"]]
    result_def = [["middle", Type.Float, None,
                   "the middle motor position"]]

    def run(self, mot):
        start = 0
        end = 2
        intervals = 2
        integtime = 0.1
        positions = []
        dscan, _ = self.createMacro('dscan',
                                    mot, start, end, intervals, integtime)
        self.runMacro(dscan)

        data = dscan.data
        len_data = len(data)
        for point_nb in xrange(len_data):
            position = data[point_nb].data[mot.getName()]
            positions.append(position)

        middle_pos = max(positions) - min(positions) / len_data
        return middle_pos
```

Macro plotting examples

This chapter consists of a series of examples demonstrating how to plot graphics from inside a macro.

The complete set of `pyplot`[879] examples can be found here[880]

```python
import math
from numpy import linspace
from scipy.integrate import quad
from scipy.special import j0

from sardana.macroserver.macro import macro, Type


def j0i(x):
    """Integral form of J_0(x)"""
    def integrand(phi):
        return math.cos(x * math.sin(phi))
    return (1.0 / math.pi) * quad(integrand, 0, math.pi)[0]

```

(continues on next page)

[879] https://matplotlib.org/api/_as_gen/matplotlib.pyplot.html#module-matplotlib.pyplot
[880] https://matplotlib.org/gallery/index.html#examples-index

(continued from previous page)

```

@macro()
def J0_plot(self):
    """Sample J0 at linspace(0, 20, 200)"""
    x = linspace(0, 20, 200)
    y = j0(x)
    x1 = x[::10]
    y1 = map(j0i, x1)
    self.pyplot.plot(x, y, label=r'$J_0(x)$')
    self.pyplot.plot(x1, y1, 'ro', label=r'$J_0^{integ}(x)$')
    self.pyplot.title(
        r'Verify $J_0(x)=\frac{1}{\pi}\int_0^{\pi}\cos(x \sin\phi)\,d\phi$')
    self.pyplot.xlabel('$x$')
    self.pyplot.legend()


from numpy import random


@macro()
def random_image(self):
    """Shows a random image 32x32"""
    img = random.random((32, 32))
    self.pyplot.matshow(img)

import numpy


@macro([["interactions", Type.Integer, None, ""],
        ["density", Type.Integer, None, ""]])
def mandelbrot(self, interactions, density):

    x_min, x_max = -2, 1
    y_min, y_max = -1.5, 1.5

    x, y = numpy.meshgrid(numpy.linspace(x_min, x_max, density),
                          numpy.linspace(y_min, y_max, density))

    c = x + 1j * y
    z = c.copy()

    fractal = numpy.zeros(z.shape, dtype=numpy.uint8) + 255

    finteractions = float(interactions)
    for n in range(interactions):
        z *= z
        z += c
        mask = (fractal == 255) & (abs(z) > 10)
        fractal[mask] = 254 * n / finteractions
    self.pyplot.imshow(fractal)
```

Macro input examples

This chapter consists of a series of examples demonstrating how to ask for user input inside macros.

A tutorial on macro input parameter can be found *here*. The *API* documentation: `input()`

```python

from sardana.macroserver.macro import imacro, Type


@imacro()
def ask_number_of_points(self):
    """asks user for the number of points"""

    nb_points = self.input("How many points?", data_type=Type.Integer)


@imacro()
def ask_for_moveable(self):
    """asks user for a motor"""

    moveable = self.input("Which moveable?", data_type=Type.Moveable)
    self.output("You selected %s which is at %f",
                moveable, moveable.getPosition())


@imacro()
def ask_for_car_brand(self):
    """asks user for a car brand"""

    car_brands = "Mazda", "Citroen", "Renault"
    car_brand = self.input("Which car brand?", data_type=car_brands)
    self.output("You selected %s", car_brand)


@imacro()
def ask_for_multiple_car_brands(self):
    """asks user for several car brands"""

    car_brands = "Mazda", "Citroen", "Renault", "Ferrari", "Porche", "Skoda"
    car_brands = self.input("Which car brand(s)?", data_type=car_brands,
                            allow_multiple=True, title="Favorites")
    self.output("You selected %s", ", ".join(car_brands))


@imacro()
def ask_peak(self):
    """asks user for peak current of points with a custom title"""

    peak = self.input("What is the peak current?", data_type=Type.Float,
                      title="Peak selection")
    self.output("You selected a peak of %f A", peak)


@imacro()
def ask_peak_v2(self):
    """asks user for peak current of points with a custom title,
    default value, label and units"""

    label, unit = "peak", "mA"
    peak = self.input("What is the peak current?", data_type=Type.Float,
                      title="Peak selection", key=label, unit=unit,
                      default_value=123.4)
```

(continues on next page)

(continued from previous page)

```
    self.output("You selected a %s of %f %s", label, peak, unit)


@imacro()
def ask_peak_v3(self):
    """asks user for peak current of points with a custom title,
    default value, label, units and ranges"""

    label, unit = "peak", "mA"
    peak = self.input("What is the peak current?", data_type=Type.Float,
                      title="Peak selection", key=label, unit=unit,
                      default_value=123.4, minimum=0.0, maximum=200.0)
    self.output("You selected a %s of %f %s", label, peak, unit)


@imacro()
def ask_peak_v4(self):
    """asks user for peak current of points with a custom title,
    default value, label, units, ranges and step size"""

    label, unit = "peak", "mA"
    peak = self.input("What is the peak current?", data_type=Type.Float,
                      title="Peak selection", key=label, unit=unit,
                      default_value=123.4, minimum=0.0, maximum=200.0,
                      step=5)
    self.output("You selected a %s of %f %s", label, peak, unit)


@imacro()
def ask_peak_v5(self):
    """asks user for peak current of points with a custom title,
    default value, label, units, ranges, step size and decimal places"""

    label, unit = "peak", "mA"
    peak = self.input("What is the peak current?", data_type=Type.Float,
                      title="Peak selection", key=label, unit=unit,
                      default_value=123.4, minimum=0.0, maximum=200.0,
                      step=5, decimals=2)
    self.output("You selected a %s of %f %s", label, peak, unit)
```

Controller examples

Sardana development guidelines

Overview

This document describes sardana from the perspective of developers. Most importantly, it gives information for people who want to contribute code to the development of sardana. So if you want to help out, read on!

How to contribute to sardana

Sardana development is managed with the Sardana github project[881].

Apart from directly contributing code, you can contribute to sardana in many ways, such as reporting bugs or proposing new features. In all cases you will probably need a github account and you are strongly encouragedto subscribe to the sardana-devel and sardana-users mailing lists[882].

The rest of this document will focus on how to contribute code.

Cloning and forking sardana from Git

You are welcome to clone the Sardana code from our main Git repository:

```
git clone https://github.com/sardana-org/sardana.git sardana
```

Code contributions (bug patches, new features) are welcome, but the review process/workflow for accepting new code is yet to be discussed. For the moment, use the sardana-devel mailing list for proposing patches.

Note that you can also fork the git repository in github to get your own github-hosted clone of the sardana repository to which you will have full access. This will create a new git repository associated to your personal account in github, so that your changes can be easily shared and eventually merged into the official repository.

Documentation

All standalone documentation should be written in plain text (`.rst`) files using reStructuredText[883] for markup and formatting. All such documentation should be placed in directory `docs/source` of the sardana source tree. The documentation in this location will serve as the main source for sardana documentation and all existing documentation should be converted to this format.

Coding conventions

- In general, we try to follow the standard Python style conventions as described in Style Guide for Python Code[884]
- Code **must** be python 2.6 compatible
- Use 4 spaces for indentation
- In the same file, different classes should be separated by 2 lines
- use `lowercase` for module names. If possible prefix module names with the word `sardana` (like `sardanautil.py`) to avoid import mistakes.
- use `CamelCase` for class names
- python module first line should be:

[881] https://github.com/sardana-org/sardana
[882] https://sourceforge.net/p/sardana/mailman/
[883] http://docutils.sourceforge.net/rst.html
[884] http://www.python.org/peps/pep-0008.html

```
#!/usr/bin/env python
```

- python module should contain license information (see template below)
- avoid poluting namespace by making private definitions private (__ prefix) or/and implementing `__all__` (see template below)
- whenever a python module can be executed from the command line, it should contain a `main` function and a call to it in a `if __name__ == "__main__"` like statement (see template below)
- document all code using Sphinx[885] extension to reStructuredText[886]

The following code can serve as a template for writing new python modules to sardana:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

##############################################################################
##
## This file is part of Sardana
##
## http://www.tango-controls.org/static/sardana/latest/doc/html/index.html
##
## Copyright 2011 CELLS / ALBA Synchrotron, Bellaterra, Spain
##
## Sardana is free software: you can redistribute it and/or modify
## it under the terms of the GNU Lesser General Public License as published by
## the Free Software Foundation, either version 3 of the License, or
## (at your option) any later version.
##
## Sardana is distributed in the hope that it will be useful,
## but WITHOUT ANY WARRANTY; without even the implied warranty of
## MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
## GNU Lesser General Public License for more details.
##
## You should have received a copy of the GNU Lesser General Public License
## along with Sardana.  If not, see <http://www.gnu.org/licenses/>.
##
##############################################################################

"""A :mod:`sardana` module written for template purposes only"""

__all__ = ["SardanaDemo"]

__docformat__ = "restructuredtext"

class SardanaDemo(object):
    """This class is written for template purposes only"""

def main():
    print "SardanaDemo"s

if __name__ == "__main__":
    main()
```

885 http://sphinx.pocoo.org/
886 http://docutils.sourceforge.net/rst.html

1.1.3 Sardana Enhancement Proposals

1.1.4 Glossary

`...` The default Python prompt of the interactive shell when entering code for an indented code block or within a pair of matching left and right delimiters (parentheses, square brackets or curly braces).

>>> The default Python prompt of the interactive shell. Often seen for code examples which can be executed interactively in the interpreter.

ADC In electronics, an analog-to-digital converter (ADC) is a system that converts an analog signal e.g. voltage into its digital representation.

API An application programming interface (API) is a particular set of rules and specifications that software programs can follow to communicate with each other. It serves as an interface between different software programs and facilitates their interaction, similar to the way the user interface facilitates interaction between humans and computers. An API can be created for applications, libraries, operating systems, etc., as a way of defining their "vocabularies" and resources request conventions (e.g. function-calling conventions). It may include specifications for routines, data structures, object classes, and protocols used to communicate between the consumer program and the implementer program of the API.

argument A value passed to a function or method, assigned to a named local variable in the function body. A function or method may have both positional arguments and keyword arguments in its definition. Positional and keyword arguments may be variable-length: `*` accepts or passes (if in the function definition or call) several positional arguments in a list, while `**` does the same for keyword arguments in a dictionary.

Any expression may be used within the argument list, and the evaluated value is passed to the local variable.

attribute A value associated with an object which is referenced by name using dotted expressions. For example, if an object *o* has an attribute *a* it would be referenced as *o.a*.

dictionary An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

CCD A charge-coupled device (CCD) is a device for the movement of electrical charge, usually from within the device to an area where the charge can be manipulated, for example conversion into a digital value. This is achieved by "shifting" the signals between stages within the device one at a time. CCDs move charge between capacitive bins in the device, with the shift allowing for the transfer of charge between bins.

class A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

CLI A command-line interface (CLI) is a mechanism for interacting with a computer operating system or software by typing commands to perform specific tasks. This text-only interface contrasts with the use of a mouse pointer with a graphical user interface (*GUI*) to click on options, or menus on a text user interface (TUI) to select options. This method of instructing a computer to perform a given task is referred to as "entering" a command: the system waits for the user to conclude the submitting of the text command by pressing the "Enter" key (a descendant of the "carriage return" key of a typewriter keyboard). A command-line interpreter then receives, parses, and executes the requested user command. The command-line interpreter may be run in a text terminal or in a terminal emulator window as a remote shell client such as PuTTY. Upon completion, the command usually returns output to the user in the form of text lines on the CLI. This output may be an answer if the command was a question, or otherwise a summary of the operation.

client-server model The client-server model of computing is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients. Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system. A server machine is a host that is running one or more server programs which share their resources with clients. A client does not share any of its resources, but requests a server's content or service function. Clients therefore initiate communication sessions with servers which await incoming requests.

closed loop A.k.a feedback loop, occurs when outputs of a system are routed back as inputs as part of a chain of cause-and-effect that forms a circuit or loop. In case of motion systems, closed loop positioning uses the position sensors e.g. encoders to measure the system's output. The measured signal is looped back to the control unit as input and is used to correct the moveable's position.

daemon In Unix and other computer multitasking operating systems, a daemon is a computer program that runs in the background, rather than under the direct control of a user. They are usually initiated as background processes. Typically daemons have names that end with the letter "d": for example, *syslogd*, the daemon that handles the system log, or *sshd*, which handles incoming SSH connections.

dial See *dial position*

dial position Position in controller units (See also *user position*).

expression A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statement*s which cannot be used as expressions, such as `print()`[887] or `if`[888]. Assignments are also statements, not expressions.

function A series of statements which returns some value to a caller. It can also be passed zero or more arguments which may be used in the execution of the body. See also *argument* and *method*.

generator A function which returns an iterator. It looks like a normal function except that it contains `yield`[889] statements for producing a series a values usable in a for-loop or that can be retrieved one at a time with the `next()`[890] function. Each `yield`[891] temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator resumes, it picks-up where it left-off (in contrast to functions which start fresh on every invocation).

generator expression An expression that returns an iterator. It looks like a normal expression followed by a `for`[892] expression defining a loop variable, range, and an optional `if`[893] expression. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))        # sum of squares 0, 1, 4, ... 81
285
```

GUI A graphical user interface (GUI) is a type of user interface that allows users to interact with electronic devices with images rather than text commands. GUIs can be used in computers, hand-held devices such as MP3 players, portable media players or gaming devices, household appliances and office equipment. A GUI represents the information and actions available to a user through graphical icons and visual indicators such as secondary notation, as opposed to text-based interfaces (*CLI*), typed command labels or text navigation. The actions are usually performed through direct manipulation of the graphical elements.

[887] https://docs.python.org/dev/library/functions.html#print
[888] https://docs.python.org/dev/reference/compound_stmts.html#if
[889] https://docs.python.org/dev/reference/simple_stmts.html#yield
[890] https://docs.python.org/dev/library/functions.html#next
[891] https://docs.python.org/dev/reference/simple_stmts.html#yield
[892] https://docs.python.org/dev/reference/compound_stmts.html#for
[893] https://docs.python.org/dev/reference/compound_stmts.html#if

interactive Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`).

interpreted Python is an interpreted language, as opposed to a compiled one, though the distinction can be blurry because of the presence of the bytecode compiler. This means that source files can be run directly without explicitly creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also *interactive*.

iterable An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`[894], `str`[895], and `tuple`[896]) and some non-sequence types like `dict`[897] and `file` and objects of any classes you define with an `__iter__()` or `__getitem__()` method. Iterables can be used in a `for`[898] loop and in many other places where a sequence is needed (`zip()`[899], `map()`[900], ...). When an iterable object is passed as an argument to the built-in function `iter()`[901], it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()`[902] or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

iterator An object representing a stream of data. Repeated calls to the iterator's `next()` method return successive items in the stream. When no more data are available a `StopIteration`[903] exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `next()` method just raise `StopIteration`[904] again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`[905]) produces a fresh new iterator each time you pass it to the `iter()`[906] function or use it in a `for`[907] loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

More information can be found in Iterator Types[908].

key function A key function or collation function is a callable that returns a value used for sorting or ordering. For example, `locale.strxfrm()`[909] is used to produce a sort key that is aware of locale specific sort conventions.

A number of tools in Python accept key functions to control how elements are ordered or grouped.

[894] https://docs.python.org/dev/library/stdtypes.html#list
[895] https://docs.python.org/dev/library/stdtypes.html#str
[896] https://docs.python.org/dev/library/stdtypes.html#tuple
[897] https://docs.python.org/dev/library/stdtypes.html#dict
[898] https://docs.python.org/dev/reference/compound_stmts.html#for
[899] https://docs.python.org/dev/library/functions.html#zip
[900] https://docs.python.org/dev/library/functions.html#map
[901] https://docs.python.org/dev/library/functions.html#iter
[902] https://docs.python.org/dev/library/functions.html#iter
[903] https://docs.python.org/dev/library/exceptions.html#StopIteration
[904] https://docs.python.org/dev/library/exceptions.html#StopIteration
[905] https://docs.python.org/dev/library/stdtypes.html#list
[906] https://docs.python.org/dev/library/functions.html#iter
[907] https://docs.python.org/dev/reference/compound_stmts.html#for
[908] https://docs.python.org/dev/library/stdtypes.html#typeiter
[909] https://docs.python.org/dev/library/locale.html#locale.strxfrm

They include `min()`[910], `max()`[911], `sorted()`[912], `list.sort()`[913], `heapq.nsmallest()`[914], `heapq.nlargest()`[915], and `itertools.groupby()`[916].

There are several ways to create a key function. For example. the `str.lower()`[917] method can serve as a key function for case insensitive sorts. Alternatively, an ad-hoc key function can be built from a `lambda`[918] expression such as `lambda r: (r[0], r[2])`. Also, the `operator`[919] module provides three key function constructors: `attrgetter()`[920], `itemgetter()`[921], and `methodcaller()`[922]. See the Sorting HOW TO[923] for examples of how to create and use key functions.

keyword argument Arguments which are preceded with a `variable_name=` in the call. The variable name designates the local name in the function to which the value is assigned. `**` is used to accept or pass a dictionary of keyword arguments. See *argument*.

lambda An anonymous inline function consisting of a single *expression* which is evaluated when the function is called. The syntax to create a lambda function is `lambda [arguments]: expression`

list A built-in Python *sequence*. Despite its name it is more akin to an array in other languages than to a linked list since access to elements are O(1).

list comprehension A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ["0x%02x" % x for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if`[924] clause is optional. If omitted, all elements in `range(256)` are processed.

MCA Multichannel Analyzer (MCA) is a device for …

method A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first *argument* (which is usually called `self`). See *function* and *nested scope*.

namespace The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `__builtin__.open()` and `os.open()`[925] are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()`[926] or `itertools.izip()` makes it clear that those functions are implemented by the `random`[927] and `itertools`[928] modules, respectively.

nested scope The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes work only for reference and not for assignment which will always write to the innermost scope. In contrast, local

[910] https://docs.python.org/dev/library/functions.html#min
[911] https://docs.python.org/dev/library/functions.html#max
[912] https://docs.python.org/dev/library/functions.html#sorted
[913] https://docs.python.org/dev/library/stdtypes.html#list.sort
[914] https://docs.python.org/dev/library/heapq.html#heapq.nsmallest
[915] https://docs.python.org/dev/library/heapq.html#heapq.nlargest
[916] https://docs.python.org/dev/library/itertools.html#itertools.groupby
[917] https://docs.python.org/dev/library/stdtypes.html#str.lower
[918] https://docs.python.org/dev/reference/expressions.html#lambda
[919] https://docs.python.org/dev/library/operator.html#module-operator
[920] https://docs.python.org/dev/library/operator.html#operator.attrgetter
[921] https://docs.python.org/dev/library/operator.html#operator.itemgetter
[922] https://docs.python.org/dev/library/operator.html#operator.methodcaller
[923] https://docs.python.org/dev/howto/sorting.html#sortinghowto
[924] https://docs.python.org/dev/reference/compound_stmts.html#if
[925] https://docs.python.org/dev/library/os.html#os.open
[926] https://docs.python.org/dev/library/random.html#random.seed
[927] https://docs.python.org/dev/library/random.html#module-random
[928] https://docs.python.org/dev/library/itertools.html#module-itertools

variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace.

new-style class Any class which inherits from `object`[929]. This includes all built-in types like `list`[930] and `dict`[931]. Only new-style classes can use Python's newer, versatile features like `__slots__`, descriptors, properties, and `__getattribute__()`.

object Any data with state (attributes or value) and defined behavior (methods). Also the ultimate base class of any *new-style class*.

OS An operating system (OS) is software, consisting of programs and data, that runs on computers, manages computer hardware resources, and provides common services for execution of various application software. Operating system is the most important type of system software in a computer system. Without an operating system, a user cannot run an application program on their computer, unless the application program is self booting.

PLC A programmable logic controller (PLC) is an industrial digital computer which has been ruggedised and adapted for the control of manufacturing processes, such as assembly lines, or robotic devices, or any activity that requires high reliability control e.g. equipment or personal protection.

plug-in a plug-in (or plugin) is a set of software components that adds specific abilities to a larger software application. If supported, plug-ins enable customizing the functionality of an application. For example, plug-ins are commonly used in web browsers to play video, scan for viruses, and display new file types.

plugin See *plug-in*.

positional argument The arguments assigned to local names inside a function or method, determined by the order in which they were given in the call. `*` is used to either accept multiple positional arguments (when in the definition), or pass several arguments as a list to a function. See *argument*.

Python 3000 Nickname for the Python 3.x release line (coined long ago when the release of version 3 was something in the distant future.) This is also abbreviated "Py3k".

Pythonic An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages. For example, a common idiom in Python is to loop over all elements of an iterable using a `for`[932] statement. Many other languages don't have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

```
for i in range(len(food)):
    print food[i]
```

As opposed to the cleaner, Pythonic method:

```
for piece in food:
    print piece
```

SCADA supervisory control and data acquisition (SCADA) generally refers to industrial control systems: computer systems that monitor and control industrial, infrastructure, or facility-based processes.

SDS Sardana Device server (SDS) is the sardana tango device server *daemon*.

sequence An *iterable* which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `len()` method that returns the length of the sequence. Some built-

[929] https://docs.python.org/dev/library/functions.html#object
[930] https://docs.python.org/dev/library/stdtypes.html#list
[931] https://docs.python.org/dev/library/stdtypes.html#dict
[932] https://docs.python.org/dev/reference/compound_stmts.html#for

in sequence types are `list`[933], `str`[934], `tuple`[935], and `unicode`. Note that `dict`[936] also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary immutable[937] keys rather than integers.

slice An object usually containing a portion of a *sequence*. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses `slice`[938] objects internally (or in older versions, `__getslice__()` and `__setslice__()`).

statement A statement is part of a suite (a "block" of code). A statement is either an *expression* or a one of several constructs with a keyword, such as `if`[939], `while`[940] or `for`[941].

stepper A stepper motor (or step motor) is a brushless DC electric motor that divides a full rotation into a number of equal steps. The motor's position can then be commanded to move and hold at one of these steps without any feedback sensor (an open-loop controller), as long as the motor is carefully sized to the application.

triple-quoted string A string which is bound by three instances of either a quotation mark (") or an apostrophe ('). While they don't provide any functionality not available with single-quoted strings, they are useful for a number of reasons. They allow you to include unescaped single and double quotes within a string and they can span multiple lines without the use of the continuation character, making them especially useful when writing docstrings.

type The type of a Python object determines what kind of object it is; every object has a type. An object's type is accessible as its `__class__` attribute or can be retrieved with `type(obj)`.

user See *user position*

user position Moveable position in user units (See also *dial position*). Dial and user units are related by the following expressions:

user = sign x dial + offset dial = controller_position / steps_per_unit

where *sign* is -1 or 1. *offset* can be any number and *steps_per_unit* must be non zero.

1.1.5 Documentation to be done

Todo: Update this chapter and distribute its contents logically around the documentation.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/sardana/checkouts/latest/doc/sou
line 8.)

Todo: document this chapter

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/sardana/checkouts/latest/doc/sou
line 6.)

[933] https://docs.python.org/dev/library/stdtypes.html#list
[934] https://docs.python.org/dev/library/stdtypes.html#str
[935] https://docs.python.org/dev/library/stdtypes.html#tuple
[936] https://docs.python.org/dev/library/stdtypes.html#dict
[937] https://docs.scipy.org/doc/numpy/glossary.html#term-immutable
[938] https://docs.python.org/dev/library/functions.html#slice
[939] https://docs.python.org/dev/reference/compound_stmts.html#if
[940] https://docs.python.org/dev/reference/compound_stmts.html#while
[941] https://docs.python.org/dev/reference/compound_stmts.html#for

Todo: complete 0D controller howto

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/sardana/checkouts/latest/doc/sou
line 9.)

Todo: document 1D controller howto

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/sardana/checkouts/latest/doc/sou
line 12.)

Todo: document 2D controller howto

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/sardana/checkouts/latest/doc/sou
line 12.)

Todo: document how to skip the readouts while acquiring

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/sardana/checkouts/latest/doc/sou
line 488.)

Todo: document IORegister controller howto

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/sardana/checkouts/latest/doc/sou
line 12.)

Todo: document pseudo motor controller howto

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/sardana/checkouts/latest/doc/sou
line 12.)

Todo: document how to write custom recorders

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/sardana/checkouts/latest/doc/sou
line 51.)

Todo: The FAQ is work-in-progress. Many answers need polishing and mostly links need to be added

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/sardana/checkouts/latest/doc/sou
line 5.)

Todo: This chapter is not ready… Sorry for inconvenience.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/sardana/checkouts/latest/doc/sou
line 62.)

Todo: This chapter is not ready… Sorry for inconvenience.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/sardana/checkouts/latest/doc/sou
line 67.)

Todo: This chapter is not ready… Sorry for inconvenince.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/sardana/checkouts/latest/doc/sou
line 127.)

Todo: Sardana Editor documentation to be written

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/sardana/checkouts/latest/doc/sou
line 12.)

Todo: This chapter in not ready… Sorry for inconvenience.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/sardana/checkouts/latest/doc/sou
line 61.)

Todo: This chapter in not ready… Sorry for inconvenience.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/sardana/checkouts/latest/doc/sou
line 66.)

1.1.6 Revision

Contributers T. Coutinho

Last Update Jul 11, 2018

History of modifications

Date	Revision	Description	Author
17/06/11	1.0	Initial Version	T. Coutinho

Version history

version	Changes
1.0	First official release

- genindex
- modindex
- search

Last Update Jul 11, 2018

Release 2.4.1-alpha

Python Module Index

Index