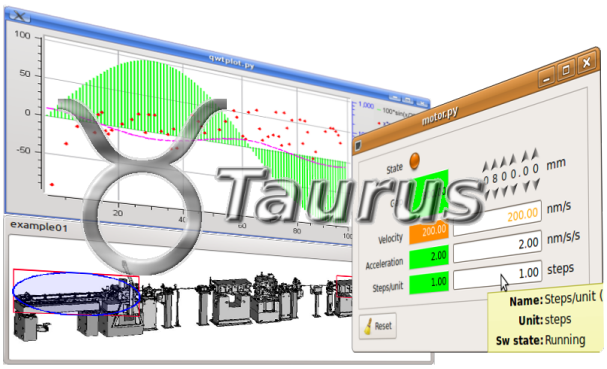
taurus Documentation

Release 4.1.2-alpha

taurus team

Aug 17, 2017

Contents



Taurus is a python framework for control and data acquisition CLIs and GUIs in scientific/industrial environments. It supports multiple control systems or data sources: Tango, EPICS, spec... New control system libraries can be integrated through plugins.

For non-programmers: Taurus allows the creation of fully-featured GUI (with forms, plots, synoptics, etc) from scratch in a few minutes using a "wizard", which can also be customized and expanded by drag-and-dropping elements around at execution time.

For programmers: Taurus gives full control to more advanced users to create and customize CLIs and GUIs programmatically using Python and a very simple and economical API which abstracts data sources as "models".

Of course, Taurus is Free Software (under LGPL). You can download it from PyPi, access its Documentation or get support from its community and the latest code from the project page.

CHAPTER 1

Projects related to Taurus

- Taurus uses PyQt for the GUIs
- Tango is supported vis PyTango
- Taurus is part of the Sardana suite

Taurus 4.1 documentation

Taurus is a free, open source, multi-platform pure Python module for creating and supporting Graphical User Interfaces for experiment control and data acquisition.

User's Guide

Introduction

Taurus was originally conceived as a library for connecting client side applications (CLIs and GUIs) to Tango device servers. Since v4.0 the Taurus core became control-system agnostic, and it supports other control systems (such as EPICS) and data sources.

Note: due to its Tango origin, this documentation will tend to use many Tango-related examples. We intend to gradually introduce more non-Tango examples

Taurus was developed within the Sardana project, but since it has been found useful for other projects not related to Sardana, it has been moved to a separate project (although both projects are kept in sync and share most of their developers).

Taurus uses the Model-View-Controller (MVC) pattern to build interfaces.

The `taurus.core` module uses plugins (known as schemes) to provide TaurusModel objects that abstract the interactions with specific sources of data and/or control objects. Some schemes are already implemented for accessing

control system libraries (the "tango" and "epics" schemes) as well as for data-processing via a Python interpreter (the "evaluation" scheme). see the *taurus core tutorial* for more information on the taurus core.

The Taurus view and controller components are typically implemented as PyQt based GUIs, but it may also consist on command line interfaces such as Sardana's spock.

The `taurus.qt` module provides a set of basic widgets (labels, LEDs, editors, forms, plots, tables, buttons, synoptics,...) that extend related Qt widgets with the capability of attaching to Taurus core models in order to display and/or change their data in pre-defined ways. For example, a TaurusPlot widget will display a curve for each attribute model to which it is attached if its value is a one-dimensional numerical array. Similarly, a TaurusForm widget will allow the user to interact with the data represented by its attached models. The actual association of a view (widget) with a model is done by providing the model name to the widget.

The following is an example on how to create a widget that allows to interact with four attributes (state, position, velocity, acceleration) of a Tango device (*motor/icepap/01*):

```
import sys
from taurus.qt.qtgui.panel import TaurusForm
from taurus.qt.qtgui.application import TaurusApplication

app = TaurusApplication(sys.argv)

attrs = [ 'state', 'position', 'velocity', 'acceleration' ]
model = [ 'motor/icepap/01/%s' % attr for attr in attrs ]

w = TaurusForm()
w.model = model
w.show()
sys.exit(app.exec_())
```

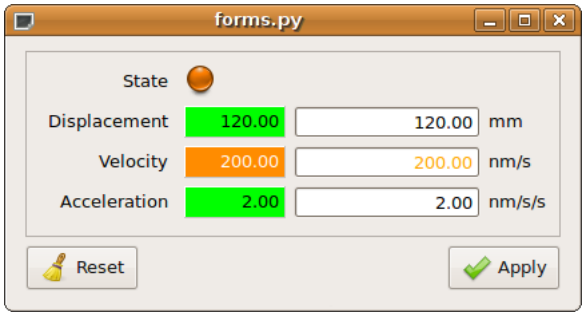


Fig. 1.1: The GUI resulting from the above code

The above example can even be achieved even without typing any code:

```
% cd taurus/qt/qtgui/panel
% taurusform motor/icepap/01/state motor/icepap/01/position motor/icepap/01/velocity
```

For many more examples, See the *Examples* page.

Taurus is a pure-python module, but it makes heavy use of PyTango, numpy, PyQt, etc. to provide good performance even when large amounts of data are involved.

Getting started

Installing

Installing with pip (platform-independent)

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

```
pip install taurus
```

You can test the installation by running:

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

Note: pip is already included in python>2.7.9

Note: some "extra" features of taurus have additional *dependencies*.

Installing from sources manually (platform-independent)

You may alternatively install from a downloaded release package:

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

   ```
   pip install .
   ```

4. test the installation:

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

Note: some "extra" features of taurus have additional *dependencies*.

Linux (Debian-based)

Since v3.0, Taurus 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-taurus
```

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

Windows

1. Install the Python(x,y) bundle (alternatively, you could install Python, PyQt, PLY, and other *dependencies* independently, but Python(x,y) will save you much worries about versions).
2. Download the latest Taurus windows binary from http://pypi.python.org/pypi/taurus
3. Run the installation executable
4. test the installation:

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

Working from Git source directly (in develop mode)

If you intend to do changes to Taurus 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 taurus from our main git repository:

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

Then, to work in develop mode, just do:

```
pip install -e ./taurus
```

Dependencies

Strictly speaking, Taurus only depends on numpy, but that will leave out most of the features normally expected of Taurus (which are considered "extras"). For example:

- Interacting with a Tango controls system requires PyTango.
- Interacting with an Epics controls system requires pyepics.
- Using the taurus Qt widgets, requires PyQt 4.x (4.8 <= v < 5). (PyQt5 support coming soon).
- The *taurus.qt.qtgui.plot* module requires PyQwt.
- The image widgets require the guiqwt library.
- The JDraw synoptics widgets require the PLY package.
- The NeXus browser widget requires PyMca5.
- The TaurusEditor widget requires spyder.
- The TaurusGui module requires lxml.

For a complete list of "extra" features and their corresponding requirements, execute the following command:

```
python -c 'import taurus; taurus.check_dependencies()'
```

How you install the required dependencies depends on your preferred installation method:

- For GNU/Linux, it is in general better to install the dependencies from your distribution repositories if available.
- For Windows users: many of these dependencies are already satisfied by installing the Python(x,y) bundle. Also, most can be installed from PyPI (e.g. using pip). For some versions, PyPI may not provide pre-built windows binaries, so pip may try to compile from sources, which takes long and may not succeed without some further work. In those cases, one may use windows binaries from other versions and/or wheel packages from the Silx_WheelHouse.
- In general, you can use pip to install dependencies for a given extra feature (if they are in PyPI or in one of your configured indexes). Use:

  ```
  pip install taurus[NAME_OF_EXTRA]
  ```

- The Conda package management system may also be used to install most of the required dependencies.
- The taurus-test Docker container provides a Docker container (based on Debian) with all the dependencies pre-installed (including Tango and Epics running environments) on which you can install taurus straight away.

User's Interface

Taurus colors

Taurus uses color codes on many of its widgets. Colors are used to represent two main things:

- the state of a taurus device
- the quality of (the reading of) an attribute.

Taurus Device state colors

Taurus Device states, as defined in `taurus.core.TaurusDevState` are represented by the following colors:

[table]xcolor

State	Background	Foreground	Preview
Ready	Green (0,255,0)	Black (0,0,0)	greenReady
NotReady	Red (255,0,0)	Black (0,0,0)	redNotReady
Undefined	Gray (128,128,128)	Black (0,0,0)	greyUndefined

Taurus Attribute Value Quality colors

The quality of an attribute measures the reliability of the current read value for that attribute. The meanings of the qualities are:

- *Invalid*: there was some problem when trying to read the attribute (the value should not be trusted)
- *Valid*: the attribute was read correctly (no reason to suspect its value validity)
- *Alarm*: the value is valid, but it exceeded its defined alarm limits
- *Warning*: like *Alarm* but for the warning limits
- *Changing*: the attribute was read correctly but it is being changed at the time of reading (so its value is likely to differ if re-read)

Taurus Attribute value qualities are represented by the following colors:

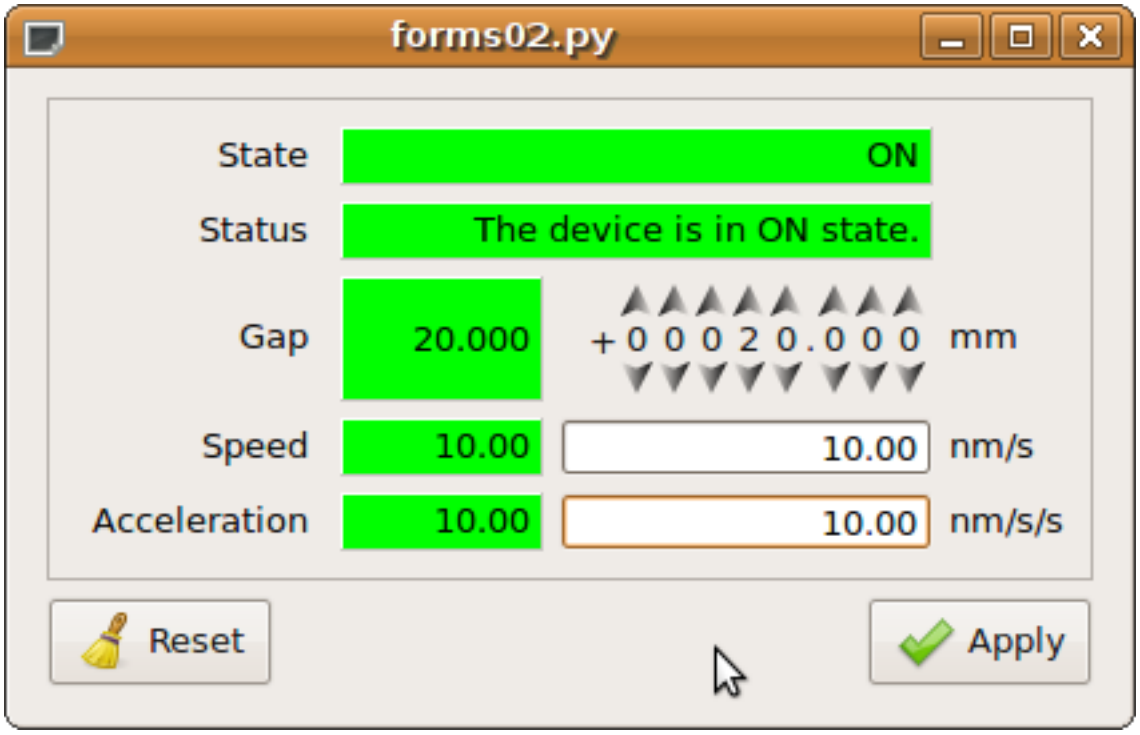
Tango-specific Device state colors

Tango Device states are richer than the generic ones. The following is a table of the colors used to represent Tango-specific device states handled by the `taurus.core.tango` scheme:

TaurusForm User's Interface

Contents

The `TaurusForm` widget is the standard way for Taurus applications to present a set of attribute and device readings and controls widgets with a form- like layout.



For each item (attribute or device) represented in a TaurusForm, the following elements will be shown (vertically aligned with those of the other items along 5 columns):

1. *label text*. It shows the name or other identification of the item.
2. *read widget*. It shows the current value of the item. If the information cannot be shown in a compact way, it may be a button that launches a separate dialog which provides detailed information. The read widget may provide information on both the read value and the quality status of the attribute, typically using the *Taurus color convention*. The read widget will extend over the second and third columns for those items that are not writable.
3. *write widget* (only shown if the item allows writting). A widget for modifying the value of the item. If it cannot be done in a compact way, it may be a button that launches a separate dialog which provides detailed information.
4. *units text*. It shows the units associated to the item (it is not shown if no units are defined for this item).
5. *extra widget*. An extra space that may be used by some custom widgets (only shown if used).

The precise widgets that are used by default for each item are determined by the type of attribute / class of device as well as by a custom mapping that is set at Taurus installation time.

TaurusForm as a stand-alone application

You may also use TaurusForm as a stand-alone application for controlling some attributes or devices from the control system. You can launch the stand-alone TaurusForm with the following command:

```
taurusform [options] [<model_list>]
```

Run the following command for more details:

```
taurusform --help
```

The model list is optional and is a space-separated list of models for TaurusForm. Valid models are: attribute names, device names or alias. See `TaurusForm` API for more information about valid models.

The widgets used for different types of attributes and devices

By default, TaurusForm tries to use the most appropriate Taurus widget for representing its attributes and/or widgets.

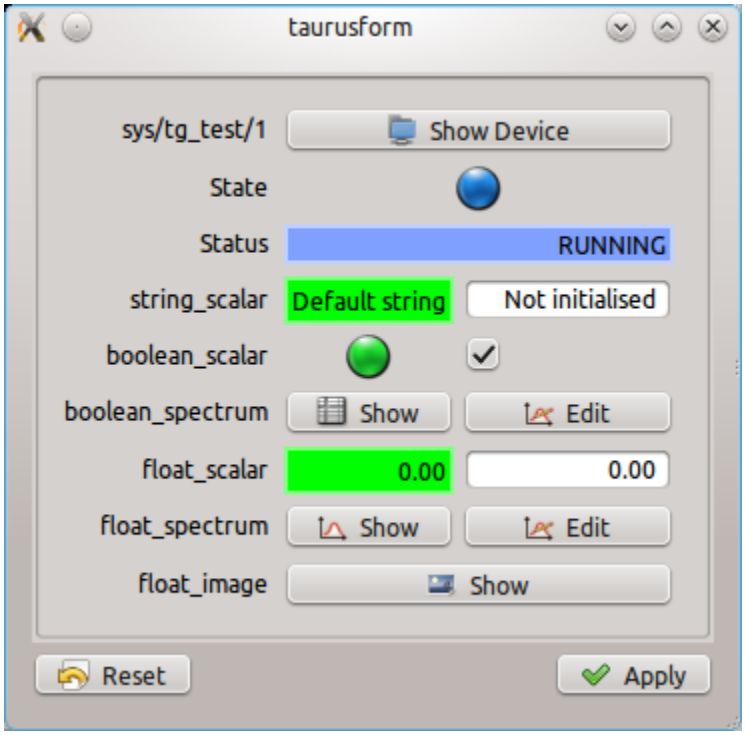


Fig. 1.2: A taurusform created with the following command *taurusform sys/tg_test/1 sys/tg_test/1/state sys/tg_test/1/status sys/tg_test/1/string_scalar sys/tg_test/1/boolean_scalar sys/tg_test/1/boolean_spectrum sys/tg_test/1/float_scalar sys/tg_test/1/float_spectrum sys/tg_test/1/float_image*

For the attributes, TaurusForm checks the type of attribute (whether it is a scalar or an array, whether it is a number or a string or a boolean, whether it is writable or read-only, etc.). For certain attributes, more than one widget may be adequate, and the form allows the user to switch between them (See the *Changing the contents of a form* section).

For Tango devices, the Tango Class of the device is searched in the *T_FORM_CUSTOM_WIDGET_MAP* map defined in *Taurus custom settings* and the given widget is used if there is a match. Otherwise, the default device representation is used, which shows a button that launches an `AttributeForm` showing *all* the attributes for that device.

As an example, Sardana makes heavy use of *T_FORM_CUSTOM_WIDGET_MAP* in order to display specific widgets for its motor and channel devices.

Changing the contents of a form

If the application using the form allows it, you can modify both *what* is shown and *how* it is shown in the form.

- You can alter *what* is shown in two ways:
  - right-clicking in the form and selecting the *Modify contents* option in the context menu. A *TaurusModelChooser* widget will let you modify the list of taurus models being represented by the form.
  - Dropping models that you drag from another taurus widget (typically another TaurusForm, or a *TaurusModelChooser*). See the *Drag and Drop support* section for more details.
- Regarding *how* it is shown, you can change the following (provided that user modifications are allowed for this form):
  - which widget is used for displaying the read value or the write value of a given item. This is done by right-clicking on the label of the item and selecting the *change Read Widget* (or *change write widget*) option.
  - Whether to use *compact mode* or not. See *Compact mode* section for more details
  - The text shown in the label widget for a value can be customised via the 'Change Label' option in the label's context menu. It can also be changed for all values in a form with the 'Change Labels' option of the form's context menu.

Tip: You can use the *Modify contents* option to re-order what is shown.

Drag and Drop support

TaurusForm supports drag&drop of models for its items. Essentially each item represented in a form is associated to a *Taurus model*. By dragging the label of a given item in a form, what happens behind the scenes is that the *model* is copied. Many Taurus widgets can be instructed to accept drops of models (e.g. `TaurusPlot`, `TaurusTrend`, `TaurusForm`,...) and alter their models according to the new model passed.

When a TaurusForm receives one or more models from a drop, it appends them to its current list of displayed models and shows them.

Tip: If you accidentally dropped a model and want to remove the new item, just use the *Modify contents* option from the form's context menu.

Compact Mode

When in compact mode, a value in a form is shown with only one column for both the read and write widget. Normally the read widget is shown, and only when the user triggers the edition mode, the write widget is shown. The edit triggers are, tipically:

- the F2 key
- Double-clicking on the read widget

The edition mode is left when the write widget loses focus, or the changes have been applied. Also, in many cases, when the "ESC" key is pressed.

You can enable/disable the compact mode for a value by right-clicking on its label and selecting 'compact mode'. You can also set the compact mode for all values in the form via the context menu of the form.

Writing to attributes

Taurus attributes can be read-only or allow writing. Those attributes that are writable have two values: the *read value* and the *write value*. The read value is displayed by the *read widget* in the second column of the TaurusForm (just right of the label). The *write widget*, in the third column of the TaurusForm allows you to modify the write value.

Note that the read value and the write value of an attribute are not necessarily equal. The write value is a "set point", or "desired value" while the read value gives the actual value as read by the control system (for example, in a power supply device, the read value of its voltage attribute oscillate around the write value due to ripple). Also note that the units associated to the read and write values may not be the same (In Tango, they are internally the same, but other schemes may have ways of defining them independently)

Since writing wrong values may be dangerous for some equipment, the default behaviour of write widgets is not to apply new values directly as you type the value or move a dial. Instead, the corresponding label of the item becomes highlighted with a blue frame, indicating that the write value has been changed (we say that the item has *pending operations*) and that these changes can be applied. Some write widgets give extra feedback apart from that of the label.

When a widget has pending operations, you can check exactly what is the pending operation by consulting its tooltip.

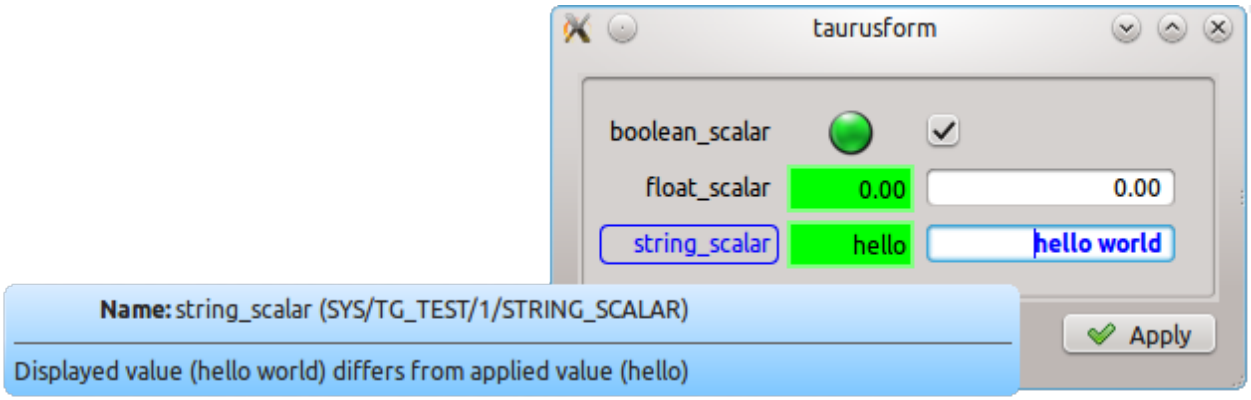


Fig. 1.3: Example of pending operation notification: the write value of the array_scalar attribute has been changed but it is not yet applied. The label for this item shows a blue frame, the write widget shows bold blue text and the tooltip gives extra information.

Pending operations in a form can be applied either individually for each item or all at once:

- The way for applying changes for a single item may depend on the write widget involved, but typically (it is the case for all default widgets) it is done by pressing the *Enter* key on the write widget (see also the *Forced apply* section).
- For applying all pending operations of a whole form at once, you can click on the *Apply* button of the form (if available).

If the form provides buttons, the *Reset* button will discard all pending operations.

Forced apply

By default, if a write value has not changed, there will be no pending operations and therefore the value cannot be re-applied. Some times, however, it may be interesting to force another write of the same value (e.g. a certain hardware device needs an extra "push" to reach the desired value). This can be done by pressing *CTRL+Enter* (instead of just *Enter*) on the write widget.

External changes of displayed attributes

When the read value of an attribute is updated in the control system, the read widget for that attribute will reflect the new value (depending on the configuration of the control system, a certain refresh period may be waited).

When the write value of an attribute is updated in the control system (a certain attribute may be accessed simultaneously from different client aplications or even from different parts of the same application), the item representing that attribute will be notified, but the value displayed by the write widget will not be changed (instead, the item will show that there is a pending operation). In this way, concurrent editions will not interfere with your own editions, but at the same time you will be aware of them.

Tip: Remember that you can check the cause of a pending operation by consulting the write widget tooltip.

TaurusModelChooser User's Interface

The `TaurusModelChooser` is a tree based widget used by Taurus applications for prompting the user to choose one or more attribute/device names of the control system.

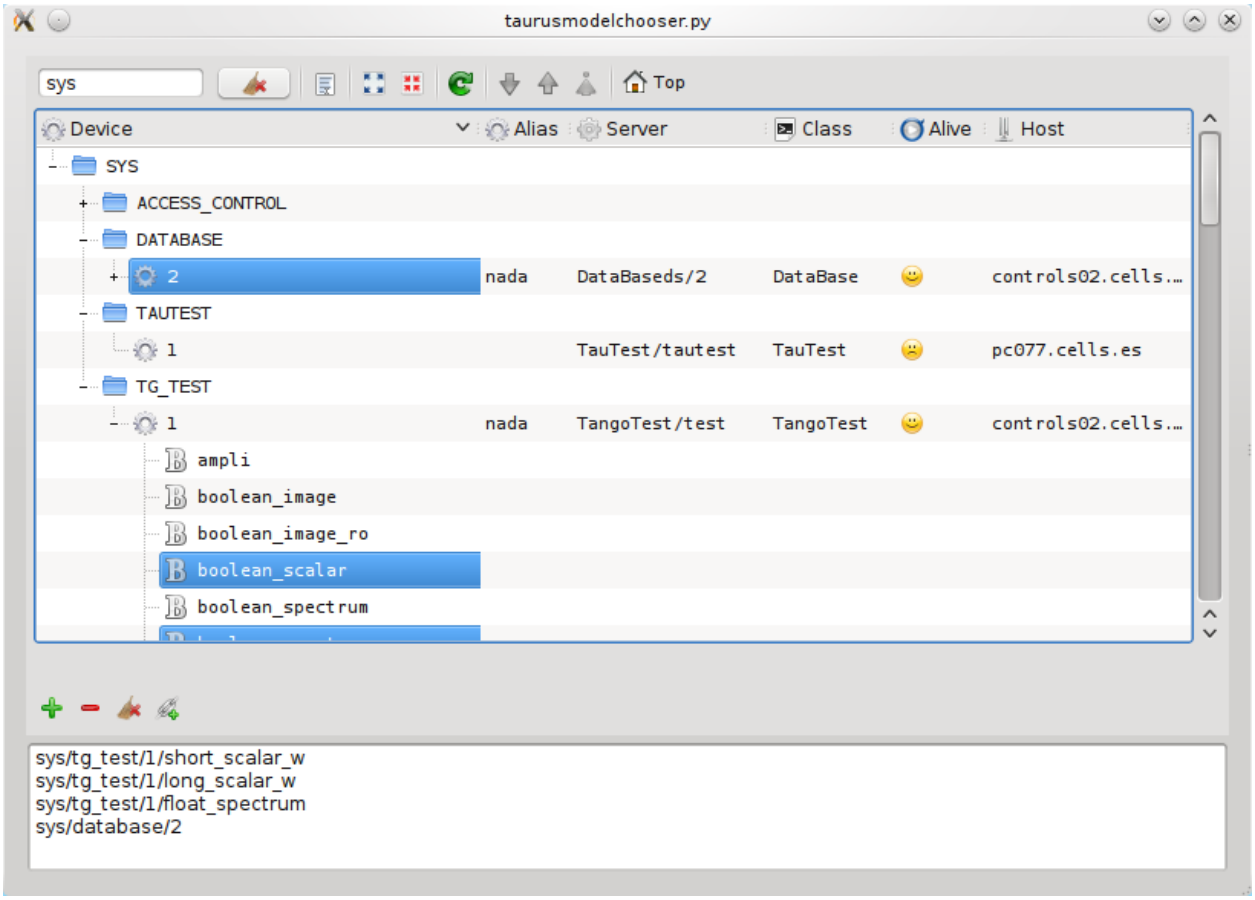
To select the attributes using `TaurusModelChooser`, you typically do the following:

1. Browse the tree to locate the devices and/or attributes you are interested in.
2. Now select one or more device/attributes (tip: you can use the CTRL key for multiple selection)
3. Add them to the *Chosen models* list (the bottom part of the widget) by using the *Add* button (the "+" icon).
4. Repeat previous steps if you want to add other models or use the *Remove* button (the "-" icon) to remove models from the *Chosen models* list.
5. Click the *Update Models* button to apply the changes when satisfied.

Important: The tree can be quite dense. You can filter it by typing part of the device name in the *Filter* box (#4 in the figure).

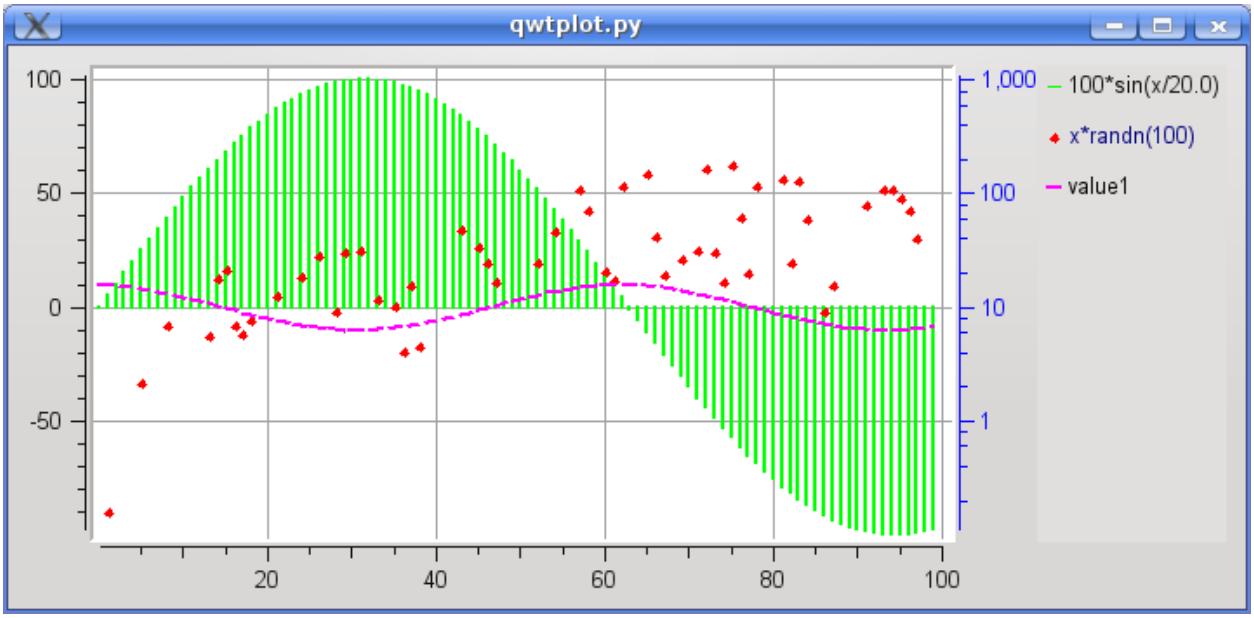
TaurusPlot User's Interface

Contents


taurusmodelchooser.py
sys
Top
Device
Alias
Server
Class
Alive
Host
SYS
ACCESS_CONTROL
DATABASE
2
nada
DataBaseds/2
DataBase
controls02.cells.…
TAUTEST
1
TauTest/tautest
TauTest
pc077.cells.es
TG_TEST
1
nada
TangoTest/test
TangoTest
controls02.cells.…
ampli
boolean_image
boolean_image_ro
boolean_scalar
boolean_spectrum
sys/tg_test/1/short_scalar_w
sys/tg_test/1/long_scalar_w
sys/tg_test/1/float_spectrum
sys/database/2

The standard way for Taurus applications to show one-dimensional data is by using a `TaurusPlot` widget.

`TaurusPlot` is shown as an area with X and Y axes where curves (data sets) are plotted. It may also show a legend.



But the `TaurusPlot` does a lot more than just showing a plot. It allows the user to interact with the plot in many ways as described below.

Note: The features described here are available *by default* in all TaurusPlot widgets, but certain GUIs may choose to disable some of these features.

TaurusPlot as a Stand-alone application

You may also use `TaurusPlot` as a stand-alone application for displaying attributes from the control system or for plotting a function. You can launch the stand-alone `TaurusPlot` with the following command:

```
taurusplot [options] [<model_list>]
```

Run the following command for more details:

```
taurusplot --help
```

The <model_list> is a space-separated list of models for `TaurusPlot`. Valid models are: SPECTRUM attribute names or alias, and *Xattrname*|*Yattrname* constructions for indicating X-Y scatter plots. See `TaurusPlot` API for more information about valid models

Working with two Y scales

A `TaurusPlot` has a X axis and one or two Y axes (left and right, also called Y1 and Y2 and drawn in black and blue, respectively). Multiple curves can be displayed simultaneously and each one will be associated to either Y1 or Y2.

By default, `TaurusPlot` only shows a legend when more than one curve is displayed (this behavior can be overridden using the *TaurusPlot context menu*).

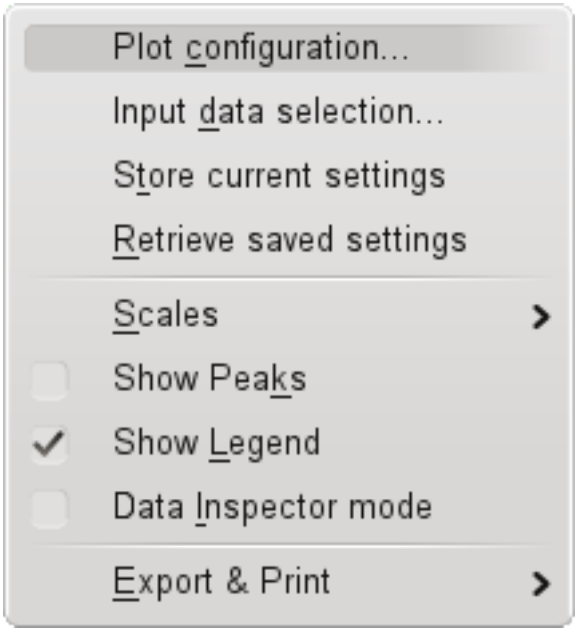
Curves associated to the Y1 axis show their title in the legend using black text, while those associated to Y2 use blue text.

You can change the axis to which a curve is associated by clicking on its title in the legend. There are three states: associated to Y1 (black legend text), associated to Y2 (blue legend text), and hidden (the curve is not displayed and its title in the legend becomes gray).

You can also change the axis to which a curve is associated using the *Plot Configuration dialog*.

TaurusPlot context menu

Most of the options of a `TaurusPlot` can be managed from a context menu that is shown when right clicking on the plot area:



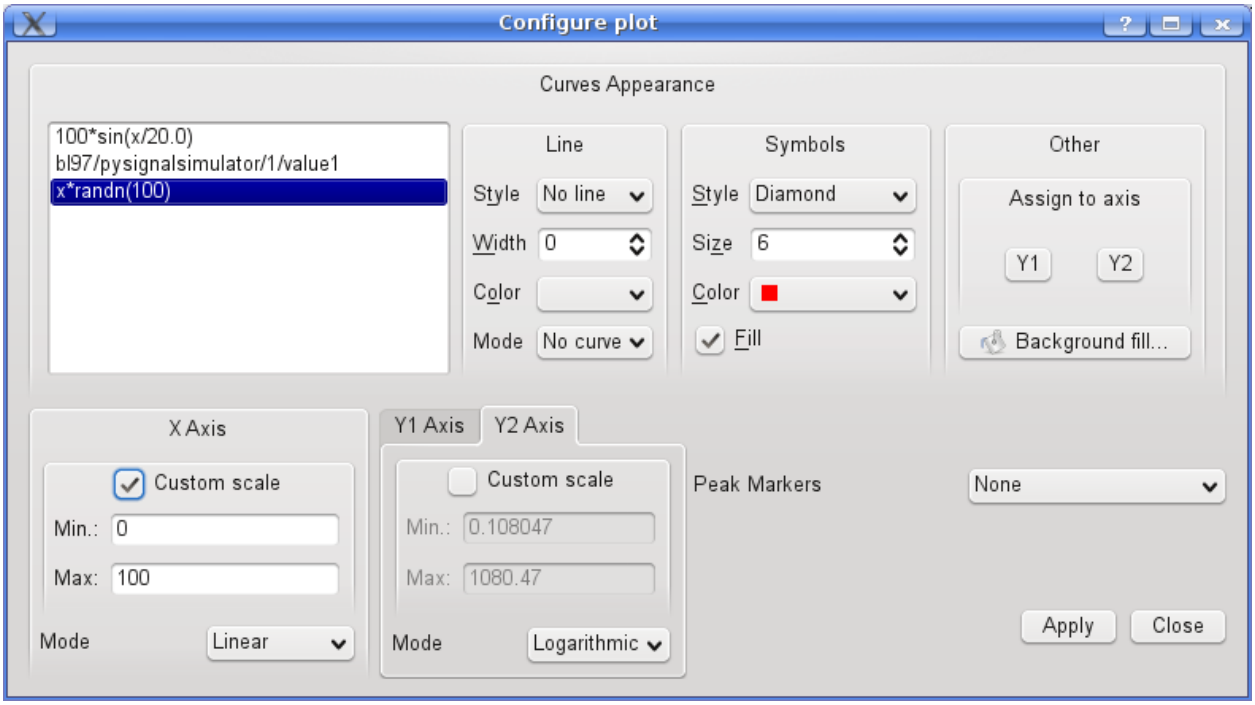
Right-clicking on an axis will show a context menu with the specific options for that axis.

Zooming and panning

There are several ways of changing the scales interactively:

- The plots can be zoomed in and out by using the mouse wheel over the plot area.
- Zooming over selected areas is also possible by dragging a selection rectangle with the left button of the mouse. Right-clicking will go back to the previous selection. Note that, if both Y1 and Y2 axes are enabled, the zooming region will only affect the curves attached to one of the axes (the region selector will be black when the active axis is Y1 and blue when it is Y2). It is possible to change which Y axis is active for zooming by either using the 'Z' key, or via *TaurusPlot context menu*
- Panning (i.e. translating without scaling) is done by holding the CTRL key down while dragging with the left button of the mouse.
- The ESC key resets the zooms and returns to auto-scale mode.
- Finally, all details about the scales are accessible at the *Plot Configuration dialog*.

Plot Configuration dialog



This dialog can be accessed from the *TaurusPlot context menu*.

On its top section you can customize the look of the displayed data (line type, symbols used, colors, thicknesses, associated axis,...). Any changes will be automatically applied to all the curves selected from the list on the left.

The curve titles can be changed by editing them directly in the list (one by one) as well as by selecting some of them and using the *Curve Title(s)...* button.

On the bottom part you can control the scales for all axes (X, Y1 and Y2):

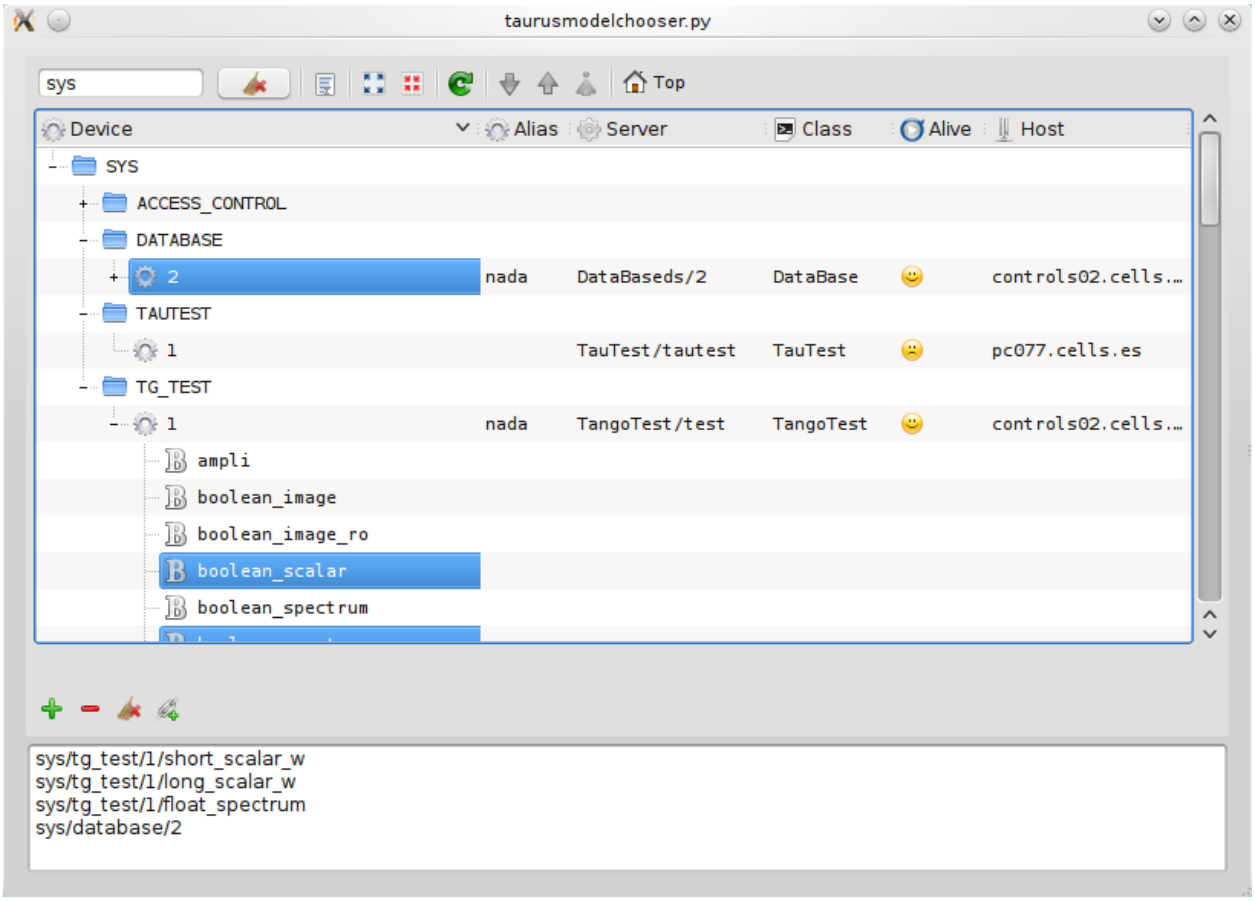
- custom scales. If the "Custom scale" check box is enabled for an axis, the range defined by the min and max values will remain fixed. If it is disabled, the axis will auto scale to accommodate all the data.
- Axis type: You can choose either Linear or Logarithmic scales. Note: non-positive points will be silently ignored when in logarithmic mode.

Choosing what is plotted

When `TaurusPlot` is used in a GUI, it is likely that some data is already plotted on it. But, say that you want to compare it against some function, or against data stored in a file, or even against some attribute of the control system...

...then the *Input data selection* option from the *TaurusPlot context menu* is for you!

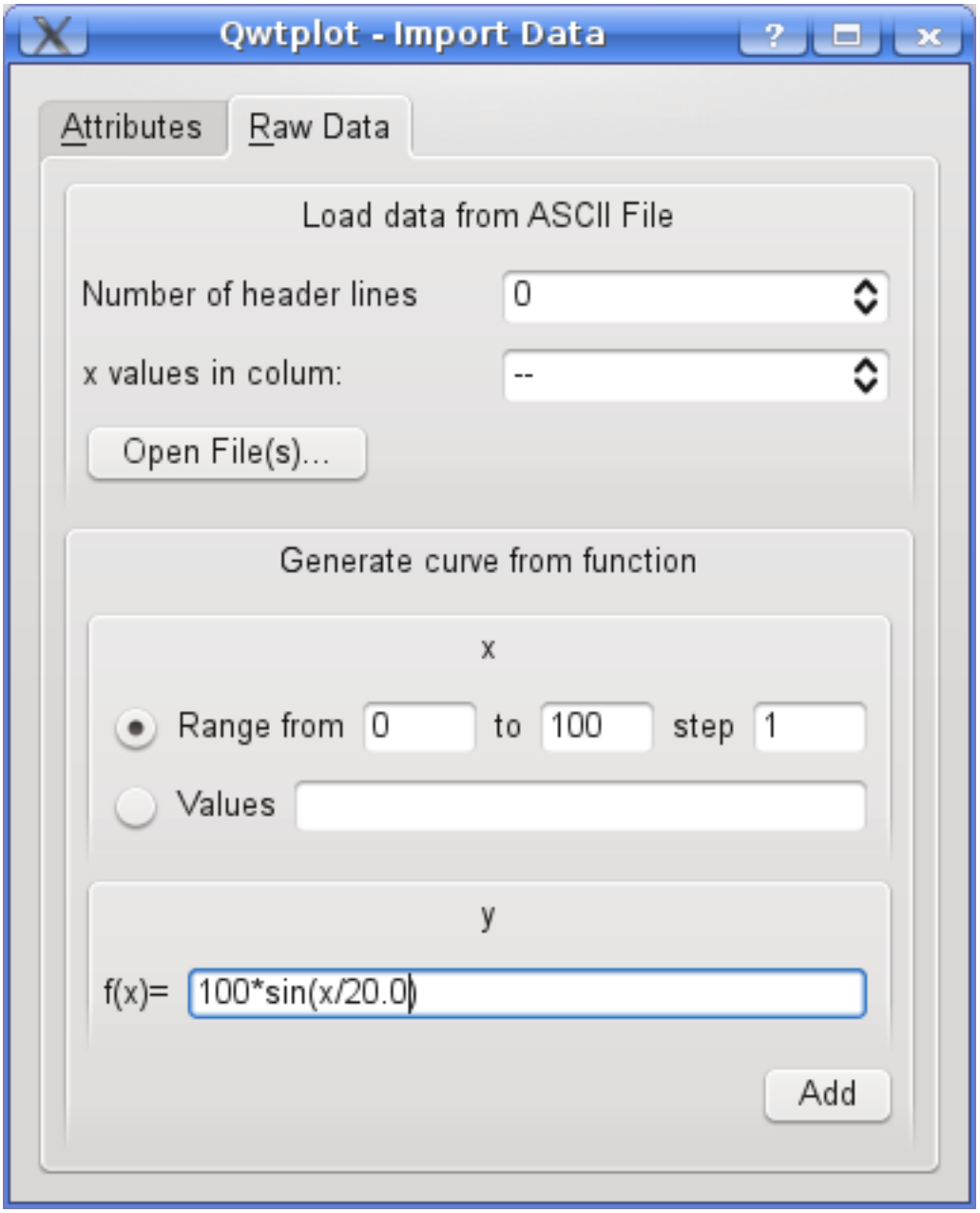
From the *Attributes* tab of the import data dialog, you can choose which Tango attributes are plotted, thanks to a *TaurusModelChooser* widget.



If the data you want to plot is not an attribute, go to the *Raw Data* tab, where you can add data sets to the plot from the following sources:

- You can read it from a file (e.g., one that was created with the *export to ASCII option* from the *TaurusPlot context menu*)
- You can add it by entering a mathematical formula. `TaurusPlot` will recognize many common functions (it evaluates the formula using a subset of numpy expressions). You can use *x* as a dependent variable, which you can set as regular steps or using an arbitrary expression.

Note that there is actually no way to remove RawData curve from the GUI.


Qwtplot - Import Data
Attributes
Raw Data
Load data from ASCII File
Number of header lines
0
x values in colum:
--
Open File(s)...
Generate curve from function
x
Range from
0
to
100
step
1
Values
y
f(x)=
100*sin(x/20.0)
Add

Storing and recovering current configuration

Once you have customized the way the plot looks (see the *Plot Configuration dialog* section), you may want to save the settings for later use. This can be done using the *Save current settings* option from the *TaurusPlot context menu*.

This will save which curves should be plotted and how they should look.

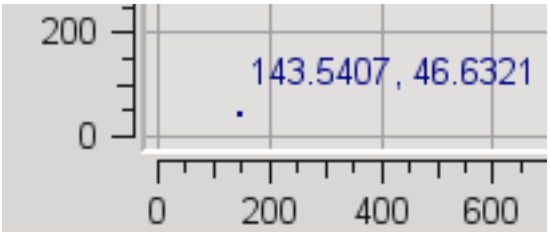
You can restore those settings at any point by using the *retrieve current settings* option from the *TaurusPlot context menu*.

Important: *note that if you saved the settings of a plot which displayed an attribute from the control system, the actual values shown when restoring the settings will be updated with the attribute value*

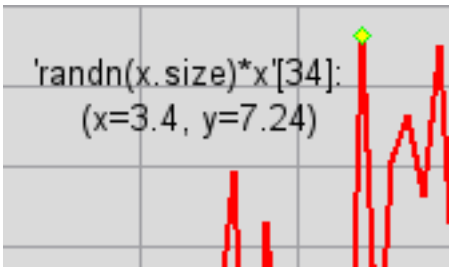
Obtaining information about the plotted values

Sometimes you want to know more about the values being plotted. The following features can be useful:

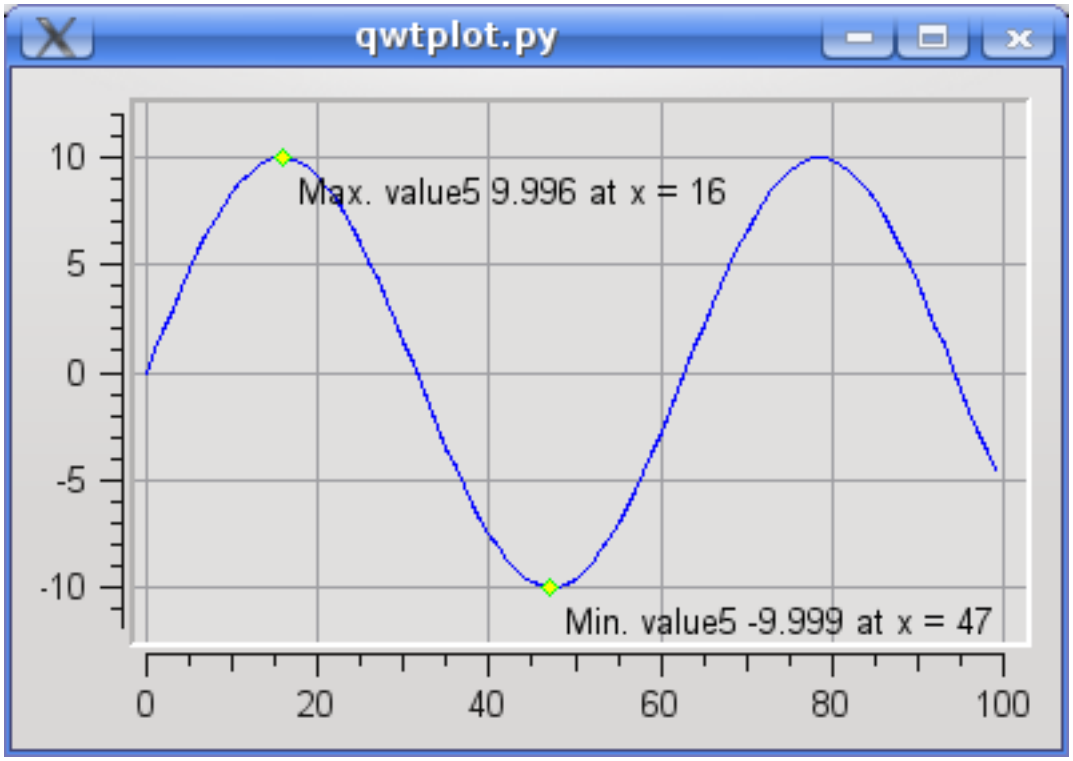
- Obtaining coordinates of the plot area: you can get the coordinates (in the X-Y1 system) for any arbitrary point of the plot by simply clicking on it.



- Data Inspector: you can inspect the value of any given data point by entering in *Inspector mode*. You can toggle this mode this by pressing the "I" key or via the *TaurusPlot context menu* (the cursor over the plot will change from the usual cross to something like this:  , depending on the system). Once in data inspector mode, you can click on a data point, which will be marked and some information about it will be displayed:



- Peak locator: `TaurusPlot` can locate and put a mark at the maximum and/or minimum points in the plotted data. You switch this option on and off using the *Show min* and *Show max* option from the *TaurusPlot context menu* or use from the *Peak Markers* option in the *Plot Configuration dialog*



- Finally, if you want to see a text list of all the data being plotted, you can also do it from the *Export to ASCII Dialog* from the *TaurusPlot context menu*

Exporting and printing the data

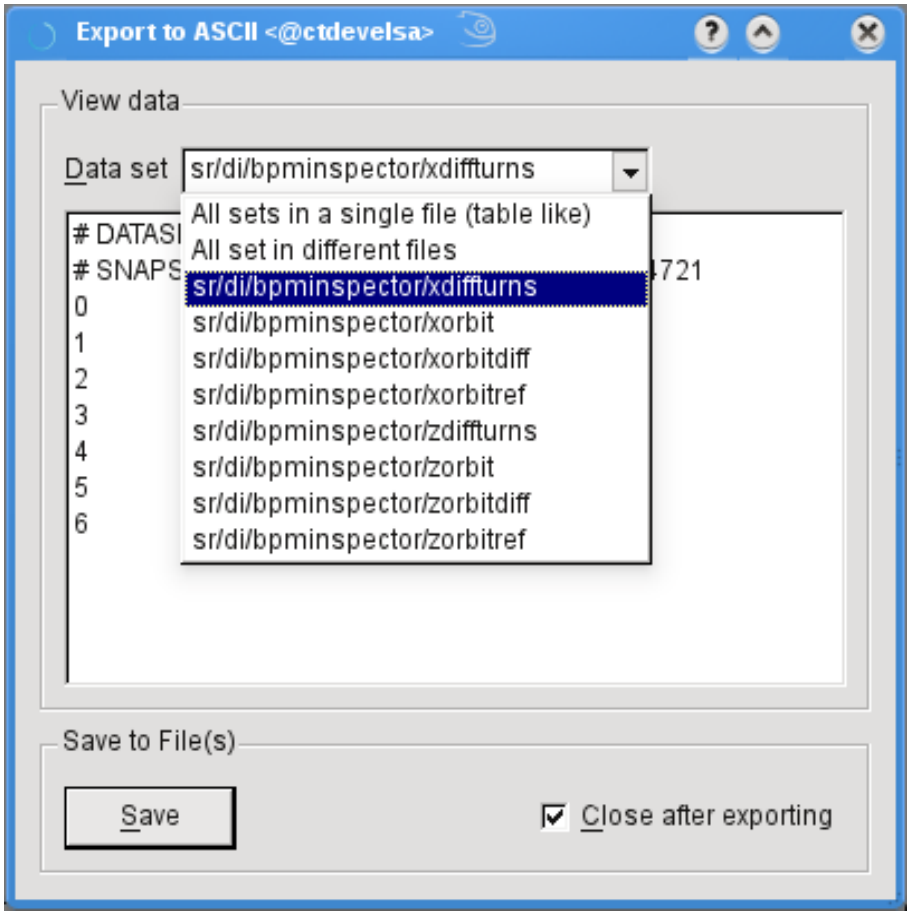
You want a figure for a logbook?

Or you want to store the plotted values in a file?

Then you will like the Export capabilities of TaurusPlot.

You can access them from the *TaurusPlot context menu* and for the moment they allow you to:

- Print the plot.
- Create a PDF from the plot. Note that it is not a simple screenshot, but a proper vectorial PDF (this means you can zoom in without loosing resolution).
- **Export the data values to ASCII (you can edit before saving!):** This will save the data in plain ASCII text format. If the plot is showing more than one curve, you can choose between:
  - saving a single curve in two-column mode
  - saving all the curves in a single table-like file (this is only possible if the X data is the same for all curves)
  - saving all curves to separate files in one go.



(note that you can also use this dialog to inspect the data and to copy it to the clipboard)

Customizing the titles of the curves

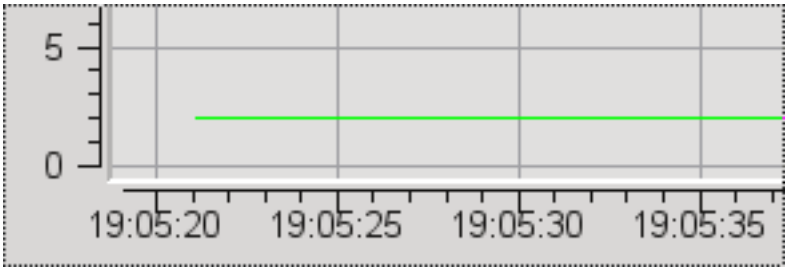
The titles of the curves (which are displayed in the legend) can be customised in several ways:

- Through the *Change Curves Titles...* option in the *TaurusPlot context menu*
- Through the *Plot Configuration dialog* (either editing the name individually, or selecting several curves and clicking on the *Curve Title(s)...* button)

Date/time support

In some applications, the values of the X axis are interpreted as date/time values (this is set either by the GUI using TaurusPlot, or by the *-xt* parameter if TaurusPlot is being launched from the command line).

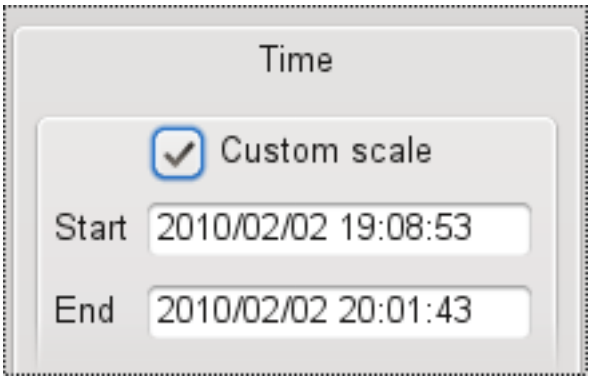
In this mode, the values of the abscissas must be valid epoch numbers, i.e. seconds since the "beginning of times" (UNIX t=0 : 1970-01-01 01:00:00). TaurusPlot will interpret such values and display well-formatted dates and times instead of the epoch number:



When working in time mode, the X axis in the *Plot Configuration dialog* changes to "Time", and the Min/Max entries change to Start/End, where you can enter a date (and, optionally a time).

Here are some tips for entering valid date/time values:

- **For absolute date/times:**
  - The date can be written in various formats. ISO format is recommended (e.g. "1917-10-25"), although others like, e.g. "25/10/1917" are also accepted.
  - The time is given in 24 hours format (e.g. "21:45") and may optionnaly include seconds if given (e.g. "21:45:01")
  - Date is mandatory while time is optional. If time is given, it must be separated from the date with a single space (e.g. "1917-10-25 21:45:01")



- **For relative date/times.**
  - You can specify the date/time relative to the current time. Do this by using a "+" or "-" symbol, followed by a number and a time unit (valid time units are "s", "m", "h", "d", "w" and "y"). For example, the following are valid relative times: "-1d", "+3w", "- 3.6e3 s". Note that the units are case-sensitive (e.g., "-1D" is not valid)
  - also, the keyword "now" (case-insensitive) can be used as a synonym of "+0s".

TaurusCurveDialog

Taurus also offers an alternative widget for plotting one-dimensional data: `TaurusCurveDialog`. This widget is based on the guiqwt library and is currently less developed and tested than :class'TaurusPlot'.

The `TaurusCurveDialog` widget can be launched as a stand-alone application with the following command:

```
tauruscurve [options] [<model_list>]
```

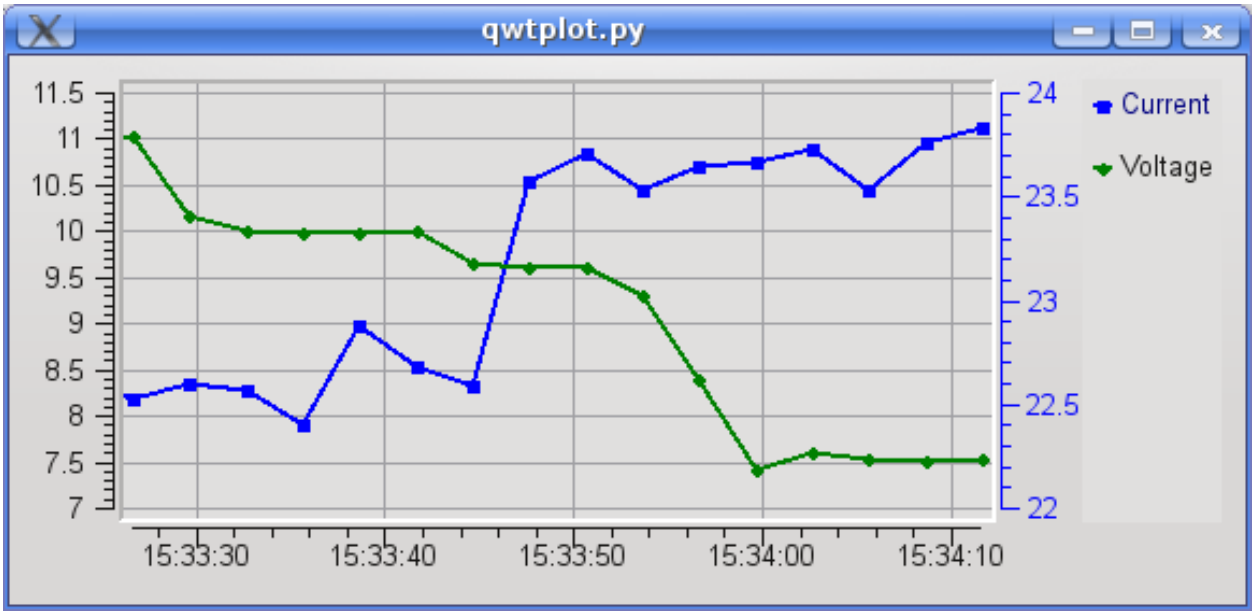
Run the following command for more details:

```
tauruscurve --help
```

TaurusTrend User's Interface

Contents

The `TaurusTrend` widget is the standard way for Taurus applications to graphically show the evolution of one or more **scalar** attributes from the control system.



The value of the scalar attributes is plotted in a Y axis with incrementing X value each time an event from this attribute is received. The X values can either be an index for the event number or a timestamp value. See *Timestamp VS "event number" mode* for more details.

TaurusTrend is based on `TaurusPlot`, and shares all its features. Please check the *TaurusPlot User's Interface Guide* for learning about them. In the following, only those features that are exclusive of TaurusTrend are discussed.

TaurusTrend as a stand-alone application

You may also use TaurusTrend as a stand-alone application for showing trends of attributes from the control system. You can launch the stand-alone TaurusTrend with the following command:

```
taurustrend [options] [<model_list>]
```

Run the following command for more details:

```
taurustrend --help
```

The model list is optional and is a space-separated list of models for TaurusTrend. Valid models are: SCALAR or SPECTRUM attribute names or alias (spectra will be treated as collections of scalars). See `TaurusTrend` API for more information about valid models

Timestamp VS "event number" mode

When TaurusTrend receives an event from an attribute, it plots the attribute value against either:

- the sequential order of arrival of the event (i.e., the *event number*), or
- the *timestamp* associated with this value. See *Date/time support* for more information.

When TaurusTrend is part of a GUI, this behavior is set by that GUI. If TaurusTrend is used as a stand-alone application, the behavior is determined by the *-x* parameter (see the *TaurusTrend as a stand-alone application* section).

The timestamp mode is in general the most used one, but it is important to be aware that the timestamp will only be as accurate as the (distributed) control system allows (not all attributes will, typically, be controlled by a centralised high-accuracy clock). See the *Known limitations* section for more details.

The Fixed-range X scale

When working with trends, you may want to see the latest changes only. For example, you may be interested in seeing the changes occurred within the last 5 minutes (assuming you are working in time scale) or only the latest 20 values (if you were working in "event number" mode).

You can switch the *Fixed-range X scale* mode on and off from either:

- The *Context Menu* (under the *Scales* submenu)
- The *Plot Configuration dialog* (under the *X axis/Time* section). Here you can choose from suggested values or simply type a range:
  - If working in Time mode, the range consists of a number followed by one of the following time units: "s", "m", "h", "d", "w" and "y". See *Date/time support* for more information.

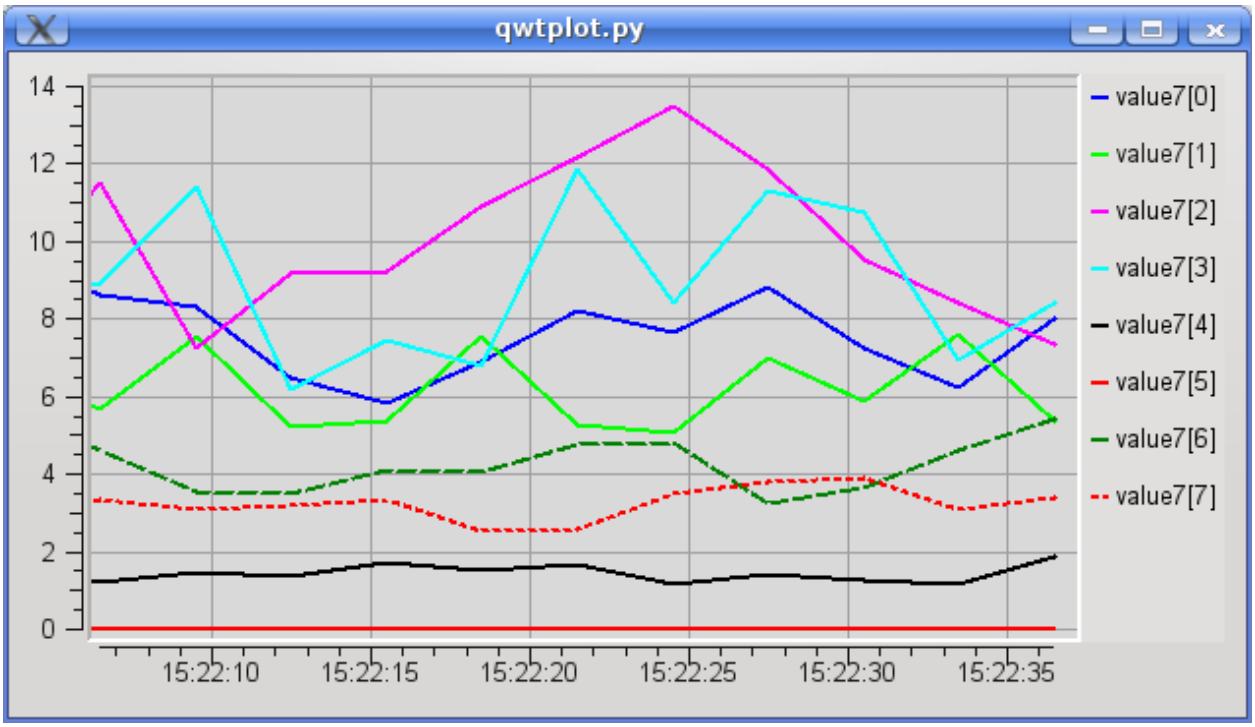
Note that if you switch the Fixed-range mode from off to on, the current range of the X axis (Max-Min or End-Start) will be used.

Using a spectrum as a collection of scalar values

TaurusTrend plots the evolution of *scalar* values. However, in many occasions the SPECTRUM attributes are (ab)used in the control system as a way of packing a set of scalar values together.

For this reason, TaurusTrend allows you to plot trends of SPECTRUM attributes assuming that each item in the spectrum is an independent scalar number (e.g., a SPECTRUM of length 8 will be plotted as 8 separate curves).

Obviously, all curves related to a single SPECTRUM attribute will be updated simultaneously, since events are received per attribute.



Accessing archived values

TaurusTrend supports connecting to a Database of archived values for the parameters being plotted, which allows to access values previous to the moment the GUI trend was created.

To improve performance, this feature is not enabled by default but it can be enabled in the *Context Menu*. If launching the TaurusTrend as a stand-alone application, you can also enable this at launch time by passing the *-a* option (see the *TaurusTrend as a stand-alone application* section).

Important: retrieving archived values can be slow. Even if archived values are accessed only when needed (i.e., when the time range being displayed requires it), you may experience performance issues if you change the scale to include large time range. For this reason a warning dialog may be shown when you try to access too much archived data.

Polling buffer

The Tango control system provides a buffer of the last values read for an attribute. This can be useful sometimes for performance reasons. The Polling Buffer support in TaurusTrend can be enabled in the *Context Menu*.

Forced read of attributes

Some attributes of the control system may be set to only send events when their value changes. In the case of TaurusTrend, this behaviour causes that attributes that do not change often, do not get new points in the trend and thus may seem to have been "lost". To avoid this issue, you can instruct the TaurusTrend to periodically force re-read the attributes being displayed. You can enable this feature (and set the re-read period) from the *Context Menu*.

If launching the TaurusTrend as a stand-alone application, you can also enable this at launch time by passing the *-r* option (see the *TaurusTrend as a stand-alone application* section).

Keep in mind the following:

- When enabling this feature, TaurusTrend will ignore all other events (to guarantee that the refresh occurs at *and only at* the desired periodic times).
- Setting very frequent re-reads may impact your control system and GUI performance. Specially if you are plotting several attributes. **So, do not use this feature unless it is really needed**

Known limitations

Limited timing precision when comparing two attributes

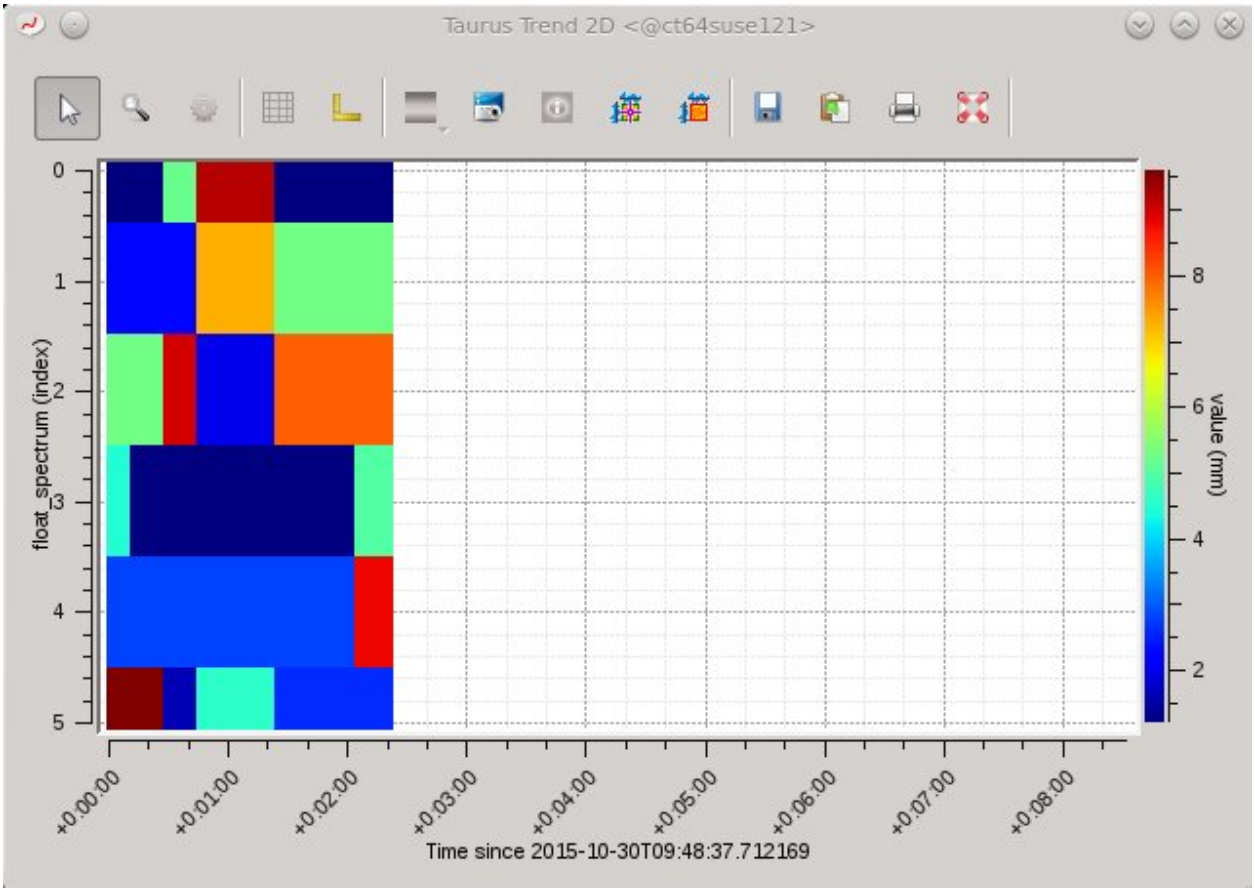
If two different attributes are plotted in the same trend, their times may appear slightly out-of-synch. For example, two parameters that are known to vary simultaneously may be apparently 0.5 seconds apart.

This occurs because TaurusTrend relies on the control system timestamps for assigning the time values. Therefore, its precision is limited by that of the control system for the given parameters.

Note that timestamps for an attribute may be assigned using the internal clock of the machine hosting the device server. Therefore, attributes managed by different machines may only be compared to the extent of the synchronization of the corresponding internal clocks. In such a case, keeping all machines synchronized is highly recommended (e.g., using NTP, accuracy should not be worse than 0.1s).

Spectrograms (2D trends)

The `TaurusTrend2DDialog` widget is the standard way for Taurus applications to graphically show the evolution of a one-dimensional attribute. The representation is done as a *spectrogram*: the time (or event number) is represented in the X axis while values of the array are color-coded along the Y axis.



Several tools from the standard guiqwt toolkit for images are available, and zooming and panning are possible using the standard mechanisms of guiqwt.

The `TaurusTrend2DDialog` widget can be launched as a stand-alone application with the following command:

```
taurustrend2d <array_attribute_name>
```

Run the following command for more details:

```
taurustrend2d --help
```

Image's interface

Contents

– *TaurusImageDialog as a stand-alone application*

The `TaurusImageDialog` widget is a Taurus Widget for displaying **image** attributes from the control system. A contour plot is created from the values of the image attribute.

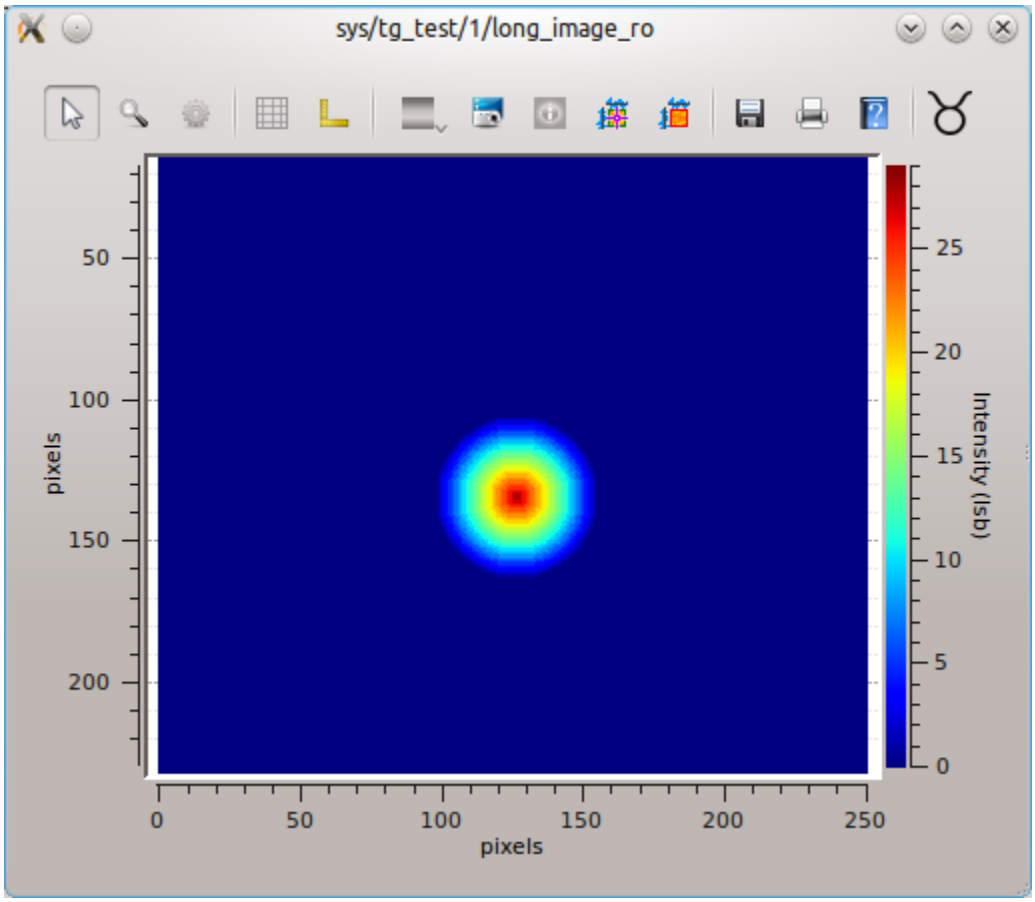


Fig. 1.4: A TaurusImage widget displaying the *sys/tg_test/1/long_image_ro* attribute

Many tools may be available, such as:

- zooming and panning
- X/Y cross sections
- Annotation tools (for creating labels, shapes,...)
- add/delete images
- ...

Note: The `TaurusImageDialog` widget is provided by the `taurus.qt.qtgui.extra_guiqwt` module which depends on the `guiqwt` module being installed. If guiqwt is not installed, the image widget will not be available.

TaurusImageDialog as a stand-alone application

You may also use TaurusImageDialog as a stand-alone application for showing image attributes from the control system. You can launch the stand-alone Taurus Image with the following command:

```
taurusimage [options] <model>
```

Options:

```
-h, --help            show this help message and exit
--demo                show a demo of the widget
--version             show program's version number and exit

Taurus Options:
  Basic options present in any taurus application

  --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 is the name of a taurus image attribute

Array Editor

The *ArrayEditor* is a widget for graphically editing a spectrum.

It consists of two *plots* and a *control point area*. The plot on top shows the current and modified spectrum. The other plot shows the difference between the current and the modified spectrum. The Control point area shows details on the control points that have been defined.

The spectrum can be modified by adding control points and moving them along the vertical axis, either by setting the value in the controls area or by dragging them in the plots.

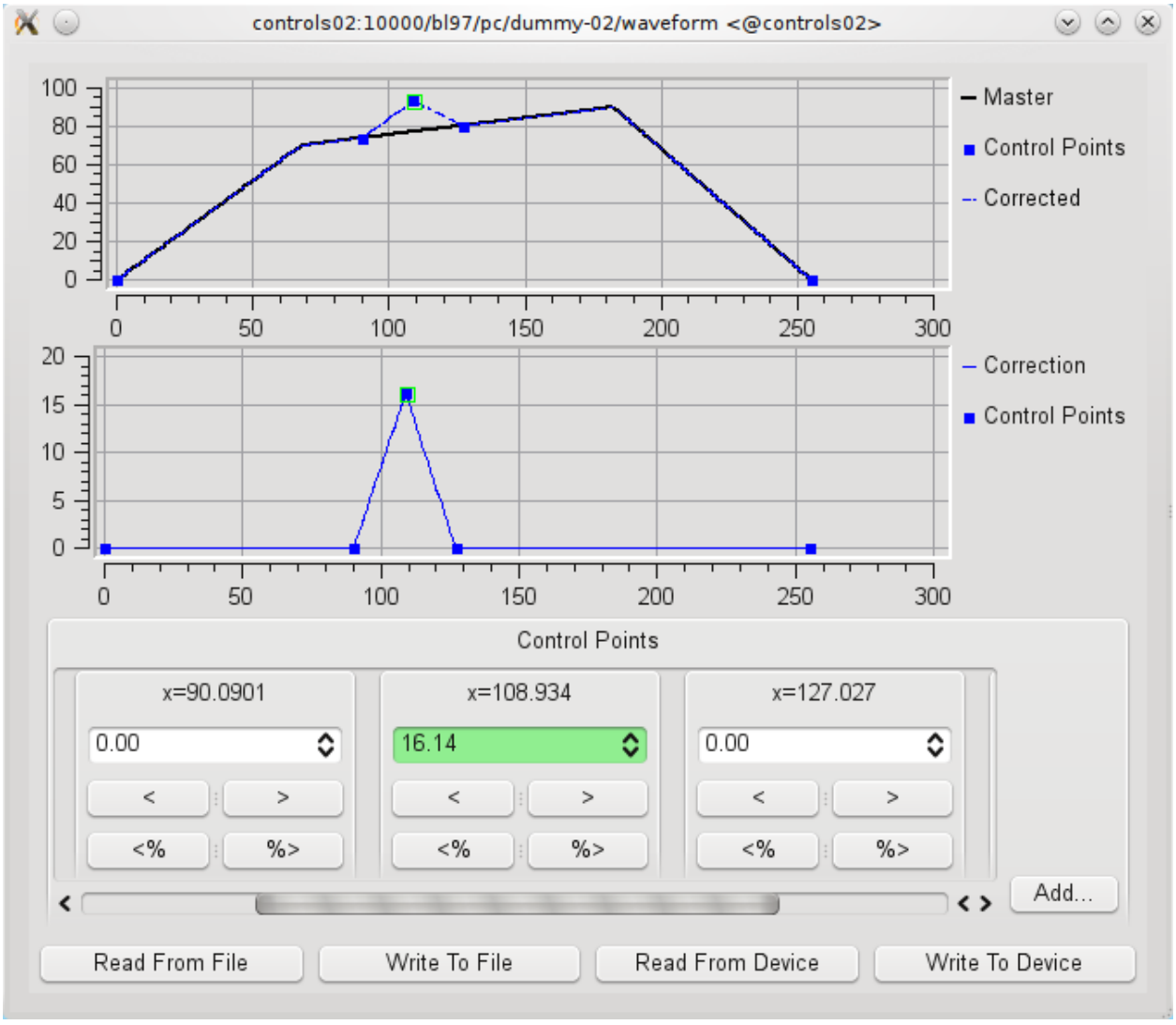
Control points are added by double-clicking in a position of the plot or by using the *Add* button (which allows the user to define a regular grid of control points).

The currently selected control point is highlighted both in the plots and in the controls area.

The arrow buttons in the controls area will help in propagating the related value to the other control points to the left or to the right of the selected one.

Synoptics

Synoptic widgets provide a graphical interface to control equipment. They may display some graphics elements (realistic representations or schematic ones) some of which may allow interactions (such as accept mouse clicks) or provide updated values and changing colors.



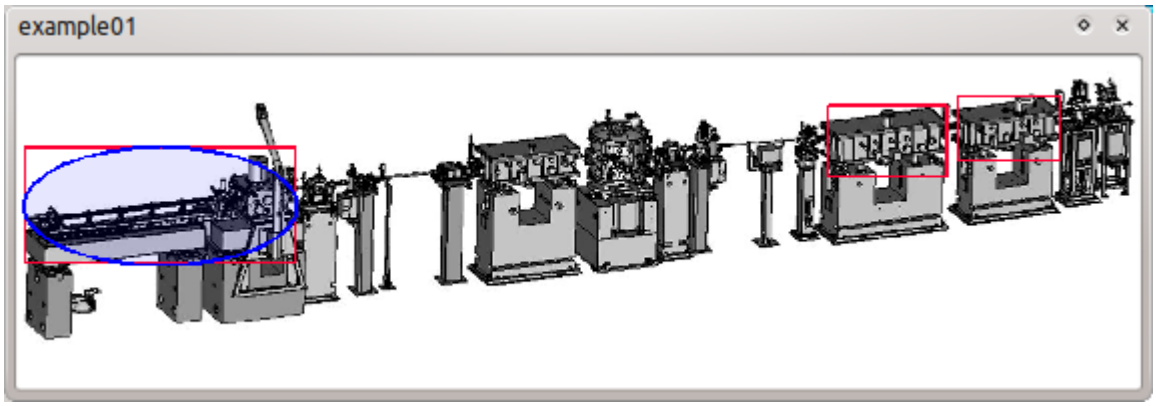


Fig. 1.5: Example of a synoptic representing a beam line. Some active elements are marked with red squares and one has been selected which is shown by a blue ellipse.

Taurus Device Panels User's Interface

There are two different widgets that can be used to provide a view on a Taurus Device: *TaurusDevicePanel* and *TaurusDevPanel*. The first is a compact widget that gives access to attributes and commands of a given device. The second is a full main window application that provides the same access plus a device selector and a trend widget in a more configurable (but less compact way).

TaurusDevicePanel as a stand-alone application

The *TaurusDevicePanel* can be launched as a stand-alone application with the following command:

```
taurusdevicepanel [options] [<device_name>]
```

Run the following command for more details:

```
taurusdevicepanel --help
```

TaurusPanel as a stand-alone application

The *TaurusDevPanel* can be launched as a stand-alone application with the following command:

```
tauruspanel [options] [<device_name>]
```

Run the following command for more details:

```
tauruspanel --help
```

TaurusGUI User's Interface

Contents

TaurusGui is not a GUI in itself but a framework for building Taurus graphical user interfaces in an efficient way (but not all Taurus-based applications are necessarily TaurusGui-based).

A specific TaurusGui-based application is defined in a "configuration directory" which is read by `TaurusGui` to construct the specific GUI.

From the user perspective the only important thing is that, while different TaurusGui-based applications may be very different, they all share some features and have a common look-and-feel. This document describes these common features. For documentation on specific aspects of a given TaurusGui-based application, please refer to the documentation of that application.

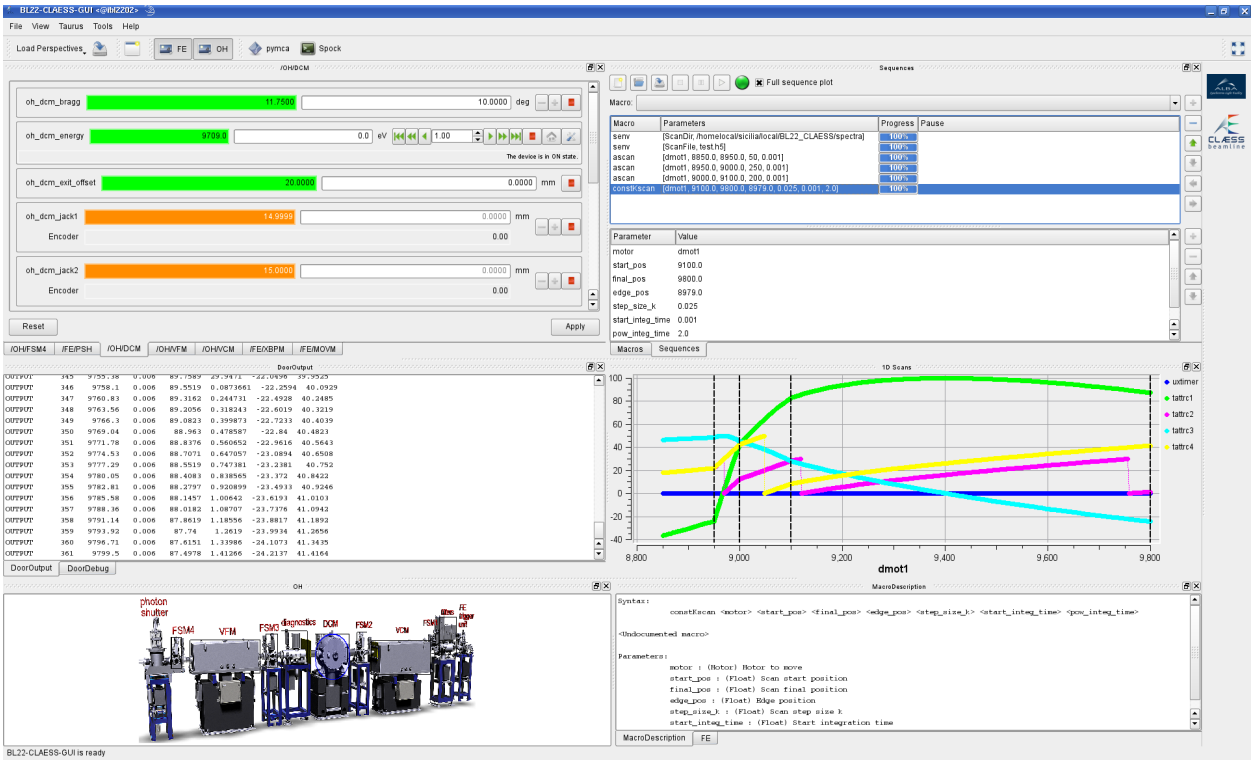


Fig. 1.6: An example of a Beamline GUI based on TaurusGui

Launching TaurusGUI

Normally you will be launching a specific GUI built on top of TaurusGUI (as opposed to launching TaurusGUI directly), so you should refer to the specific instructions for launching that GUI.

Still, you may want to launch TaurusGui directly for debugging an specific application or for creating a new GUI from scratch. For this, use the following command:

```
taurusgui [options]
```

Options:

```
-h, --help            show this help message and exit
--config-dir=CONFIG_DIR
                      use the given configuration directory for
                      initialization
--new-gui             launch a wizard for creating a new TaurusGUI
                      application
--version             show program's version number and exit

Taurus Options:
  Basic options present in any taurus application

  --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
```

Note:

for backwards compatibility: taurusgui CONFIG_DIR

is equivalent to: taurusgui –config-dir=CONFIG_DIR

General structure of a TaurusGUI application

TaurusGui-based applications consist in a main window containing a number of *panels*.

Each panel contains a widget (e.g., a *form*, a *plot*, a *synoptic*...) and is largely independent on the rest of the panels.

You can *re-arrange the panels* by moving and resizing them in the window, or hide/show them or leave some of them floating as independent dialogs. You can even *add new panels* to the application "on-the-fly", selecting them from a catalog of available widgets.

This is because a key feature of TaurusGui applications is that you can adapt them to your needs. But since your needs are not always the same, the TaurusGui- based applications allow you to save the panel arrangement and other user modifiable choices as a *perspective*. You can then easily switch to different application configurations.

Another characteristic element of Taurus-Gui based applications is the *Applets* area on the right side of the main window. This area is typically populated with small widgets providing quick access to some relevant information or control. It also contains the the logo of the institute (or some other general logo) and the logo of the application.

Other common components of TaurusGui-based applications (although not necessarily present) are:

- *Synoptic panels*
- *Macro Execution panels*
- Help browser

Re-arranging panels (moving, resizing, hiding,...)

Note: In order to prevent accidental rearranging of panels, the view can be locked, which will disable the drag and drop of panels. You can lock/unlock the view from the *View Menu*

Panels can be moved around by dragging them from their title bar. When dropping a panel, you can choose whether to push other panels to make room for the new one, or to drop it on top of an existing panel (this will "tabify" the panels) or to leave it floating outside of the main window.

You can also modify the dimensions of a panel without changing its position: just drag the boundaries between panels.

If the view has not been locked, the title bar of each panel contains 2 buttons: one for switching from floating to docked state and another for hiding the panel. Hidden panels are not shown, but are still available in the application. You can show them again by using the View->Panels menu.

Also, some panels may be connected to a synoptic panel and will be shown (if hidden) when selecting a certain element in the synoptic. Refer to the *Synoptic Panels* section for more details.

Tip: If, for some reason, you changed the panels too much and would like to revert the changes, you can always load a previously saved *perspective*.

Creating custom panels

Typically, a TaurusGui-based application will come out-of-the-box with one or more pre-defined panels providing the application functionality. But you may want to extend the application by adding custom panels to provide more features (e.g., to add an extra plot panel, or a new form).

You can add a new panel by clicking in the "New Panel" button of the main tool bar (or selecting *Panels->New Panel...*). This will open a dialog offering a catalog of different panel types and options for your new panel. Once accepted, the new panel will appear floating, ready to be docked to the main window.

Tip: A common use case is creating a new form panel (empty) and then fill it by dropping elements from other existing forms in order to group together controls that are scattered over several panels

When you close the application (or when you save a *perspective*), a dialog will be shown if you have created any custom panels. In this dialog you can choose which of the custom panels you want to keep for future use and which are only meant for the current session.

You can also open this dialog from the *Panels->Switch temporary/permanent status...* option.

Tip: if you want to remove a custom panel from an application, just hide it and use the *Switch temporary/permanent status...* option for making it "temporary" so that it is not stored for future sessions.

Perspectives

Different tasks often require different application layouts and different control widgets to be at hand (e.g., casual use VS expert use, or equipment calibration VS equipment operation). TaurusGui-based applications typically provide a set of pre-defined *perspectives* which are just configurations that can be loaded at any moment to set a given panel arrangement suited for a given task.

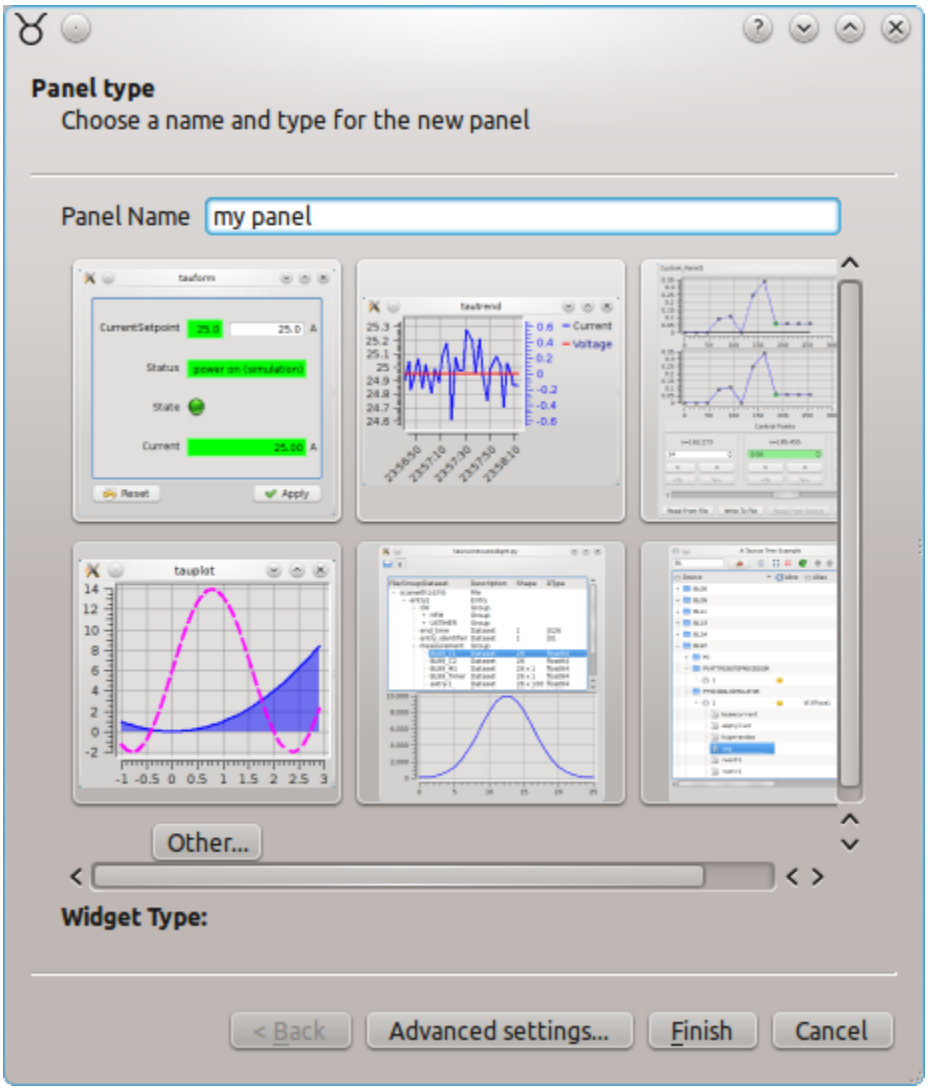


Fig. 1.7: The New Panel dialog

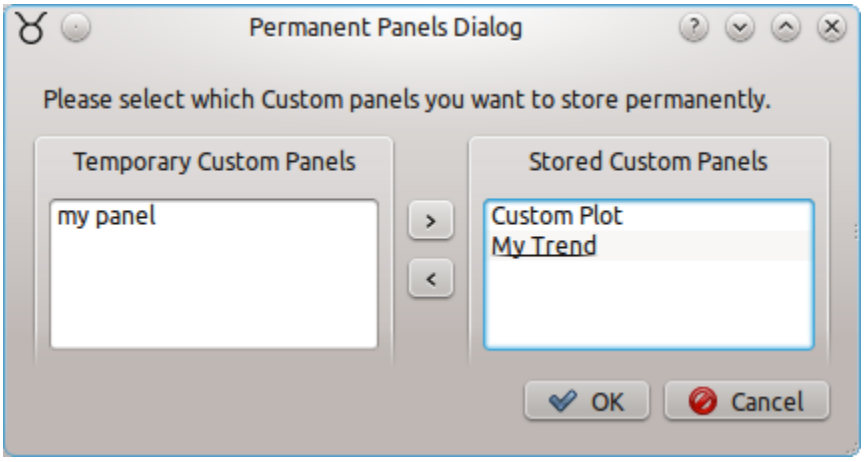


Fig. 1.8: The dialog for selection which custom panels are to be stored permanently

All TaurusGui based applications provide a Perspectives Toolbar (if it is not shown, you can enable it from *View->Toolbars*, or you can access its functionalities from the *View* menu).

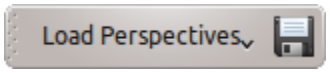


Fig. 1.9: The perspectives toolbar. The button on the left is a drop-down that shows all available perspectives. The button on the right allows you to save the current arrangement as a perspective.

Switching to another perspective can be done using the *Load perspectives* drop-down button in the perspectives toolbar (or using the *View->Load perspective* option).

Apart from the pre-defined perspectives, you can always re-arrange panels and store your preferred arrangement as a perspective using the *Save perspective* button in the perspectives toolbar (or using the *View->Save perspective* option).

Tip: If you want to backup your current perspectives, or you want to use some perspectives you created in one computer in another computer (or another user of the same computer) you can do so by using the *File->Export Settings ...* option. Similarly, use the *File->Import Settings ...* option to update the application perspectives with those contained in the imported file.

Synoptic panels

A special type of panel present in some TaurusGui-based applications is the Synoptics. Synoptics panels are typically used for providing a visual representation of the hardware that is controlled by the GUI. Some elements or areas of the synoptic panel may be *active*, meaning that they can be selected with the mouse.

A default feature of the TaurusGUI framework is that active elements of synoptic panels whose name matches that of an existing panel of the application will cause that panel to be shown (and conversely, if the panel is selected, the element in the synoptic will be highlighted). This is very useful because synoptic panels can be used as a sort of quick index or browser to navigate in panel-crowded applications.

To add a Synoptic panel to a taurusgui after the creation of the taurusgui, use the "Add Panel" button (or menu), select the "TaurusJDrawSynopticsView", enter "Advanced settings..." to enter the full path of the JDraw file into the "Model" field.

Also note that you can find a button in the application toolbar for showing/hiding each synoptic panel.

External Application Launchers

TaurusGui-based applications allow you to add and remove launchers for "external applications". An external application is some other program already installed in the system which will be launched as an independent process (i.e., the TaurusGui just provides a convenience launcher for some related but independent program).

External application launchers may be pre-defined in the application and new ones can be added and removed at any moment by using the corresponding option of the *Tools->External Applications* menu.

Just as with the *custom panels*, when you close the application (or when you save a *perspective*), a dialog will be shown if you have created any custom launchers. In this dialog you can choose which of the custom launchers you want to keep for future use and which are only meant for the current session.

Tip: You may find external application launchers useful for launching some arbitrary python script that automates some task related to the purpose of the GUI.

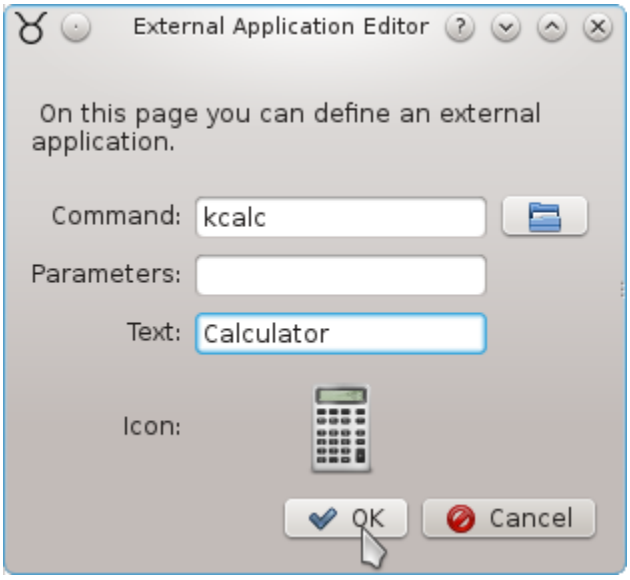


Fig. 1.10: External application editor. You can associate a command, a text and an icon to a new launcher

Sardana integration (Macroserver & Pool)

Macro execution panels

Many TaurusGui-based applications are used to interact with a Sardana *Pool* and *MacroServer* in order to run macros (e.g. scans). In these cases several macro- related panels will be available:

- *Macros*, a macro executor widget. See this for further information.
- *Sequences*, a macro sequencer widget. See this for further information.
- *MacroDescription*, a text panel which provides documentation on the selected macro.
- *DoorOutput*, a text panel that logs the output of macro executions (similar to what you would see if launching the macro in a *spock* console)
- *DoorDebug*, a text panel that logs the debug messages of macro executions.
- *DoorResult*, a text panel that logs the return value of the macro executions.
- *Experiment Config*, *an experiment configuration* <*expconf_ui*> widget for configuring acquisition and display parameters
- *Sardana Editor*, *a Sardana-aware code editor* <*sardanaeditor_ui*> for creating/modifying Sardana macros.

Also, some other temporary panels may be dynamically created depending on the experiment configuration:

- *1D Scans*, *a trend* that plots the values of scalar attributes during some scan macro executions.

Note: You can run *taurusgui macrogui* for seeing an example of a TaurusGUI- based application that provides the aforementioned panels

---

In most specific GUIs the macroserver and door name to use are pre-configured and the user does not need to change them. Sometimes though, you may want to alter it. You can do so by using the *Taurus->Macro execution configuration...* option.

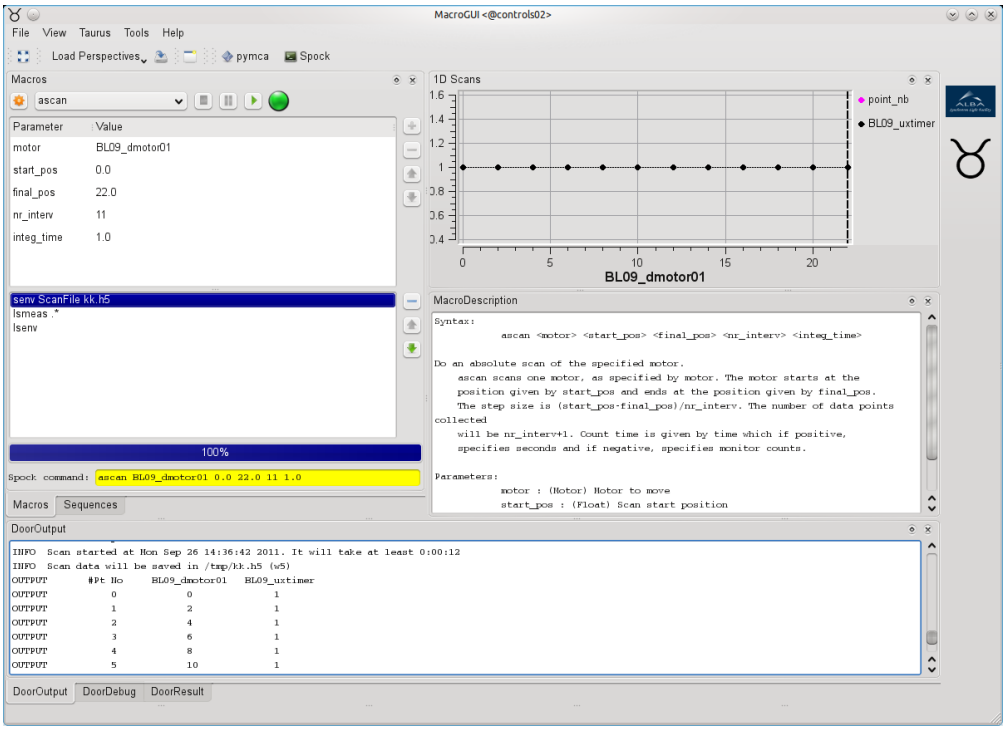


Fig. 1.11: Macro-related panels (taken from the macrogui example that can be launched with *taurusgui macrogui*)

Automatic creation of Instrument panels from Pool info

Some TaurusGui-based GUIs may also use information provided by the Pool Device Server about the Control System to automatically create *TaurusForm* panels containing all the controls related to the various instruments to be controlled by the GUI. An important characteristic is that, since these so-called *instrument panels* are created dynamically when launching the application, their number and/or contents may vary, reflecting changes in the pool configuration.

Examples of TaurusGui based applications

Taurus includes examples of TaurusGUI-based applications for demonstration purposes.

Table 1.1: Examples of Taurus-GUI based applications

GUI name	launch command	Comment
example01	taurusgui example01	Included in Taurus (<taurus>/qt/qtgui/taurusgui/conf/tgconf_example01)
Macrogui	taurusgui macrogui	Included in Taurus (<taurus>/qt/qtgui/taurusgui/conf/tgconf_macrogui)
Alba's beam lines control GUIs	ctblxx (xx is the beamline number)	Internal ALBA development

Known Issues

Cannot drop a panel (it stays floating)

This typically occurs because there is no enought room to acomodate the panel width or height in any of the available "slots". Try to make room by hiding some other panel, or tabifying other panels together, or increasing the main window size.

Taurus Demo User's Interface

Contents

Taurus Demo is a GUI providing a launcher of many of the Taurus Widgets for demonstration purposes.

Taurus Demo Application

taurusdemo is an application that gives an overview of some of the widgets available in Taurus. taurusdemo can be launched with the following command:

```
taurusdemo
```

Run the following command for more details:

```
taurusdemo --help
```

Taurus Remote Log Monitor User's Interface

Contents

Taurus provides an application to visualize logs generated by a running Taurus Application. The visualization can be done either in a GUI (based on `QLoggingWidget`) or in the command line interface.

This application can be launched using the following command:

```
taurusremotelogmonitor [options]
```

Run the following command for more details:

```
taurusremotelogmonitor --help
```

Taurus Configuration Browser User's Interface

Fig. 1.12: Taurusdemo application

Contents

Taurus provides an user interface for browsing TaurusMainWindow setting files (.ini files), allowing a power user to visualize and edit the different perspectives of a Taurus GUI. The configuration tree is organized in branches, with the perspective at the root.

Taurus Configuration Browser Application

You may browse the configuration of a TaurusMainWindow-based application (e.g. any TaurusGUI) by launching the taurusconfigbrowser application with the following command:

```
taurusconfigbrowser [<configuration.ini>]
```

Run the following command for more details:

```
taurusconfigbrowser --help
```

In the figure below the taurusconfigbrowser application shows a taurus configuration .ini file containing three perspectives: 'LAST', 'second_perspective' and 'third_perspective'. 'LAST' is an special perspective automatically stored just before the closing of the Taurus GUI; it reflects the state of the GUI just before the last closing. The other two perspectives were saved explicitly during the Taurus GUI execution.

This section explains some features that are common to most applications built with taurus. This is done from the GUI user point of view (not for developers).

For a detailed list of features of each widget, please refer to the *Developer's Guide*

Screenshots

Here you will find a host of example figures.

Developer's Guide

Taurus 3.x to 4.x migration guide

This chapter explains how to migrate from an application written using Taurus 3.x to Taurus 4.x.

Todo

This section needs to be expanded. **Help wanted!**. In the meanwhile, the following documents may be useful:

- TEP3
- TEP14
- Taurus 4 API changes
- Taurus 4 slides

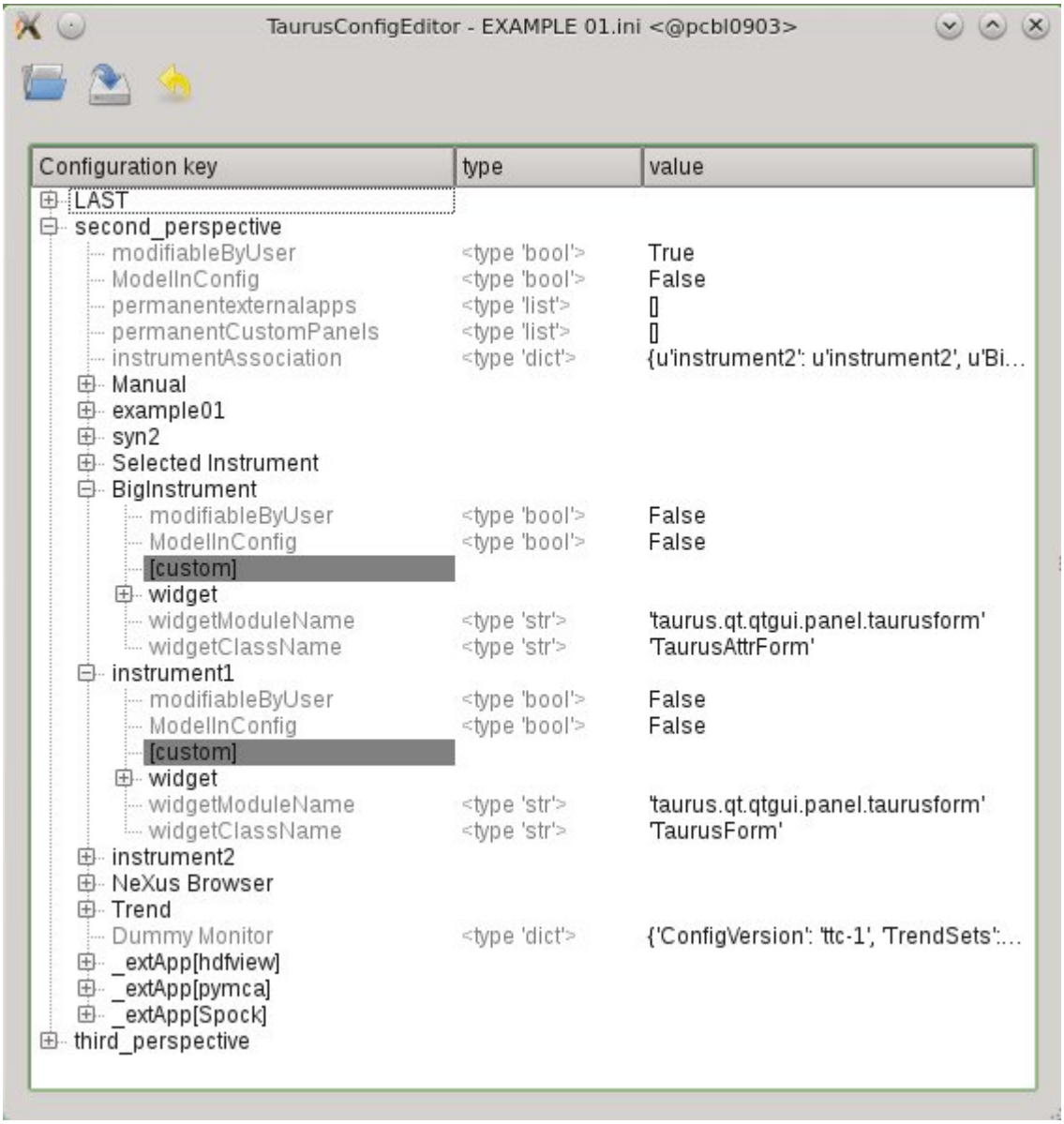


Fig. 1.13: taurusconfigbrowser with a perspective unfolded to show the panel entries

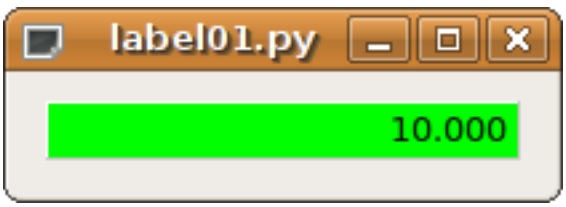


Fig. 1.14: Display attribute value

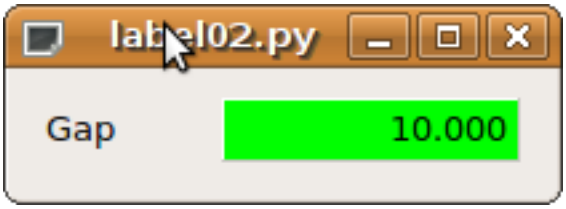


Fig. 1.15: Display attribute value with label

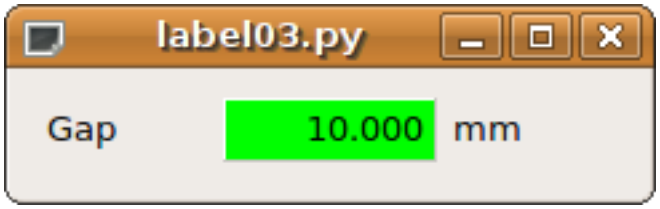


Fig. 1.16: Display attribute value with label & unit

Fig. 1.17: Interactively display attribute

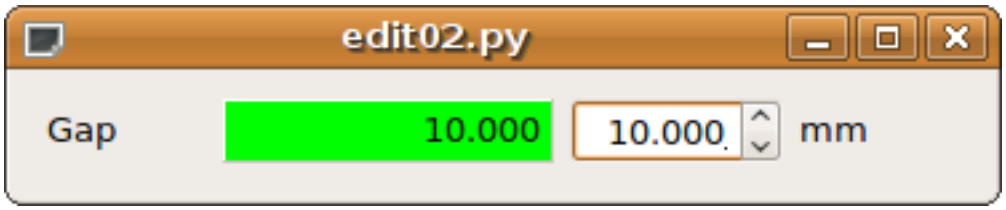


Fig. 1.18: Interactively display attribute with spinbox

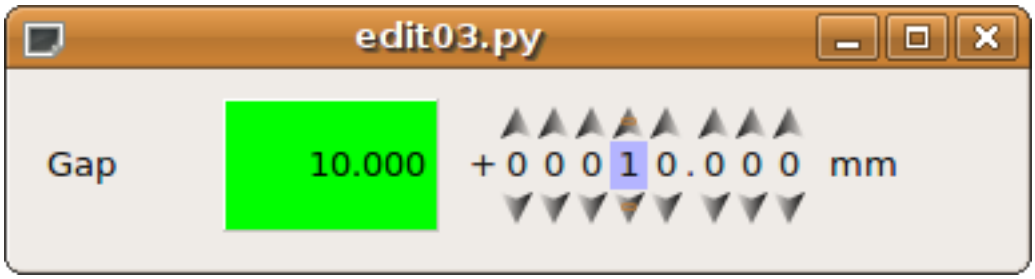


Fig. 1.19: Interactively display attribute with wheel

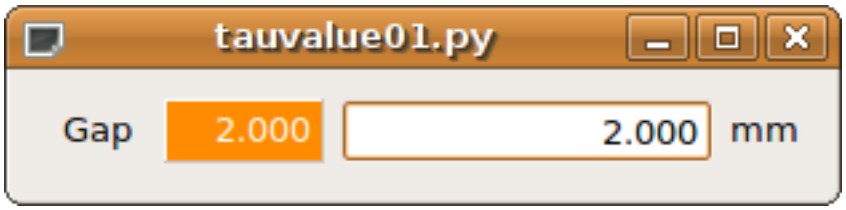


Fig. 1.20: The standard attribute display widget

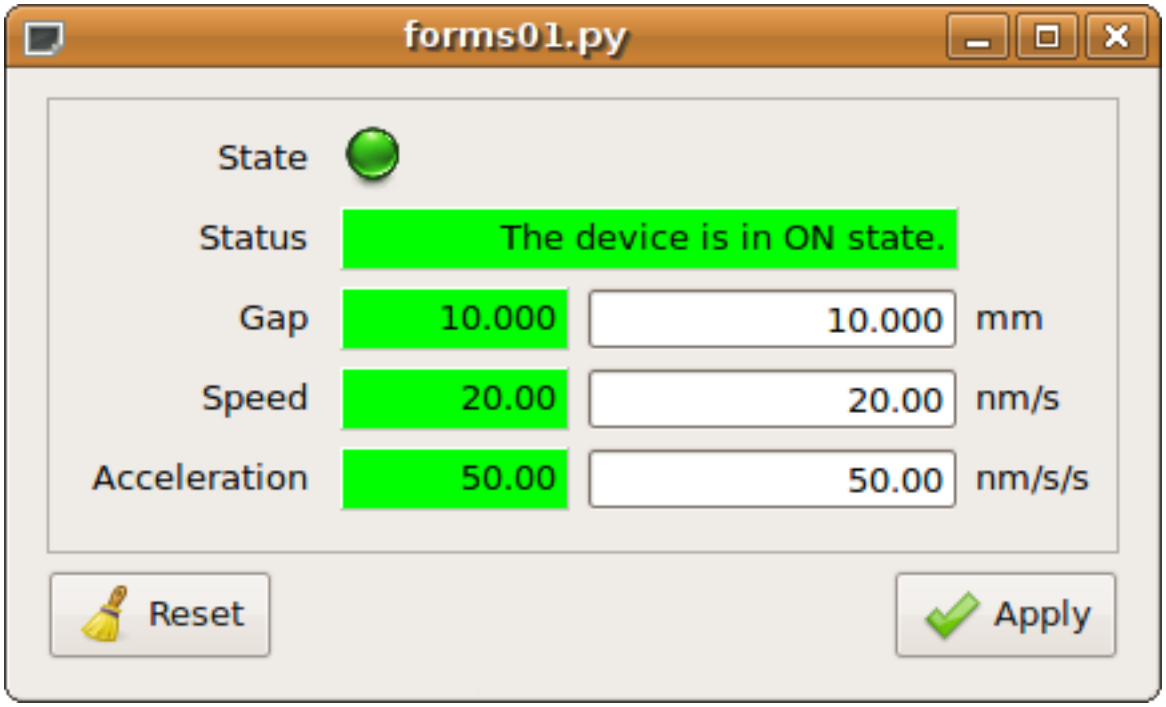


Fig. 1.21: Using forms

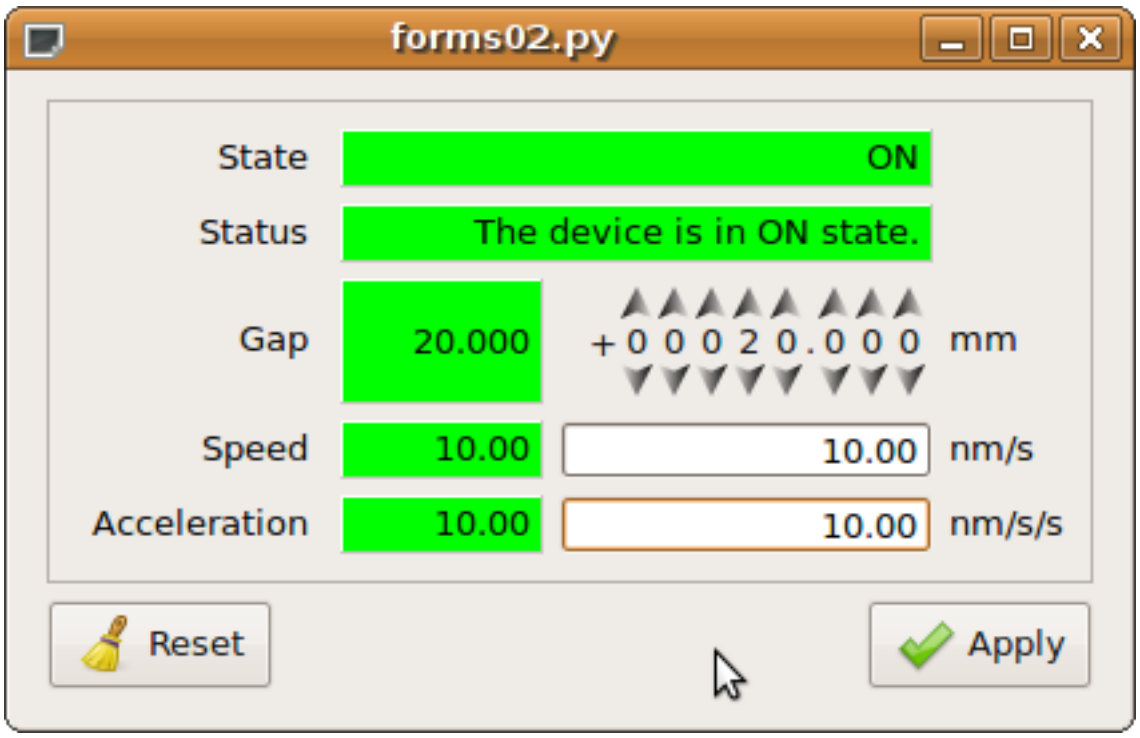


Fig. 1.22: Customized form

Taurus development guidelines

Overview

This document describes taurus from the perspective of developers. Most importantly, it gives information for people who want to contribute to the development of taurus. So if you want to help out, read on!

How to contribute to taurus

Taurus is Free Software developed in open way. Contributions to code, documentation, etc. are always welcome.

The "official" Taurus source code is hosted in a git repository.

The details in how to contribute are described in the *CONTRIBUTING.md* file at the root of the git repository.

Documentation

All standalone documentation should be written in plain text (`.rst`) files using reStructuredText for markup and formatting. All such documentation should be placed in directory `docs/source` of the taurus source tree. The documentation in this location will serve as the main source for taurus documentation and all existing documentation should be converted to this format.

Coding conventions

- Code in Taurus should follow the standard Python style conventions as described in PEP8. Specially:

  - Use 4 spaces for indentation
  - Respect the maximum of 79 characters per line
  - Surround top-level function and class definitions with two blank lines.
  - use `lower_case` for module names. If possible prefix module names with the word `taurus` (like `taurusutil.py`) to avoid import mistakes.
  - use `CamelCase` for class names
  - use `lower_case` for method names, except in the context of taurus.qt where the prevailing convention is `mixedCase` due to influence from PyQt
- Code must be python 2.7 compatible, and, if possible, new contributions should also consider being compatible with python3.5 (to prepare for python3 support)
- Every python module file should contain license information (see template below). The preferred license is the LGPL. If you need/want to use a different one, it should be compatible with the LGPL v3+.
- avoid polluting 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)
- All public API code should be documented (modules, classes and public API) using Sphinx extension to reStructuredText

The following code can serve as a template for writing new python modules to taurus:

```
#!/usr/bin/env python

##############################################################################
##
# This file is part of Taurus
##
# http://taurus-scada.org
##
# Copyright 2011 CELLS / ALBA Synchrotron, Bellaterra, Spain
##
# Taurus 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.
##
# Taurus 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 Taurus.  If not, see <http://www.gnu.org/licenses/>.
##
##############################################################################

"""A :mod:`taurus` module written for template purposes only"""

__all__ = ["TaurusDemo"]

__docformat__ = "restructuredtext"
```

```
class TaurusDemo(object):
    """This class is written for template purposes only"""

def main():
    print "TaurusDemo"

if __name__ == "__main__":
    main()
```

Special notes about Qt programming

The following Qt guidelines are intended to ensure compatibility between all PyQt4, PyQt5 and PySide versions.

1. Avoid importing PyQt / PySide directly. Imports like:

   ```
   from PyQt4 import Qt
   from PyQt4 import QtCore
   from PyQt4 import QtGui
   from PyQt4 import QtNetwork
   from PyQt4 import QtWebKit
   from PyQt4 import Qwt5
   ```

   Should be replaced by:

   ```
   from taurus.external.qt import Qt
   from taurus.external.qt import QtCore
   from taurus.external.qt import QtGui
   from taurus.external.qt import QtNetwork
   from taurus.external.qt import QtWebKit
   from taurus.external.qt import Qwt5
   ```

2. Since Taurus v>=4.0, Qt-based code in Taurus may assume that PyQt API v2 is used. PyQt API 1 code, which was supported by Taurus 3, is no longer guaranteed to work.

   - Use standard python strings (e.g., use `str` for Qt strings instead of `QString`). Code like:

     ```
     my_string = Qt.QString(" hello ")
     my_string2 = my_string.trimmed()
     ```

     Should be replaced by:

     ```
     my_string = " hello "
     my_string2 = my_string.strip()
     ```

   - Do not use `QVariant`. QVariant objects don't exist in PySide or in the new PyQt4 API 2. Code like:

     ```
     def setData(self, index, qvalue, role=Qt.Qt.EditRole):
         value = qvalue.toString()  # this assumes qvalue to be a :class:`QVariant`
         self.buffer[index.column()] = value

     def data(self, index, role=Qt.Qt.DisplayRole):
         value = self.buffer[index.column()]

         if role == Qt.Qt.DisplayRole:
             return Qt.QVariant(value)
         else:
             return Qt.QVariant()
     ```

Should be replaced by:

```
def setData(self, index, value, role=Qt.Qt.EditRole):
    self.buffer[index.column()] = value  # value is already a python object

def data(self, index, role=Qt.Qt.DisplayRole):
    value = self.buffer[index.column()]

    if role == Qt.Qt.DisplayRole:
        return value
    else:
        return None
```

For compatibility reasons, `Qt()` defines *QVariant* and *from_qvariant* which is internally used used to write code that supports both API v1 and v2 for QVariant. But new code in Taurus v>=4 may assume v2 only.

3. Use new-style signals. Old-style code like the following:

```
class MyWidget(Qt.QWidget):

def foo(self):
    self.connect(self, Qt.SIGNAL('mySignal(int)', self.bar))
    self.emit(Qt.SIGNAL('mySignal(int)', 123))
```

Should be replaced by:

```
class MyWidget(Qt.QWidget):

    mySignal = Qt.pyqtSignal(int)

    def foo(self):
        self.mySignal.connect(self.bar)
        self.mySignal.emit(123)
```

4. Use of *taurus.qt.qtgui.application.TaurusApplication* instead of `QApplication` is recommended (it takes care of various initialization and exit tasks that are convenient).

Creating GUIs with the TaurusGUI framework

The easiest way to create a new GUI using Taurus is by invoking:

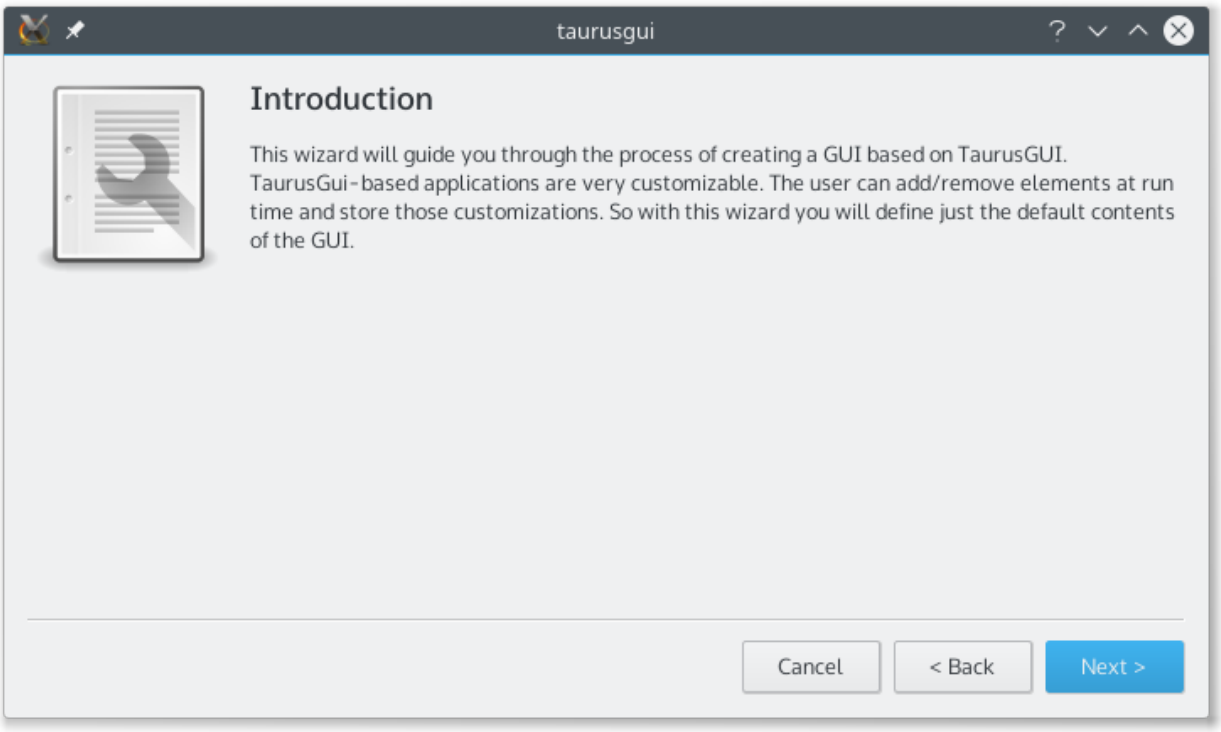
```
taurusgui --new-gui
```

This shows a "wizard" application that will guide you through the process of creating a *TaurusGui*-based GUI in a few minutes without having to program a single line of code.

The taurus GUI thus created can be populated with *panels* containing any Qt widget (typically from Taurus, but it may also be from any other arbitrary module).

The GUI can also be modified and extended at execution time as explained in *this section of the user guide*.

The Taurus widgets can be associated with models when they are added to the GUI and, in many cases, they may also accept drag & drop of the model name(s) from other widgets (or from a model selector) at any time, e.g.: the user can start plotting the time evolution of a given value by just dragging its name from a TaurusForm and "dropping" it into a TaurusTrend widget.



Advanced control over the GUI

While the procedure described above is enough in many cases, sometimes more control over the GUI contents, appearance and behaviour is required. This more advanced control can be exerted at several levels:

- First, it is possible to edit the configuration files that define a TaurusGUI-based application. These are declarative python and XML files (editable as plain text) complemented by Qt settings files (editable with the provided *taurusconfigbrowser* application).
- On a lower level, custom specific widgets (created either programmatically, as in the *Examples* or via the *Qt designer*) can be added as panels to a TaurusGUI application.
- At the same level, it is also possible to do simple inter-panel communication thanks to a `taurus.qt.qtcore.communication.SharedDataManager` broker component instantiated by the GUI.
- Finally, the maximum level of control can be achieved by programmatically accessing the `TaurusGui` class itself. In this way, all the higher level features described before are still available, while there are no limitations on the customizations that can be done.

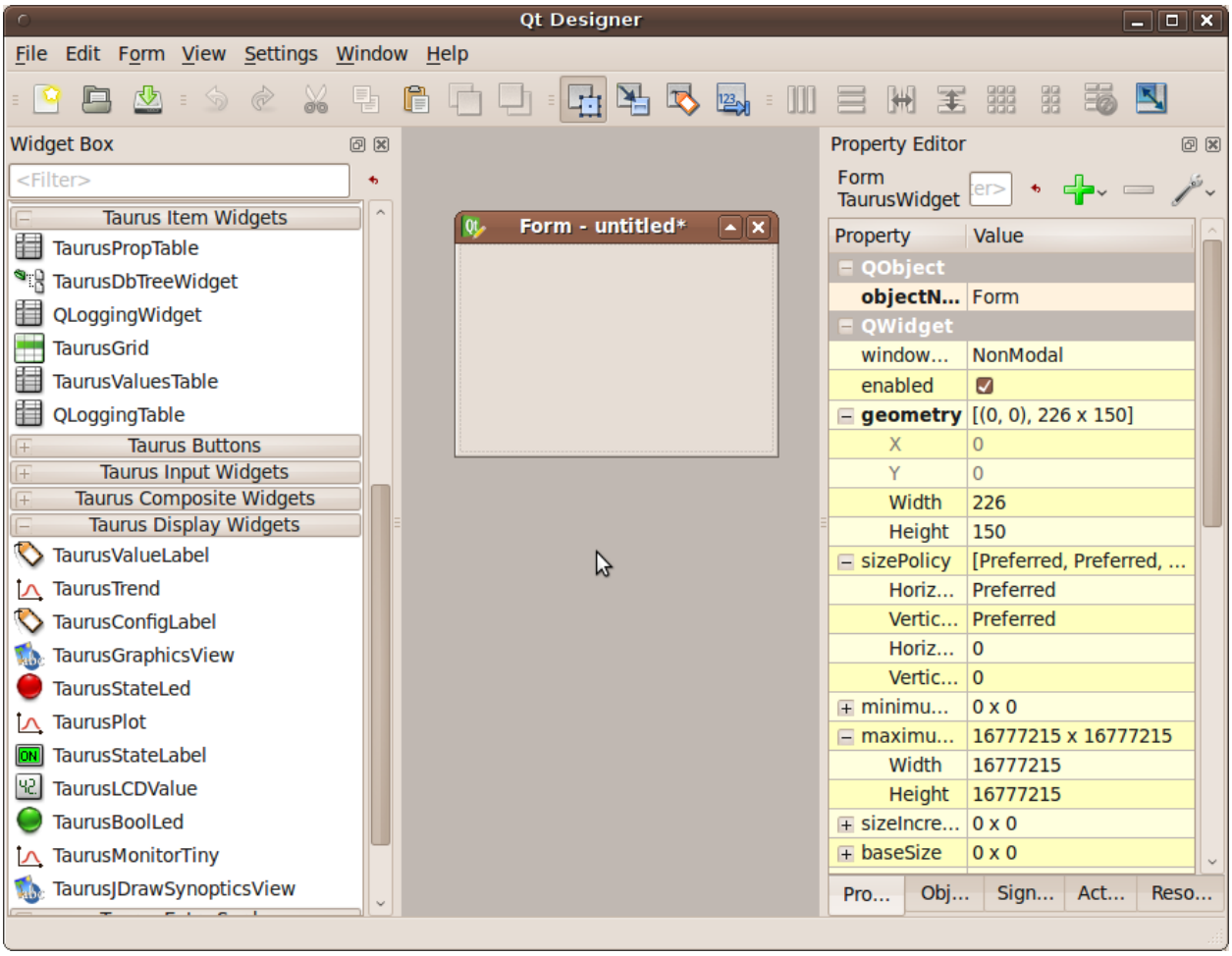
Taurus Qt Designer tutorial

Taurus widgets behave just as any other Qt widget, and as such, they can be used to create GUIs in a regular way, both programmatically or using the Qt designer. For convenience, Taurus provides the *taurusdesigner* command that launches the standard Qt designer application extended to show also the widgets provided by Taurus.

To launch it, just execute:

```
taurusdesigner
```

Tip: `--help` argument will give you the complete list of options



You can then design your application/widget using not only the standard Qt widgets but also the taurus widgets.

You can use the Taurus Qt Designer to define a full GUI, but instead we recommend to create the GUIs using the *TaurusGUI framework* and use the Taurus Qt Designer just for creating widgets to be inserted as panels in a `taurus.qt.qtgui.taurusgui.TaurusGui`-based GUI.

Using the .ui file

The Qt designer will produce a .ui file that is an XML representation of the application/widget that you designed.

This .ui file can then be used in your own widget by using the `taurus.qt.qtgui.util.UILoadable()` decorator.

See TEP11 for more details.

Taurus icon guide

Usually the application/widget you are developing will require some icons. Some of these icons will be used to represent a standard actions, applications, places or status like “open a file”, “desktop” or “preferences”.

Qt (and therefore, Taurus) supports theme icons for many common cases. You can access them via the standard `QIcon.fromTheme()` method. Using theme icons will make your application be more integrated with the rest of the system, but keep in mind that different people will see the icons differently depending on their default theme (so do not use a theme icon for something not related to its specification just because in *your* theme it happens to look as what you want)

Apart from the theme icons, Taurus provides some collections of icons (and you can add more (see *taurus.qt.qtgui.icon*). The paths containing these collections are automatically added to *QDir*'s search paths under various prefixes when you import *taurus.qt.qtgui* (or any of its submodules).

The following example shows how to use theme and non-theme icons in your application:

```
from taurus.external.qt import Qt
from taurus.qt.qtgui.application import TaurusApplication


class MyGUI(Qt.QMainWindow):

    def __init__(self, parent=None):
        Qt.QMainWindow.__init__(self, parent)
        toolbar = self.addToolBar("Tools")

        # get icon from theme
        icon1 = Qt.QIcon.fromTheme("document-open")

        # get icon using prefix + filename
        icon2 = Qt.QIcon("actions:exit.svg")

        toolbar.addAction(icon1, "Open HDF5", self.open_file)
        toolbar.addAction(icon2, "Exit", self.close)


    def open_file(self):
        pass    # do something

if __name__ == "__main__":
    import sys
    app = TaurusApplication()
    gui = MyGUI()
    gui.show()
    sys.exit(app.exec_())
```

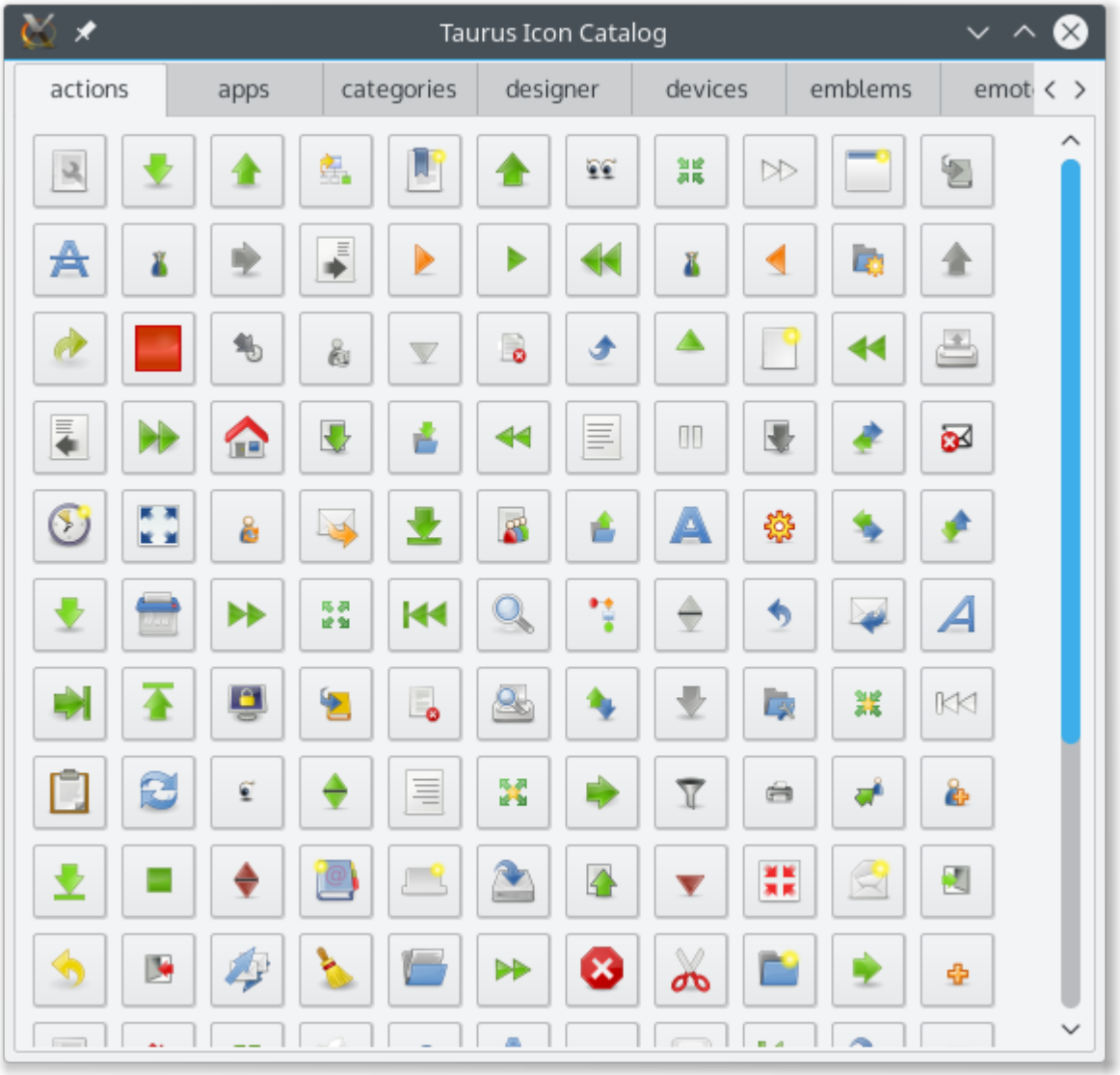
Taurus icon catalog

In order to explore the icon collections provided by Taurus, you can use the *taurusiconcatalog* application, which will let you browse the icons.

By clicking on an icon of the catalog, you will obtain a larger view of the icon as well as detailed information on how to access it from your application.

taurus core tutorial

The core has been designed to provide a model-based abstraction to the various sources of data and/or control objects supported via the Taurus schemes (we use the term "model" to refer to the model component in an MVC driven architecture).


Taurus Icon Catalog
actions
apps
categories
designer
devices
emblems
emot

An scheme is a plugin for Taurus that provides the glue between Taurus and a given source of data and/or of objects that can be controlled. For example, schemes exist for various control system libraries (such as Tango, or EPICS) as well as for processing data (e.g. the `taurus.core.evaluation` scheme).

Each scheme implements at least a Factory (derived from `taurus.core.TaurusFactory`) which provides Taurus model objects , for a given model name.

Model concept

All Taurus Elements (Devices, Attributes, etc) are model objects with an associated unique name. The model name is an URI (as defined in RFC3986).

In practice, the URIs have the following form (for a complete and rigorous description refer to RFC3986):

[<scheme>:][<authority>][<path>][?<query>][#<fragment>]

Notes:

- The <authority>, if present, starts by '//'
- The <path>, if present, starts by '/' (except for relative URIs)

A model object (also referred to occasionally as Taurus Element) is an instance of a class derived from one of `taurus.core.TaurusAuthority`, `taurus.core.TaurusDevice`, `taurus.core.TaurusAttribute`.

Examples of model names

Different schemes may choose different conventions to name the models that they provide.

The following are some examples for the `taurus.core.tango` scheme:

The full Taurus model name for a Tango device *sys/tg_test/1* registered in a Tango Database running on *machine:10000* is:

`tango://machine:10000/sys/tg_test/1`

Now, if we assume that:

- `tango` is set as the default scheme and that
- `machine:10000` is set as the default TANGO_HOST
- and that `tgtest1` is set as an alias of `sys/tg_test/1`

then the same Tango device could be accessed as:

`tgtest1`

In the case of Tango attributes, here are some equivalent examples given the above assumptions:

`tango://machine:10000/sys/tg_test/1/double_scalar`,

`sys/tg_test/1/double_scalar`,

`tango:tgtest1/double_scalar`,

`tgtest1/double_scalar`

See *taurus.core.tango* for a more exhaustive description and more examples related to Tango.

The following are some examples for the *taurus.core.evaluation* scheme:

An evaluation attribute that generates an array of dimensionless random values when read:

`eval:rand(256)`

An evaluation attribute that applies a multiplication factor to an existing tango attribute (and which is updated every time that the tango attribute changes):

`eval:123.4*{tango:sys/tg_test/1/double_scalar}`

Or one that adds noise to a tango image attribute:

`eval:img={tango:sys/tg_test/1/short_image_ro};img+10*rand(*img.shape)`

And, by using custom evaluators, one can easily access virtually anything available from a python module. For example, using the datetime module to get today's date as a Taurus attribute:

`eval:@datetime.*/date.today().isoformat()`

See *taurus.core.evaluation* for a more exhaustive description and some tricks with the Evaluation scheme and the custom evaluators.

Now an example for the *taurus.core.epics* scheme. The model name for the EPICS process variable (PV) "my:example.RBV" is:

`epics:my:example.RBV`

Note that you can create your own schemes and add them to taurus (e.g., an scheme to access your own home-brew control system). Some schemes that are in our TO-DO list are:

- A scheme to access datasets in HDF5 files as Taurus attributes
- A scheme to access ranges of cells in a spreadsheet file as Taurus attributes
- A scheme to access column/row data in ASCII files as Taurus attributes
- A scheme to access data from mySQL databases as Taurus attributes
- A scheme to access Tango-archived data as Taurus attributes

model access

Taurus users are encouraged to write code that is "scheme-agnostic", that is, that it neither assumes the availability of certain schemes nor uses any scheme-specific feature. For this, Taurus provides several high-level scheme-agnostic helpers to obtain the Taurus Element associated to a given model name:

- *taurus.Authority()*
- *taurus.Device()*
- *taurus.Attribute()*
- *taurus.Object()*

The first four helpers require you to know which type of Element (e.g., Attribute, Device,...) is represented by the model name. If you do not know that beforehand, you can use *taurus.Object()* which will automatically find the type and provide you with the corresponding model object (but of course this is slightly less efficient than using one of the first three helpers).

These helpers will automatically find out which scheme corresponds to the given model and will delegate the creation of the model object to the corresponding scheme-specific Factory. Therefore, the returned model object will be of a specialized subclass of the corresponding Taurus generic Element and it will expose the scheme-agnostic API plus optionally some scheme-specific methods (e.g., *taurus.core.tango.TangoDevice* objects provide all the API of a *taurus.core.TaurusDevice* but they also provide all the methods from a `PyTango.DeviceProxy`)

For example, obtaining the device model object for a TangoTest Device can be done as follows:

```
import taurus
testDev = taurus.Device('sys/tg_test/1')
```

or, using *taurus.Object()*:

```
import taurus
testDev = taurus.Object('sys/tg_test/1')
```

Also for example, obtaining the Taurus Attribute model corresponding to the EPICS Process Variable called "my:example.RBV" is just:

```
import taurus
testDev = taurus.Attribute('epics:my:example.RBV')
```

Taurus also provides other helpers to access lower level objects for dealing with models:

- *taurus.Factory()*
- *taurus.Manager()*

And also some useful methods to validate names, find out the element type(s) for a given name and other related tasks:

- *taurus.isValidName()*
- *taurus.getValidTypesForName()*
- *taurus.getSchemeFromName()*

Advantages of accessing Tango via Taurus over PyTango

If you are familiar with PyTango you may be asking yourself what is the real advantage of using taurus instead of PyTango directly for accessing Tango objects. There are actually many benefits from using taurus. Here is a list of the most important ones.

integration with other schemes Taurus is not just Tango. For example, you can treat a Tango Attribute just as you would treat an EPICS attribute, and use them both in the same application.

model unicity: you may request the same model many times without performance hit, since taurus will give you the same object:

```
>>> import taurus
>>> sim1 = taurus.Device('sys/tg_test/1')
>>> sim2 = taurus.Device('sys/tg_test/1')
>>> print sim1 == sim2
True
```

Whereas in PyTango the same code always results in the construction of new DeviceProxy objects:

```
>>> import PyTango
>>> sim1 = PyTango.DeviceProxy('sys/tg_test/1')
>>> sim2 = PyTango.DeviceProxy('sys/tg_test/1')
>>> print sim1 == sim2
False
```

model intelligence: taurus is clever enough to know that, for example, 'sys/tg_test/1' represents the same model as 'tango:SYS/Tg_TEST/1' so:

```
>>> import taurus
>>> sim1 = taurus.Device('sys/tg_test/1')
>>> sim2 = taurus.Device('tango:SYS/Tg_TEST/1')
>>> print sim1 == sim2
True
```

tango event abstraction: taurus cleverly hides the complexities and restrictions of the tango event system. With taurus you can:

- subscribe to the same event multiple times
- handle tango events from any thread

Some optimizations are also done to ensure that the tango event thread is not blocked by the user event handle code.

Taurus custom settings

Taurus provides a module located at its root directory called *tauruscustomsettings*. This module contains some Taurus-wide default configurations.

The idea is that the final user may edit the values here to customize certain aspects of Taurus.

DEFAULT_QT_API = 'pyqt'
Set preffered API if not is already loaded

QT_AUTO_INIT_LOG = True
Auto initialize Qt logging to python logging

QT_AUTO_REMOVE_INPUTHOOK = True
Remove input hook (only valid for PyQt4)

QT_THEME_DIR = ''
Select the theme to be used: set the theme dir and the theme name. The path can be absolute or relative to the dir of taurus.qt.qtgui.icon If not set, the dir of taurus.qt.qtgui.icon will be used

QT_THEME_FORCE_ON_LINUX = True
: In Linux the QT_THEME_NAME is not applied (to respect the system theme) setting QT_THEME_FORCE_ON_LINUX=True overrides this.

QT_THEME_NAME = 'Tango'
: The name of the icon theme (e.g. 'Tango', 'Oxygen', etc). Default='Tango'

Examples

Here you will find a host of example figures with the code that generated them.

In order for the examples to work on your computer, you need to have a Tango device server running. The following section explains how to do this.

Setup

The device server used for the examples can be obtained `here`.

In order for the examples to work as they are provided a TaurusTest device must be created and running with the following configuration:

Server (ServerName/Instance): TaurusTest/taurustest

Class: TaurusTest

Devices: sys/taurustest/1

You can easily configure it from Jive by going to Edit->Create server and type the above parameters in the dialog that pops up.

Common

For the sake of simplicity the code presented below (except for the first example) does not include the following header and footer code lines:

header:

```
import sys
from taurus.external.qt import Qt
from taurus.qt.qtgui.application import TaurusApplication

app = TaurusApplication(sys.argv)
panel = Qt.QWidget()
layout = Qt.QHBoxLayout()
panel.setLayout(layout)
```

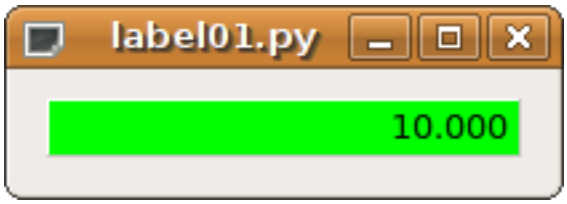
footer:

```
panel.show()
sys.exit(app.exec_())
```

You must prepend and append the above code in order for the examples to work properly.

Display attribute value

Displaying a tango attribute value in a GUI is easy with taurus and *display.TaurusLabel*



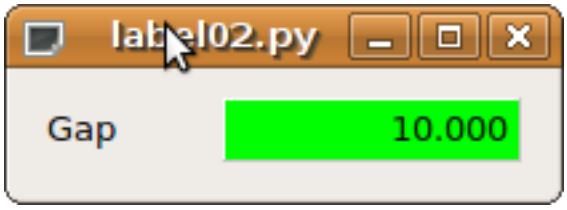
code:

```
import sys
from taurus.external.qt import Qt
from taurus.qt.qtgui.application import TaurusApplication

app = TaurusApplication(sys.argv)
panel = Qt.QWidget()
layout = Qt.QHBoxLayout()
panel.setLayout(layout)

from taurus.qt.qtgui.display import TaurusLabel
w = TaurusLabel()
layout.addWidget(w)
w.model = 'sys/taurustest/1/position'

panel.show()
sys.exit(app.exec_())
```

not much code to write, but... boring!

Display attribute value with label

Let's spice it up a bit: add the tango label for the position attribute so it looks something like this:



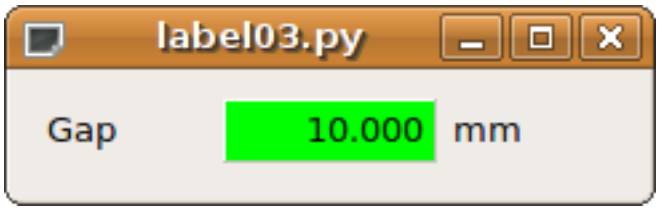
code:

```
from taurus.qt.qtgui.display import TaurusLabel
w1, w2 = TaurusLabel(), TaurusLabel()
layout.addWidget(w1)
layout.addWidget(w2)
w1.model, w1.bgRole = 'sys/taurustest/1/position#label', ''
w2.model = 'sys/taurustest/1/position'
```

Much better indeed!

Display attribute value with label and separate units

And little bit more... add the units.



code:

```
from taurus.qt.qtgui.container import TaurusWidget
from taurus.qt.qtgui.display import TaurusLabel

w1, w2, w3 = TaurusLabel(), TaurusLabel(), TaurusLabel()
layout.addWidget(w1)
layout.addWidget(w2)
layout.addWidget(w3)
w1.model, w1.bgRole = 'sys/taurustest/1/position#label', ''
w2.model = 'sys/taurustest/1/position#rvalue.magnitude'
w3.model, w3.bgRole = 'sys/taurustest/1/position#rvalue.units', ''
```

Nice isn't it?

Interactively display attribute

Humm... Now suppose the user wants to change this value. *input.TaurusValueLineEdit* does this job well (and so does *input.TaurusValueSpinBox* and *input.TaurusWheelEdit*)

Fig. 1.23: With TaurusValueLineEdit

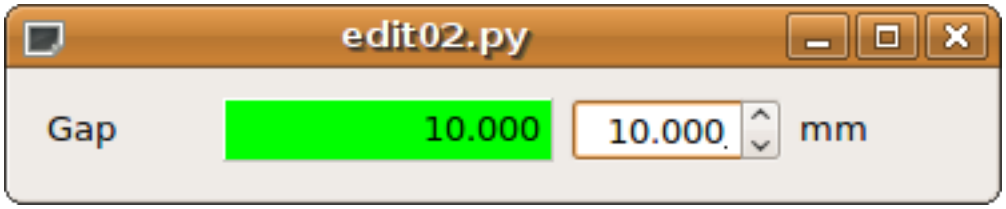


Fig. 1.24: With TaurusValueSpinBox

code:

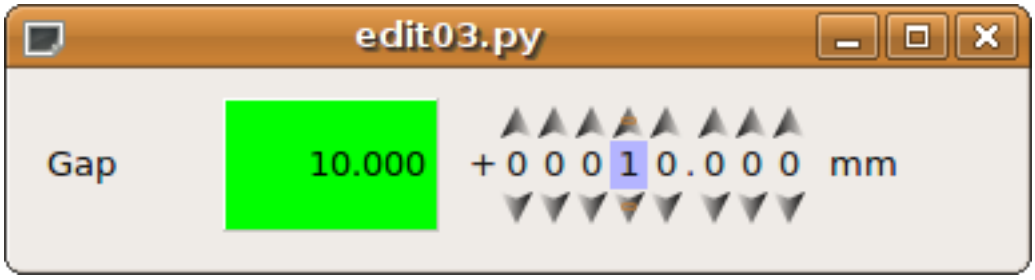


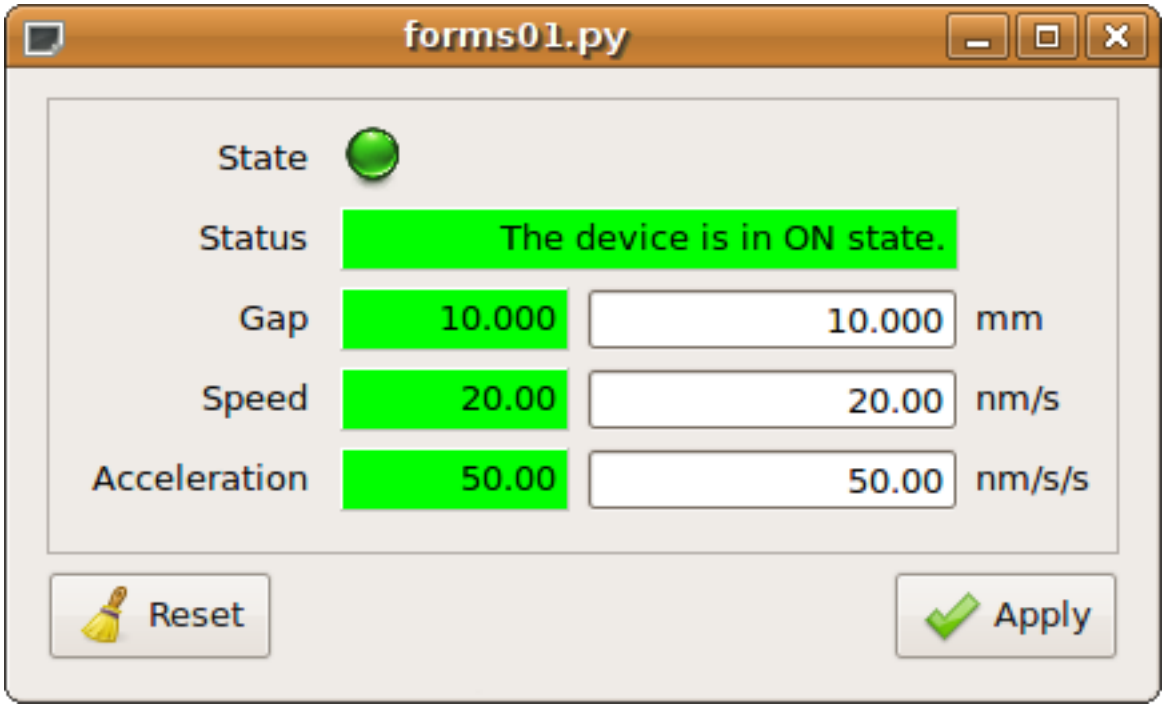
Fig. 1.25: With TaurusWheelEdit

```
from taurus.qt.qtgui.display import TaurusLabel
from taurus.qt.qtgui.input import TaurusValueLineEdit, TaurusValueSpinBox,␣
↪TaurusWheelEdit

w1 = TaurusLabel()
w2 = TaurusLabel()
w3 = TaurusValueLineEdit() # or TaurusValueSpinBox or TaurusWheelEdit
w4 = TaurusLabel()
layout.addWidget(w1)
layout.addWidget(w2)
layout.addWidget(w3)
layout.addWidget(w4)
w1.model, w1.bgRole = 'sys/taurustest/1/position#label', ''
w2.model = 'sys/taurustest/1/position'
w3.model = 'sys/taurustest/1/position'
w4.model, w4.bgRole = 'sys/taurustest/1/position#rvalue.units', ''
```

Now it seems a little bit more useful, doesn't it?

A higher level of abstraction: forms

Now let's say you want to display not only one but a dozen attributes... the programming becomes quite tedious. Taurus provides a higher level of abstraction: the *panel.TaurusForm*.



code:

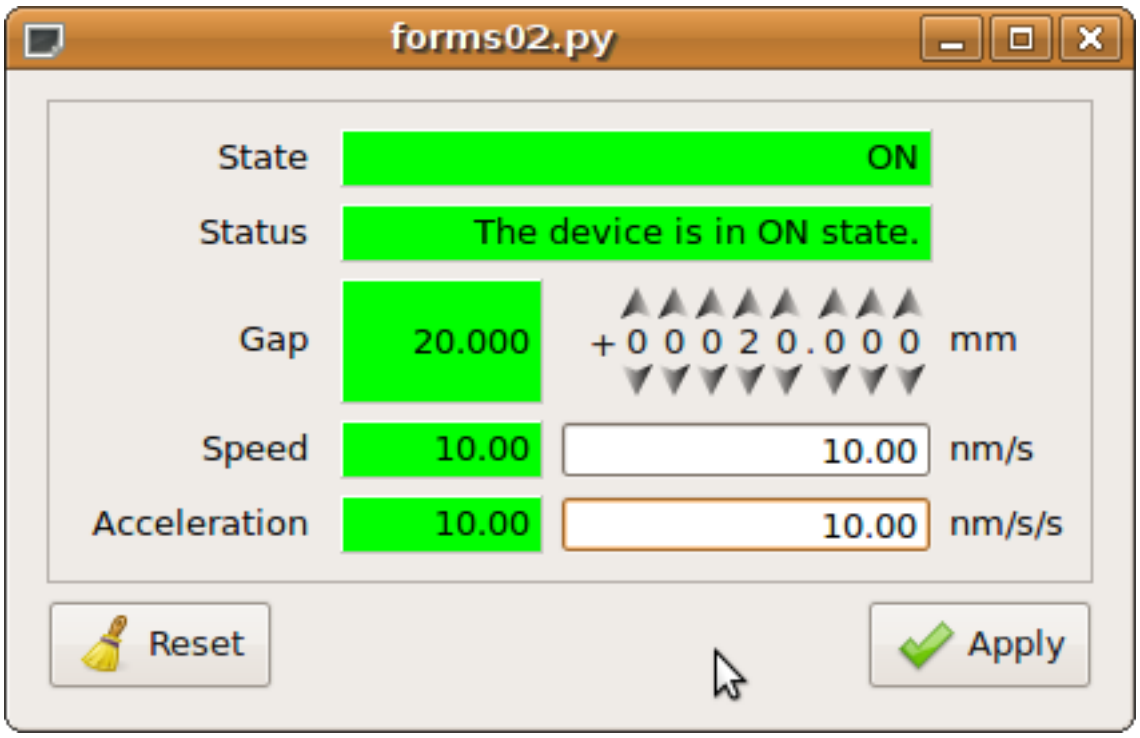
```
from taurus.qt.qtgui.panel import TaurusForm

panel = TaurusForm()
props = [ 'state', 'status', 'position', 'velocity', 'acceleration' ]
model = [ 'sys/taurustest/1/%s' % p for p in props ]
panel.setModel(model)
```

...and don't worry: *panel.TaurusForm* properly aligns the labels, manages the apply buttons and most important, it automagically decides which are the most appropriate widgets to use depending on the kind of attribute (you do not need to worry about whether the attribute is a scalar or a spectrum; or if it is read-only or writable; a boolean or a float, etc).

I specially enjoyed this one... let's see what's next!

Customizing forms

TaurusForm is highly customizable. This example shows how you can change the default widget for some attributes according to the user needs.



code:

```
from taurus.qt.qtgui.panel import TaurusForm
from taurus.qt.qtgui.display import TaurusLabel

panel = TaurusForm()
props = [ 'state', 'status', 'position', 'velocity', 'acceleration' ]
model = [ 'sys/taurustest/1/%s' % p for p in props ]
panel.setModel(model)
panel[0].readWidgetClass = TaurusLabel          # you can provide an arbitrary class...
panel[2].writeWidgetClass = 'TaurusWheelEdit'   # ...or, if it is a Taurus class you
↪can just give its name
```

A little configuration goes a long way!

Synoptics one-o-one

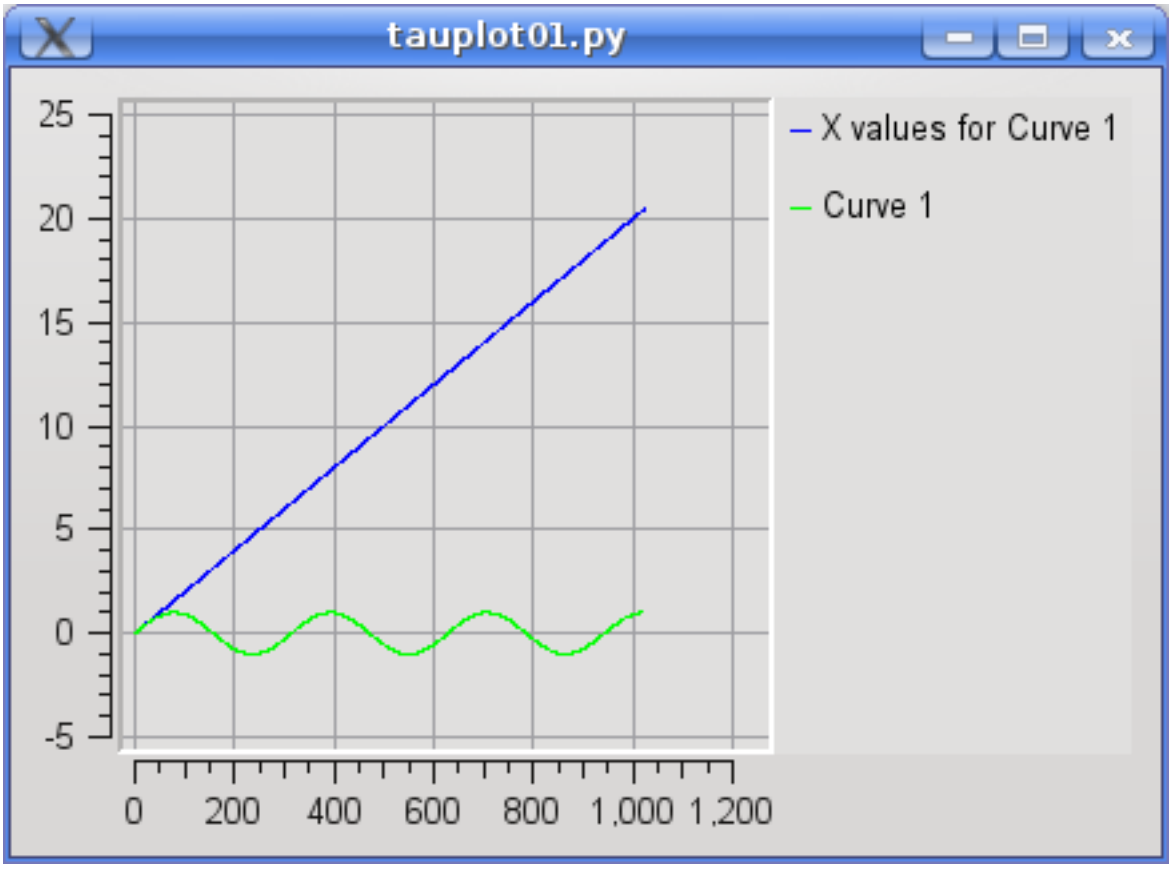
Todo

put a jdraw synoptics here

Let's go graphical

Simple plotting of various spectrum attributes

Say you want to plot two SPECTRUM attributes and watch them changing on-line? Taurus provides a very complete widget: *plot.TaurusPlot* (which makes use of the PyQwt library) .



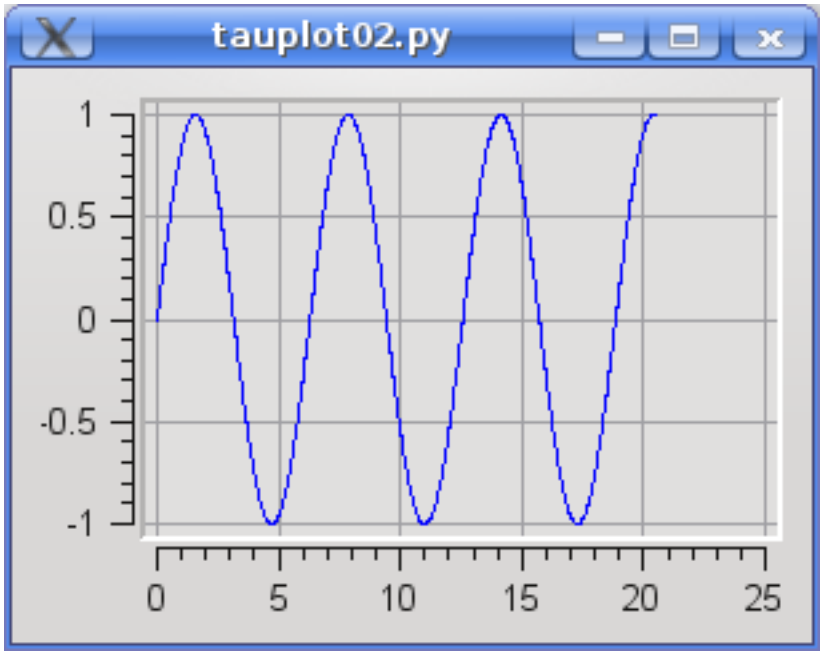
code:

```
from taurus.qt.qtgui.plot import TaurusPlot

panel = TaurusPlot()
model = ['sys/taurustest/1/abscissas', 'sys/taurustest/1/curve']
panel.setModel(model)
```

Scatter plots (Y vs X plots)

In the former example each element of the spectrum attributes, was assigned its position index as the x-value (i.e., the "abscissas" attribute was plotted as a spectrum). But, what if you want to create a scatter plot where you want to read the x values from one attribute and the y-values from another?

Solution: you use *xValuesAttrName*|*yValuesAttrName* as a member of the models list.



code:

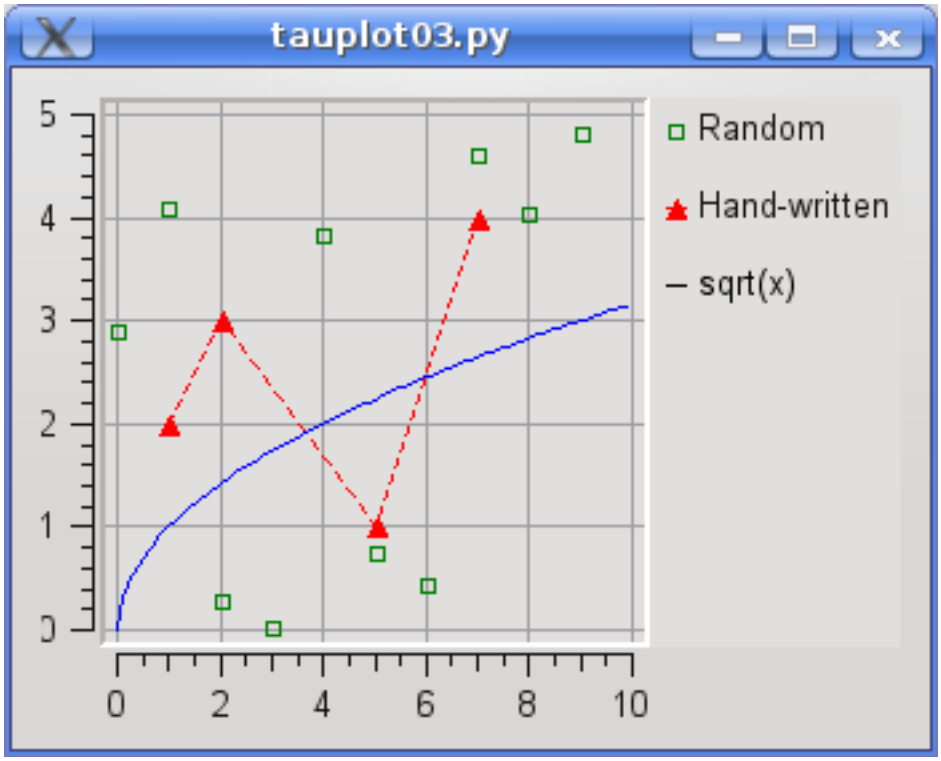
```
from taurus.qt.qtgui.plot import TaurusPlot

panel = TaurusPlot()
model = ['sys/taurustest/1/abscissas|sys/taurustest/1/curve']
panel.setModel(model)
```

Note that now the *sys/taurustest/1/abscissas* attribute is being used as x-values instead of being considered as another spectrum to plot like before.

Plotting data that is not an attribute

You are *not* limited to plotting data from Tango attributes. With *plot.TaurusPlot* you can also include arbitrary points (or even functions) in the plot.

Oh, and you can can change the display properties of any curve:



code:

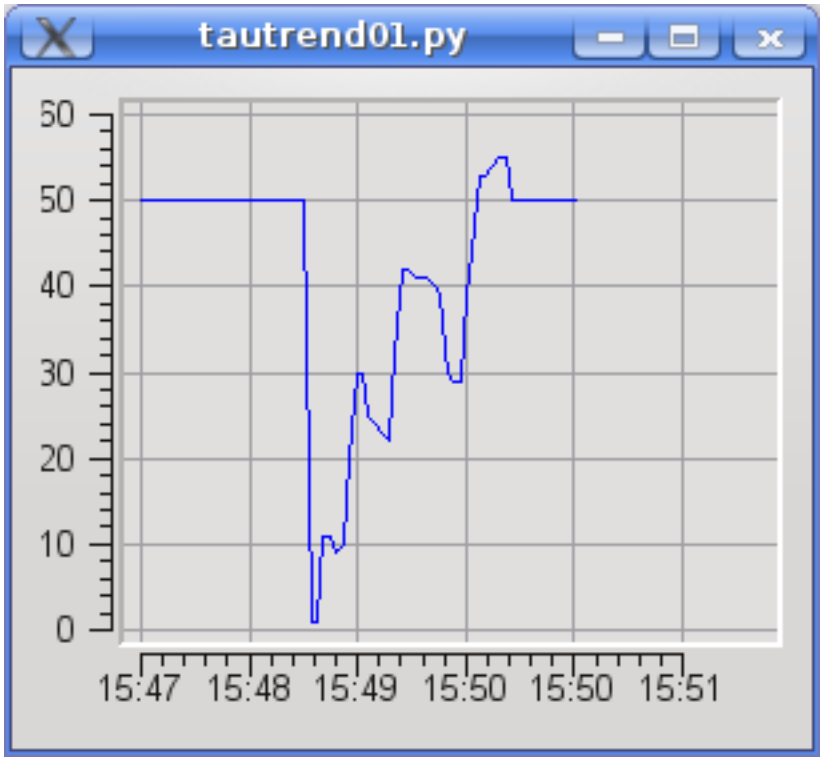
```
import numpy
from taurus.qt import Qwt5
from taurus.qt.qtgui.plot import TaurusPlot, CurveAppearanceProperties

panel = TaurusPlot()

rawdata1 = {"y":5*numpy.random.random(10), "name":"Random"}
rawdata2 = {"x":[1, 2, 5, 7], "y":[2, 3, 1, 4], "name":"Hand-written"}
rawdata3 = {"x":numpy.arange(0,10,0.1), "f(x)":"sqrt(x)"}

p1 = CurveAppearanceProperties(sStyle=Qwt5.QwtSymbol.Rect,
                               sSize=5,
                               sColor="green",
                               sFill=False,
                               lStyle=Qt.Qt.NoPen)

p2 = CurveAppearanceProperties(sStyle=Qwt5.QwtSymbol.Triangle,
                               sSize=8,
                               sColor="red",
                               sFill=True,
                               lColor="red",
                               lStyle=Qt.Qt.DashLine)

panel.attachRawData(rawdata1, properties=p1)
panel.attachRawData(rawdata2, properties=p2)
panel.attachRawData(rawdata3)
```

...note the third curve: its definition is just a string defining a mathematical formula!

TaurusPlot knows maths!

Plotting Trends

Many times we are interested in showing how a scalar attribute evolves with time. A close-cousin of the TaurusPlot called *plot.TaurusTrend* is here to help you:



code:

```
from taurus.qt.qtgui.plot import TaurusTrend

panel = TaurusTrend()
model = ['sys/taurustest/1/position']
panel.setXIsTime(True) #to show the x values as time
panel.setModel(model)
```

Note: if you pass a model that is a Tango SPECTRUM attribute (instead of a scalar), TaurusTrend will interpret it as a collection of scalar values and will plot a separate trend line for each.

Even higher level: creating a TaurusGui

taurusgui.TaurusGui provides very convenient way of creating feature-rich and very configurable GUIs by using existing widgets as "panels". TaurusGuis can be created via a wizard application (no programming at all!) with a few clicks. You can try it out by running:

```
taurusgui --new-gui
```

For more details and tricks regarding TaurusGui, check *this*.

taurus

The main taurus module. It contains a reduced set of wrappers around the real taurus model classes and information regarding the current release.

Modules

taurus.console

Modules

taurus.console.util

taurus.core

The core module

Modules

taurus.core.epics

Epics extension for taurus core model.

The epics extension provides access to Epics control system objects via Channel Access

---

Note: The Epics scheme is only a proof of concept. The final syntax of the model names is not yet set in stone and only basic functionality is implemented.

The Epics extension implements *taurus.core* objects that connect to Epics PVs. The scheme name for channel access epics models is 'ca' ('epics' also works at this moment).

You should never instantiate models from epics model classes directly. Instead, use the `taurus.core.taurusmanager.TaurusManager` and `taurus.core.taurusmanager.TaurusFactory` APIs to access all elements.

For example, to get a reference to the epics process variable (PV) "my:example.RBV" you should do something like:

```
>>> import taurus
>>> myattr = taurus.Attribute('ca:my:example.RBV')
```

Epics attributes (should) work just as other Taurus attributes and can be referred by their model name wherever a Taurus Attribute model is expected. For example, you can launch a *TaurusForm* with an epics attribute:

```
$> taurusform ca:my:example
```

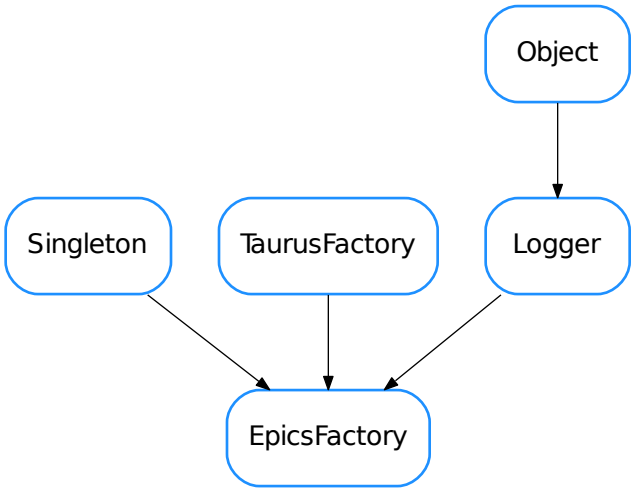
Similarly, you can combine epics attributes with attributes from other schemes:

```
$> taurusform 'ca:my:example' 'tango:sys/tg_test/1/float_scalar'       'eval:
↪{ca:my:example}*{tango:sys/tg_test/1/float_scalar}'
```

Currently, the taurus epics scheme just supports epics PVs, implementing them as taurus attributes. Other model types such as the Authority, and Device classes are just convenience dummy objects in the epics scheme at this point. Epics records may eventually be mapped as Devices.

Classes

EpicsFactory



class EpicsFactory

Bases: taurus.core.util.singleton.Singleton, taurus.core.taurusfactory.TaurusFactory, taurus.core.util.log.Logger

A Singleton class that provides Epics related objects.

DEFAULT_AUTHORITY = 'ca://'

DEFAULT_DEVICE = 'ca:'

caseSensitive = False

elementTypesMap = {18: <class 'taurus.core.epics.epicsauthority.EpicsAuthority'>, 3: <class 'taurus.core.epics.epicsde

getAttribute(*attr_name*)

Obtain the object corresponding to the given attribute name. If the corresponding attribute already exists, the existing instance is returned. Otherwise a new instance is stored and returned. The device associated to this attribute will also be created if necessary.

Parameters **attr_name** (str) – the attribute name string. See *taurus.core.epics* for valid attribute names

Return type EpicsAttribute

Returns

Raise

TaurusException if the given name is invalid.

getAttributeNameValidator()
Return EpicsAttributeNameValidator

getAuthority(*name=None*)
Obtain the Epics (ca) authority object.

Parameters **name** (str) – only a dummy authority (“ca://”) is supported

Return type EpicsAuthority

Returns

getAuthorityNameValidator()
Return EpicsAuthorityNameValidator

getDevice(*dev_name*)
Obtain the EpicsDevice object.

Parameters **dev_name** (str) – only one dummy device (“”) is supported

Return type EpicsDevice

Returns

Todo

epics.Device may be wrapped as taurus device...

---

getDeviceNameValidator()
Return EpicsDeviceNameValidator

init(**args*, ***kwargs*)
Singleton instance initialization.

schemes = (‘ca’, ‘epics’)

- *EpicsFactory*

taurus.core.evaluation

Evaluation extension for taurus core model.

The evaluation extension is a special extension that provides evaluation objects. The official scheme name is ‘eval’.

The main usage for this extension is to provide a way of performing mathematical evaluations with data from other source, as well as allowing fast interfacing with sources of data not supported by specific schemes.

The Evaluation Factory (*EvaluationFactory*) uses the following object naming for referring to attributes (*EvaluationAttribute*):

eval:[//<authority>][@<evaluator>/][<subst>;]<expr>

or the following for referring to evaluation devices (*EvaluationDevice*):

eval:[//<authority>]@<evaluator>

or the following for referring to an evaluation authority (*EvaluationAuthority*):

eval://<authority>

where:

- The <*authority*> segment is optional (except when referring to an EvaluationDatabase). At this point, only *//localhost* is supported.
- The @<*evaluator*> is optional (except when referring to an EvaluationDevice). If not given, it defaults to *DefaultEvaluator*. See below for further details
- <*expr*> is a mathematical expression (using python syntax) that may have references to other taurus **attributes** by enclosing them between *{* and *}*. Expressions will be evaluated by the evaluator device to which the attribute is assigned.
- The optional <*subst*> segment is used to provide substitution symbols. <*subst*> is a semicolon-separated string of <*key*>=<*value*> strings.

The evaluator device inherits from `SafeEvaluator` which by default includes a large subset of mathematical functions from the `numpy` module. If access to other symbols are required, a custom evaluator can be used. <*evaluator*> is a unique identification name for the evaluator device object and may define a source of additional symbols to be present for the evaluation. The supported syntax for @<*evaluator*> is:

- @<*ID*> (cannot contain dots or any of */ ? # : =*). This indicates just an alternative name for the EvaluationDevice, It does not add any extra symbol to the evaluation context.
- @<*modulename*>.* (<modulename> may include dots for submodules). It will make all symbols found in the given module available during the evaluation (i.e., it emulates doing *from* <*modulename*> *import* * in the evaluation context).
- @<*modulename*>.<*customdeviceclass*>. Use your own custom EvaluationDevice based class. This allows to define custom symbols see `<taurus>/core/evaluation/test/res/dev_example.py`, **but** note that this syntax is is now superseded by the "instance-based" one (see below), which is easier to use and provides write attribute support.
- @<*inst*>=<*modulename*>.<*class*>*()* (e.g. *@c=mymod.MyClass()*). This will import a class from a module, then instantiate it and then make the instance available for evaluation with the given name. Note that the <*inst*>= part may be omitted, in which case the instance will be available for evaluation as *self*. **IMPORTANT:** If the given class declares writable properties, EvaluationAttributes that access one such property will automatically be considered writable. See examples of usage in `<taurus>/core/evaluation/test/res/mymod.py` and in `<taurus>/core/evaluation/test/res/ipap_example.py`

Some examples of valid evaluation models are:

- An attribute that multiplies a tango attribute by 2:

  eval:2{tango:a/b/c/d}*

- Same as above, but using substitutions:

  *eval:k=2;a={tango:a/b/c/d};k*a*

- An attribute that adds two tango attributes together (assuming that tango is set as the default scheme)

  eval:{a/b/c/d}+{f/g/h/i}

- An attribute that generates an array of random values:

  eval:rand(256)

- Same as above, but with units:

  eval:Q(rand(256),'V')

- An attribute that adds noise to a tango image attribute:

  *eval:img={tango:sys/tg_test/1/short_image_ro};img+10*rand(*img.shape)*

- An attribute that accesses a method from a given module (in this case to use os.path.exists):

eval:@os./path.exists("/some/file")*

- Same as before, for getting today's date as an attribute:

  'eval:@datetime.*/date.today().isoformat()'

- A default evaluator device named *foo*:

  eval:@foo

- A custom evaluator device (implemented as class *MyEvalDev* in the *mymod* module):

  eval:@mymod.MyEvalDev

- A custom evaluator device (implemented as class *MyEvalDev* in the *mymod* module):

  eval:@mymod.MyEvalDev

- A writable attribute foo (implemented as a writable property of the *MyClass* class from the *mymod* module):

  eval:@c=mymod.MyClass()/c.foo

  assuming that the *mymod* module defines *MyClass* as:

  class MyClass(object): (...) get_foo(self):

  (...)

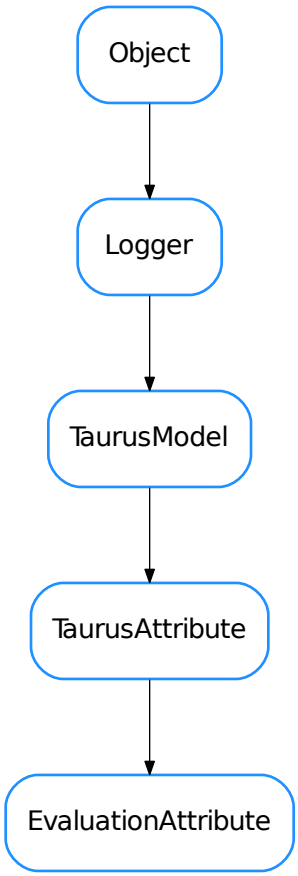
  set_foo(self, value): (...)

  foo = property(get_foo, set_foo) (...)

Note: Previous to SEP3, a RFC3986 non-compliant syntax was used for the evaluation scheme (e.g., allowing names such as `tango://db=foo;dev=bar;a*b?k=2;a={tango:a/b/c/d}`). This syntax is now deprecated and should not be used. Taurus will issue warnings if detected.

Classes

EvaluationAttribute



class EvaluationAttribute (*name*, *parent*, ***kwargs*)

Bases: `taurus.core.taurusattribute.TaurusAttribute`

A `TaurusAttribute` that can be used to perform mathematical operations involving other arbitrary Taurus attributes. The mathematical operation is described in the attribute name itself. An Evaluation Attribute will keep references to any other attributes being referenced and it will update its own value whenever any of the referenced attributes change.

See also:

taurus.core.evaluation

> **Warning:** In most cases this class should not be instantiated directly. Instead it should be done via the *EvaluationFactory.getAttribute()*

addListener (*listener*)

Add a TaurusListener object in the listeners list. If it is the first listener, it triggers the subscription to the

referenced attributes. If the listener is already registered nothing happens.

applyTransformation()

decode(*attr_value*)

encode(*value*)

eventReceived(*evt_src*, *evt_type*, *evt_value*)

getDisplayValue(*cache=True*)

static **getId**(*obj*, *idFormat='_V%i_'*)

returns an id string for the given object which has the following

two properties:

- It is unique for this object during all its life
- It is a string that can be used as a variable or method name

Parameters

- **obj** (object) – the python object whose id is requested
- **idFormat** (str) – a format string containing a "*%i*" which, when expanded must be a valid variable name (i.e. it must match *[a-zA-Z_][a-zA-Z0-9_]**). The default is *_V%i_*

isBoolean()

isUsingEvents()

poll()

preProcessTransformation(*trstring*)

parses the transformation string and creates the necessary symbols for the evaluator. It also connects any referenced attributes so that the transformation gets re-evaluated if they change.

Parameters **trstring** (str) – a string to be pre-processed

Return type tuple <str, bool>

Returns a tuple containing the processed string and a boolean indicating if the preprocessing was successful. if ok==True, the string is ready to be evaluated

read(*cache=True*)

returns the value of the attribute.

Parameters **cache** (bool) – If True (default), the last calculated value will be returned. If False, the referenced values will be re- read and the transformation string will be re-evaluated

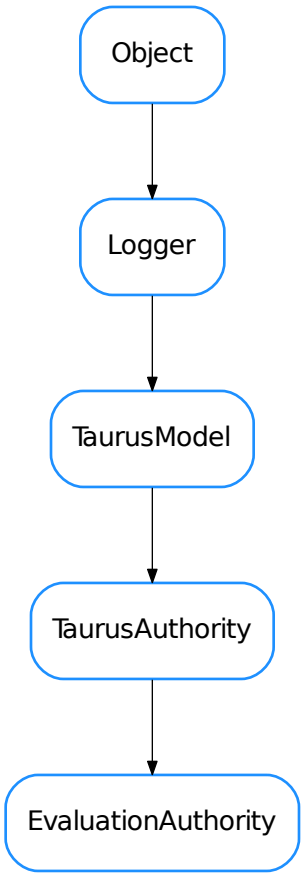
Returns attribute value

removeListener(*listener*)

Remove a TaurusListener from the listeners list. If polling enabled and it is the last element then stop the polling timer. If the listener is not registered nothing happens.

write(*value*, *with_read=True*)

EvaluationAuthority



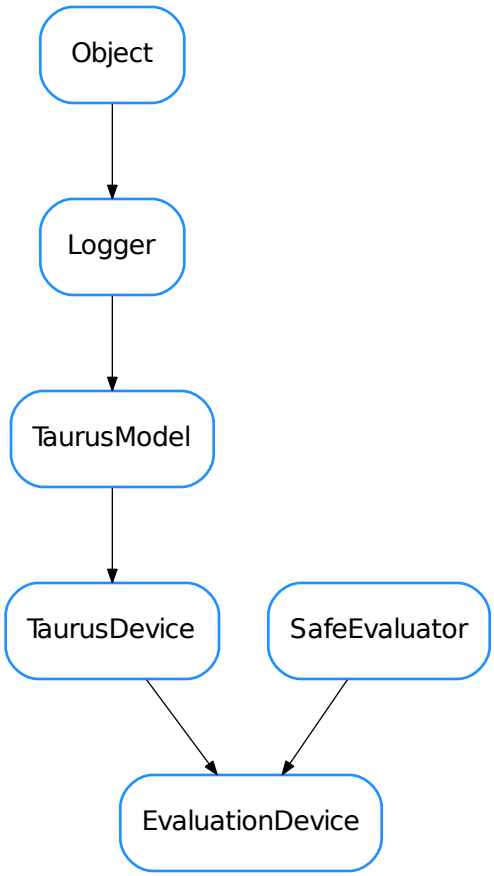
class **EvaluationAuthority**(*complete_name*, *parent=None*)

Bases: `taurus.core.taurusauthority.TaurusAuthority`

Dummy authority class for Evaluation (the authority concept is not yet used in the Evaluation scheme)

> **Warning:** In most cases this class should not be instantiated directly. Instead it should be done via the *EvaluationFactory.getAuthority()*

EvaluationDevice



class EvaluationDevice(*name*, ***kw*)

Bases: `taurus.core.taurusdevice.TaurusDevice`, `taurus.core.util.safeeval.SafeEvaluator`

The evaluator object. It is a `TaurusDevice` and is used as the parent of *`EvaluationAttribute`* objects for which it performs the mathematical evaluation.

See also:

`taurus.core.evaluation`

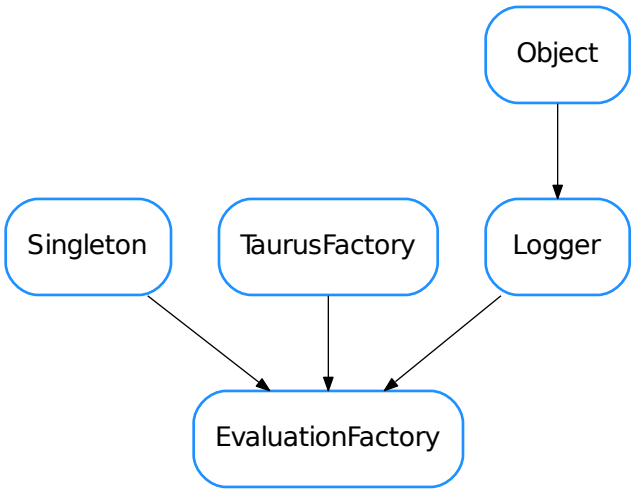
> **Warning:** In most cases this class should not be instantiated directly. Instead it should be done via the *`EvaluationFactory.getDevice()`*

decode(*event_value*)

getAttribute(*attrname*)

Returns the attribute object given its name

EvaluationFactory



class EvaluationFactory

Bases: taurus.core.util.singleton.Singleton, taurus.core.taurusfactory.TaurusFactory, taurus.core.util.log.Logger

A Singleton class that provides Evaluation related objects.

DEFAULT_AUTHORITY = '//localhost'

DEFAULT_DATABASE = '_DefaultEvalDB'

DEFAULT_DEVICE = '@DefaultEvaluator'

elementTypesMap = {18: <class 'taurus.core.evaluation.evalauthority.EvaluationAuthority'>, 3: <class 'taurus.core.eva

findObjectClass(*absolute_name*)

Operation models are always OperationAttributes

getAttribute(*attr_name*, ***kwargs*)

Obtain the object corresponding to the given attribute name. If the corresponding attribute already exists, the existing instance is returned. Otherwise a new instance is stored and returned. The evaluator device associated to this attribute will also be created if necessary.

Parameters **attr_name** (str) – the attribute name string. See *taurus.core.evaluation* for valid attribute names

Any aditional keyword arguments will be passed directly to the constructor of *:class:EvaluationAttribute*

Return type *EvaluationAttribute*

Returns

@throws TaurusException if the given name is invalid.

getAttributeNameValidator()

Return EvaluationAttributeNameValidator

getAuthority(*name=None*)
Obtain the EvaluationDatabase object.

Parameters **db_name** (str) – this is ignored because only one database is supported

Return type EvaluationDatabase

Returns

getAuthorityNameValidator()
Return EvaluationAuthorityNameValidator

getDevice(*dev_name*)
Obtain the object corresponding to the given device name. If the corresponding device already exists, the existing instance is returned. Otherwise a new instance is stored and returned.

Parameters **dev_name** (str) – the device name string. See *taurus.core.evaluation* for valid device names

Return type *EvaluationDevice*

Returns

@throws TaurusException if the given name is invalid.

getDeviceNameValidator()
Return EvaluationDeviceNameValidator

init(**args*, ***kwargs*)
Singleton instance initialization.

schemes = ('eval', 'evaluation')

- *EvaluationAttribute*
- *EvaluationAuthority*
- *EvaluationDevice*
- *EvaluationFactory*

taurus.core.resource

Resource scheme extension for taurus core mode. The resource extension is a special extension that acts like a name map for actual model names.

This allows to indirect the hardcoded model names of your application and keep the actual specific model names grouped in one place for better portability and reusability of your application.

The scheme name is 'res'. The map can be implemented either as python modules or as dicts (see below).

The main class for the extension module is *ResourcesFactory* and you can add resource maps with *ResourcesFactory.loadResource()*.

By default, the *ResourcesFactory* will always try to load a resource module named 'taurus_resources' from the application directory (so you can create a file called *taurus_resources.py* in your application directory and skip the step of calling *ResourcesFactory.loadResource()*.

Mapping implemented as python modules

If a resource is a python module, the factory will use its global variables as the resource keys. The variable value, which must be a string will be used as the mapped model. For example, if the *taurus_resources.py* file in the application directory contains the following definitions:

```
my_device = 'tango:a/b/c'
my_state = my_device + '/state'
```

Then, in your code, you can access the Device and Attribute objects by doing:

```
>>> import taurus
>>> my_device_obj = taurus.Device('res:my_device')
>>> my_state_obj = taurus.Attribute('res:my_state')
```

Note that you can use python code to automate the contents of the module. Example:

```
base = 'my/motor/'
g = globals()

for i in xrange(256):
    i_str = str(i)
    g['mym"+i_str] = base + i_str
```

Mapping implemented as dictionaries

Dictionaries can also be registered (as an alternative to modules) as resource maps:

```
>>> d = {'my_device':'tango:a/b/c', 'my_state':'tango:a/b/c/state'}
>>> import taurus
>>> factory = taurus.Factory('res')
>>> factory.loadResource(d)
>>> my_device_obj = taurus.Device('res:my_device')
>>> my_state_obj = taurus.Attribute('res:my_state')
```

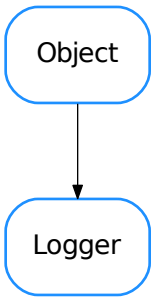
Note: for coherence with the file mapping, valid key names are restricted to valid variable names (i.e. to the following regexp: *[a-zA-Z_][a-zA-Z0-9_]**

Important: Models object type is the mapped type

The model objects returned by the *ResourcesFactory* will be of the mapped model type. In other words: if the model name 'eval:Q(rand())' is mapped to the *myattr* resource name, calling *taurus.Attribute('res:myattr')* will return a `EvaluationAttribute`, not a *ResAttribute* (*ResAttribute* is not even defined).

Classes

Logger



class Logger (*name=''*, *parent=None*, *format=None*)
Bases: taurus.core.util.object.Object

The taurus logger class. All taurus pertinent classes should inherit directly or indirectly from this class if they need taurus logging facilities.

Critical = 50
Critical message level (constant)

Debug = 10
Debug message level (constant)

DftLogFormat = <logging.Formatter object>
Default log format (constant)

DftLogLevel = 20
Default log level (constant)

DftLogMessageFormat = '%(threadName)-14s %(levelname)-8s %(asctime)s %(name)s: %(message)s'
Default log message format (constant)

Error = 40
Error message level (constant)

Fatal = 50
Fatal message level (constant)

Info = 20
Info message level (constant)

Trace = 5
Trace message level (constant)

Warning = 30
Warning message level (constant)

addChild (*child*)
Adds a new logging child

Parameters **child** (Logger) – the new child

classmethod addLevelName(*level_no*, *level_name*)
Registers a new log level

Parameters

- **level_no** (int) – the level number
- **level_name** (str) – the corresponding name

addLogHandler(*handler*)
Registers a new handler in this object's logger

Parameters **handler** (Handler) – the new handler to be added

classmethod addRootLogHandler(*h*)
Adds a new handler to the root logger

Parameters **h** (Handler) – the new log handler

changeLogName(*name*)
Change the log name for this object.

Parameters **name** (str) – the new log name

cleanUp()
The cleanUp. Default implementation does nothing Overwrite when necessary

copyLogHandlers(*other*)
Copies the log handlers of other object to this object

Parameters **other** (object) – object which contains 'log_handlers'

critical(*msg*, **args*, ***kw*)
Record a critical message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.critical().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

debug(*msg*, **args*, ***kw*)
Record a debug message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.debug().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

deprecated(*msg=None*, *dep=None*, *alt=None*, *rel=None*, *dbg_msg=None*, *_callerinfo=None*, ***kw*)
Record a deprecated warning message in this object's logger. If message is not passed, a estandard deprecation message is constructued using dep, alt, rel arguments. Also, an extra debug message can be recorded, followed by traceback info.

Parameters

- **msg** (str) – the message to be recorded (if None passed, it will be constructed using dep (and, optionally, alt and rel)
- **dep** (str) – name of deprecated feature (in case msg is None)

- **alt** (str) – name of alternative feature (in case msg is None)
- **rel** (str) – name of release from which the feature was deprecated (in case msg is None)
- **dbg_msg** (str) – msg for debug (or None to log only the warning)
- **_callerinfo** – for internal use only. Do not use this argument.
- **kw** – any additional keyword arguments, are passed to `logging.Logger.warning()`

classmethod disableLogOutput()
Disables the `logging.StreamHandler` which dumps log records, by default, to the stderr.

classmethod enableLogOutput()
Enables the `logging.StreamHandler` which dumps log records, by default, to the stderr.

error(*msg*, **args*, ***kw*)
Record an error message in this object's logger. Accepted *args* and *kwargs* are the same as `logging.Logger.error()`.

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

exception(*msg*, **args*)
Log a message with severity 'ERROR' on the root logger, with exception information.. Accepted *args* are the same as `logging.Logger.exception()`.

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments

fatal(*msg*, **args*, ***kw*)
Record a fatal message in this object's logger. Accepted *args* and *kwargs* are the same as `logging.Logger.fatal()`.

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

flushOutput()
Flushes the log output

getChildren()
Returns the log children for this object

Return type Logger

Returns the list of log children

classmethod getLogFormat()
Retuns the current log message format (the root log format)

Return type str

Returns the log message format

getLogFullName()
Gets the full log name for this object

Return type str

Returns the full log name

classmethod getLogLevel()
Retuns the current log level (the root log level)

Return type int

Returns a number representing the log level

getLogName()
Gets the log name for this object

Return type str

Returns the log name

getLogObj()
Returns the log object for this object

Return type Logger

Returns the log object

classmethod getLogger(*name=None*)

getParent()
Returns the log parent for this object or None if no parent exists

Return type Logger or None

Returns the log parent for this object

classmethod getRootLog()
Retuns the root logger

Return type Logger

Returns the root logger

info(*msg*, **args*, ***kw*)
Record an info message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.info().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

classmethod initRoot()
Class method to initialize the root logger. Do **NOT** call this method directly in your code

log(*level*, *msg*, **args*, ***kw*)
Record a log message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.log().

Parameters

- **level** (int) – the record level
- **msg** (str) – the message to be recorded

- **args** – list of arguments
- **kw** – list of keyword arguments

log_format = <logging.Formatter object>
Default log message format

log_level = 20
Current global log level

classmethod removeRootLogHandler(*h*)
Removes the given handler from the root logger

Parameters **h** (Handler) – the handler to be removed

classmethod resetLogFormat()
Resets the log message format (the root log format)

classmethod resetLogLevel()
Resets the log level (the root log level)

root_init_lock = <thread.lock object>
Internal usage

root_inited = True
Internal usage

classmethod setLogFormat(*format*)
sets the new log message format

Parameters **level** (str) – the new log message format

classmethod setLogLevel(*level*)
sets the new log level (the root log level)

Parameters **level** (int) – the new log level

stack(*target=5*)
Log the usual stack information, followed by a listing of all the local variables in each frame.

Parameters **target** (int) – the log level assigned to the record

Return type str

Returns The stack string representation

stream_handler = <logging.StreamHandler object>
the main stream handler

syncLog()
Synchronises the log output

trace(*msg*, **args*, ***kw*)
Record a trace message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.log().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

traceback(*level=5*, *extended=True*)
Log the usual traceback information, followed by a listing of all the local variables in each frame.

Parameters

- **level** (int) – the log level assigned to the traceback record
- **extended** (bool) – if True, the log record message will have multiple lines

Return type str

Returns The traceback string representation

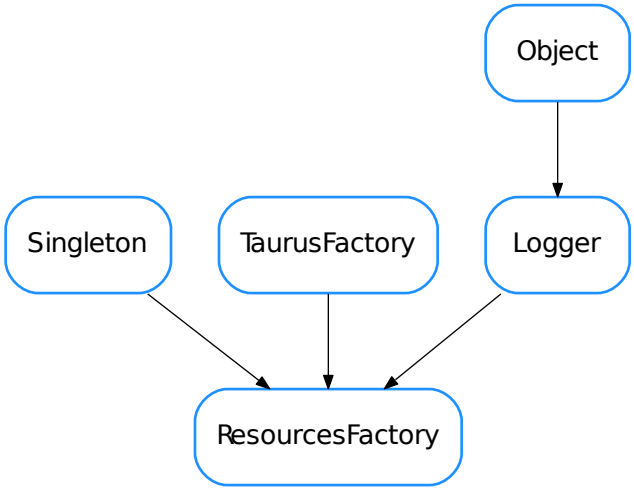
warning(*msg*, **args*, ***kw*)

Record a warning message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.warning().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

ResourcesFactory



class ResourcesFactory

Bases: taurus.core.util.singleton.Singleton, taurus.core.taurusfactory.TaurusFactory, taurus.core.util.log.Logger

A Singleton class designed to provide Simulation related objects.

DftResourceName = 'taurus_resources.py'

the default resource file name

DftResourcePriority = 10

priority for the default resource

clear()

findObjectClass(*absolute_name*)
Obtain the class object corresponding to the given name.

Parameters **absolute_name** (str) – the object absolute name string

Return type TaurusModel or None

Returns the class for the model object mapped by absolute_name, or None if absolute_name is invalid.

getAttribute(*name*)
Obtain the attribute model object referenced by name.

Parameters **name** (str) – name

Return type TaurusAttribute

Returns attribute object

Raise (TaurusException) if name is invalid

getAttributeNameValidator()
Return ResourceAttributeNameValidator

getAuthority(*name=None*)
Obtain the authority model object referenced by name.

Parameters **name** (str) – name

Return type TaurusAuthority

Returns authority object

Raise (TaurusException) if name is invalid

getAuthorityNameValidator()
Return ResourceAuthorityNameValidator

getDevice(*name*)
Obtain the device model object referenced by name.

Parameters **name** (str) – name

Return type TaurusDevice

Returns device object

Raise (TaurusException) if name is invalid

getDeviceNameValidator()
Return ResourceDeviceNameValidator

getValue(*key*)
Returns the value for a given key

Parameters **key** (str) – a key

Return type str

Returns the value for the given key

init(**args*, ***kwargs*)
Singleton instance initialization. **For internal usage only**

loadResource(*obj=None*, *priority=1*, *name=None*)
(Re)Loads the given resource.

Parameters

- **obj** (dict or file or None) – the resource object. Default is None meaning in will (re)load the default resource: taurus_resources.py from the application directory
- **priority** (int) – the resource priority. Default is 1 meaning maximum priority
- **name** (str) – an optional name to give to the resource

Return type dict

Returns a dictionary version of the given resource object

reloadResource(*obj=None*, *priority=1*, *name=None*)
(Re)Loads the given resource.

Parameters

- **obj** (dict or file or None) – the resource object. Default is None meaning in will (re)load the default resource: taurus_resources.py from the application directory
- **priority** (int) – the resource priority. Default is 1 meaning maximum priority
- **name** (str) – an optional name to give to the resource

Return type dict

Returns a dictionary version of the given resource object

schemes = ('res', 'resource')
the list of schemes that this factory supports. For this factory: 'res' and 'resources' are the supported schemes

Singleton

class Singleton
Bases: object

This class allows Singleton objects The __new__ method is overriden to force Singleton behaviour. The Singleton is created for the lowest subclass. Usage:

```
from taurus.core.util.singleton import Singleton

class MyManager(Singleton):

    def init(self, *args, **kwargs):
        print "Singleton initialization"
```

command line:

```
>>> manager1 = MyManager()
Singleton initialization

>>> manager2 = MyManager()

>>> print(manager1,manager2)
<__main__.MyManager object at 0x9c2a0ec>
<__main__.MyManager object at 0x9c2a0ec>
```

Notice that the two instances of manager point to the same object even though you *tried* to construct two instances of MyManager.

> **Warning:** although __new__ is overriden __init__ is still being called for each instance=Singleton()

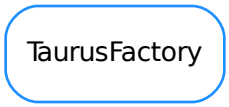
init (**p*, ***k*)

TaurusException

class TaurusException (*description*, *code=None*)

Bases: exceptions.Exception

TaurusFactory



class TaurusFactory

Bases: object

The base class for valid Factories in Taurus.

DefaultPollingPeriod = 3000

addAttributeToPolling(*attribute*, *period*, *unsubscribe_evts=False*)
Activates the polling (client side) for the given attribute with the given period (seconds).

Parameters

- **attribute** (*TangoAttribute*) – attribute name.
- **period** (float) – polling period (in seconds)
- **unsubscribe_evts** (bool) – whether or not to unsubscribe from events

caseSensitive = True

changeDefaultPollingPeriod(*period*)

cleanUp()
Reimplement if you need to execute code on program execution exit. Default implementation does nothing.

disablePolling()
Disable the application tango polling

elementTypesMap = None

enablePolling()
Enable the application tango polling

findObject(*absolute_name*)
Must give an absolute name

findObjectClass(*absolute_name*)
Obtain the class object corresponding to the given name.

Note, this generic implementation expects that derived classes provide a an attribute called elementTypesMap consisting in a dictionary whose keys are TaurusElementTypes and whose values are the corresponding specific object classes. e.g., the FooFactory should provide:

```
class FooFactory(TaurusFactory):
    elementTypesMap = {TaurusElementType.Authority: FooAuthority,
                       TaurusElementType.Device: FooDevice,
                       TaurusElementType.Attribute: FooAttribute,
                       }
    (...)
```

Parameters **absolute_name** (str) – the object absolute name string

Return type TaurusModel or None

Returns a TaurusModel class derived type or None if the name is not valid

getAttribute(*name*)
Obtain the model object corresponding to the given attribute name. If the corresponding attribute already exists, the existing instance is returned. Otherwise a new instance is stored and returned.

Parameters **name** (str) – attribute name

Returns a taurus.core.taurusattribute.TaurusAttribute object

Raise

TaurusException if the given name is invalid.

getAttributeNameValidator()

getAuthority(*name=None*)
Obtain the model object corresponding to the given authority name. If the corresponding authority already exists, the existing instance is returned. Otherwise a new instance is stored and returned.

Parameters **name** (str) – authority name

Returns a taurus.core.taurusauthority.TaurusAuthority object

Raise

TaurusException if the given name is invalid.

getAuthorityNameValidator()

getDefaultPollingPeriod()

getDevice(*name*, ***kw*)
Obtain the model object corresponding to the given device name. If the corresponding device already exists, the existing instance is returned. Otherwise a new instance is stored and returned.

Parameters **name** (str) – device name

Returns a taurus.core.taurusdevice.TaurusDevice object

Raise

TaurusException if the given name is invalid.

getDeviceNameValidator()

getObject(*cls*, *name*)

getSerializationMode()
Gives the serialization operation mode.

Return type TaurusSerializationMode

Returns the current serialization mode

getValidTypesForName(*name*, *strict=None*)
Returns a list of all Taurus element types for which *name* is a valid model name (while in many cases a name may only be valid for one element type, this is not necessarily true in general)

In this base implementation, name is checked first for Attribute, then for Device and finally for Authority, and the return value is sorted in that same order.

If a given schema requires a different ordering, reimplement this method

Parameters **name** (str) – taurus model name

Return type list <element>

Returns where element can be one of: *Attribute*, *Device* or *Authority*

isPollingEnabled()
Tells if the Taurus polling is enabled

Return type bool

Returns whether or not the polling is enabled

registerAttributeClass(*attr_name*, *attr_klass*)

registerDeviceClass(*dev_klass_name*, *dev_klass*)

removeAttributeFromPolling(*attribute*)
Deactivate the polling (client side) for the given attribute. If the polling of the attribute was not previously enabled, nothing happens.

Parameters **attribute** (str) – attribute name.

schemes = ()

setSerializationMode(*mode*)
Sets the serialization mode for the system.

Parameters **mode** (TaurusSerializationMode) – the new serialization mode

supportsScheme(*scheme*)
Returns whether the given scheme is supported by this factory

Parameters **scheme** (str) – the name of the schem to be checked

Return type bool

Returns True if the scheme is supported (False otherwise)

unregisterAttributeClass(*attr_name*)

unregisterDeviceClass(*dev_klass_name*)

- *Logger*
- *ResourcesFactory*
- *Singleton*
- *TaurusException*
- *TaurusFactory*

Functions

Manager()
Returns the one and only TaurusManager

It is a shortcut to:

```
import taurus.core
manager = taurus.core.taurusmanager.TaurusManager()
```

Returns the TaurusManager

Return type taurus.core.taurusmanager.TaurusManager

See also:

taurus.core.taurusmanager.TaurusManager

taurus.core.tango

Tango extension for taurus core mode. The Tango extension implements *taurus.core* objects that connect to Tango objects. The official scheme name is, obviously, 'tango'.

This extension maps the (Py)Tango objects into Taurus objects as follows:

- A Tango database is represented as a subclass of *taurus.core.TaurusAuthority*
- A Tango device is represented as a subclass of *taurus.core.TaurusDevice*
- A Tango device attribute is represented as a subclass *taurus.core.TaurusAttribute*

You should never create objects from the above classes directly. Instead, you should use *taurus.Authority()*, *taurus.Device()*, and *taurus.Attribute()* helper functions, as in the examples below, or if you require more control, use the `taurus.core.taurusmanager.TaurusManager` or `taurus.core.taurusfactory.TaurusFactory` APIs.

Here are some examples:

The Taurus Authority associated with the Tango database running on host "machine01" and port 10000 is named "//machine:10000" (note that Taurus authority names are always prefixed by "//", to comply with RFC3986). And you can get the corresponding Taurus Authority object as:

```
>>> import taurus
>>> my_db = taurus.Authority('tango://machine:10000')
```

If "tango" is configured as the default scheme for Taurus, the 'tango:' prefix could be avoided and same database could be accessed as:

```
>>> my_db = taurus.Authority('//machine:10000')
```

Now, assume that a TangoTest device is registered in the above database as `sys/tg_test/1`. In this case, the corresponding Taurus device full name would be `tango://machine:10000/sys/tg_test/1` and it could be accessed as:

```
>>> import taurus
>>> my_device = taurus.Device('tango://machine:10000/sys/tg_test/1')
```

If "tango" is configured as the default scheme for Taurus, the previous name could be shortened to `//machine:10000/sys/tg_test/1` or even to `sys/tg_test/1` if the TANGO_HOST environment variable (or tango.rc file) point to *machine:10000* as the default tango database. Furthermore, if, on top of that, this device is aliased as `tgtest1` in the database, it could be accessed as:

```
>>> import taurus
>>> my_device = taurus.Device('tgtest1')
```

Similarly, accessing the `ampli` attribute from the above Tango device can be done using its full name:

```
>>> import taurus
>>> my_attr = taurus.Attribute('tango://machine:10000/sys/tg_test/1/ampli')
```

And of course shorter names can also be used for attributes. Following the examples for the device above, the following names could also have been passed to *taurus.Attribute()*:

- `//machine:10000/sys/tg_test/1/ampli`
- `sys/tg_test/1/ampli`
- `tgtest1/ampli`

Finally, the TangoFactory object can be accessed as:

```
>>> import taurus
>>> tg_factory = taurus.Factory('tango')
```

Note: Previous to TEP3, a RFC3986 non-compliant syntax was used for the Tango scheme (e.g., allowing names such as `tango://a/b/c/d` -note the double slash which should not be there). This syntax is now deprecated and should not be used. Taurus will issue warnings if detected.

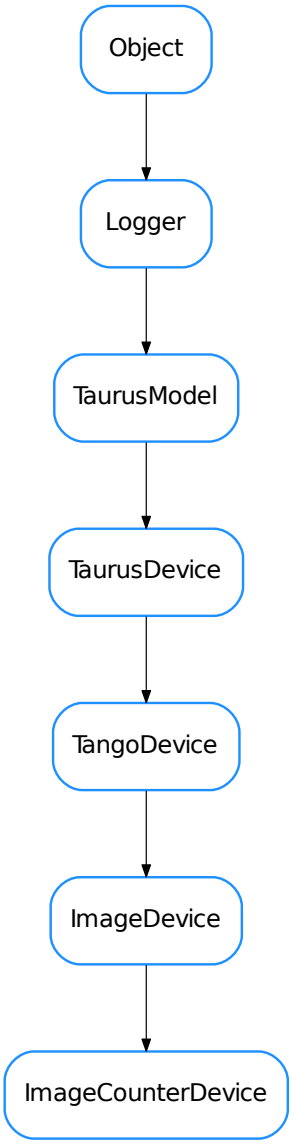
Modules

taurus.core.tango.img

The img package. It contains specific part of tango devices dedicated to images (CCDs, detectors, etc)

Classes

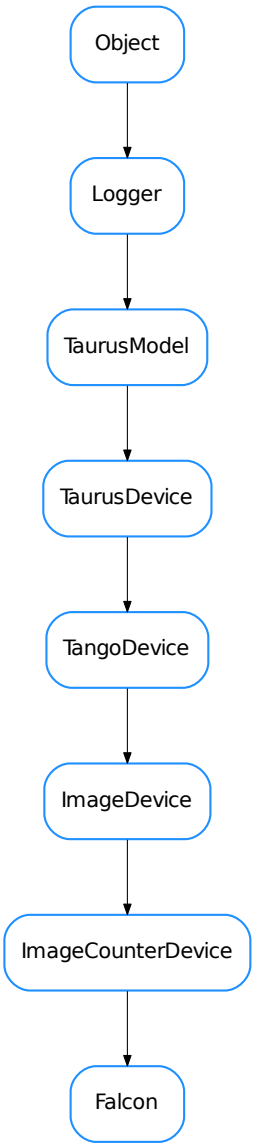
CCDPVCAM



CCDPVCAM

alias of *ImageCounterDevice*

Falcon



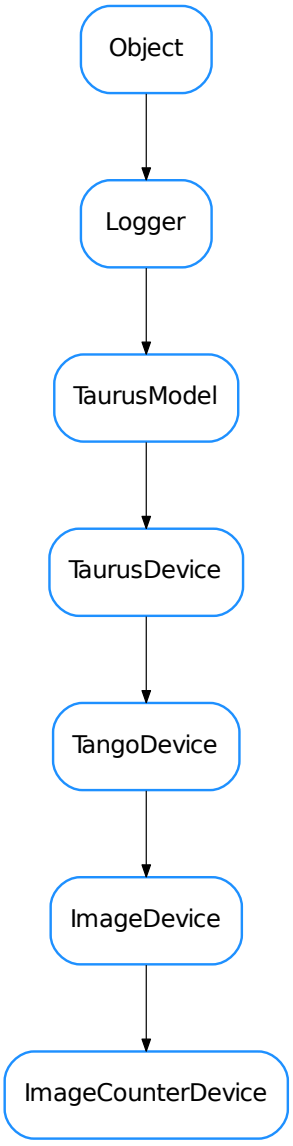
class Falcon (*name*, *image_name='image'*, ***kw*)

Bases: `taurus.core.tango.img.img.ImageCounterDevice`

eventReceived (*evt_src*, *evt_type*, *evt_value*)

getImageData (*names=None*)

ImageCounterDevice



class **ImageCounterDevice**(*name*, *image_name='image'*, *image_ct='imagecounter'*, ***kw*)

Bases: `taurus.core.tango.img.img.ImageDevice`

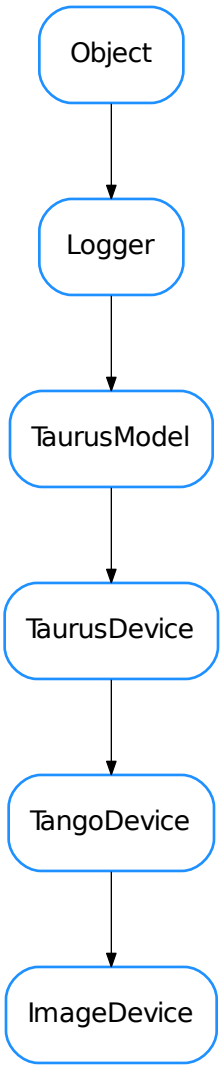
A class encapsulating a generic image device that has an image counter attribute

eventReceived(*evt_src*, *evt_type*, *evt_value*)

getImageData(*names=None*)

getImageIDAttrName()

ImageDevice



class ImageDevice (*name*, *image_name='image'*, ***kw*)

Bases: `taurus.core.tango.tangodevice.TangoDevice`

A class encapsulating a generic image device

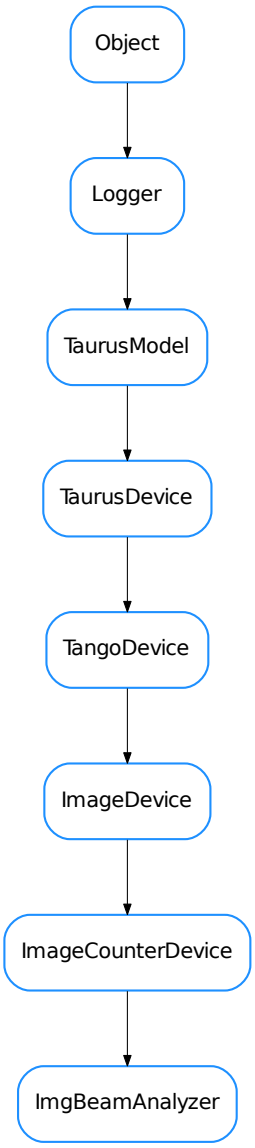
addImageAttrName (*attr_name*)

getImageAttrName (*idx=0*)

getImageAttrNames ()

setImageAttrName (*attr_name*)

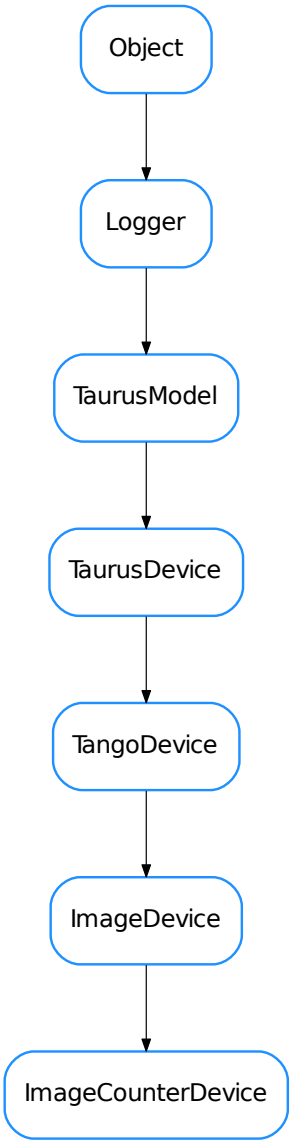
ImgBeamAnalyzer



class ImgBeamAnalyzer(*name*, *image_name='roiimage'*, ***kw*)

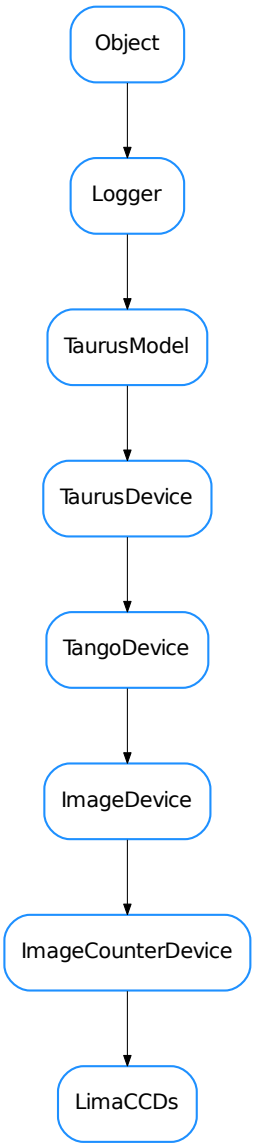
Bases: `taurus.core.tango.img.img.ImageCounterDevice`

ImgGrabber



ImgGrabber
alias of *ImageCounterDevice*

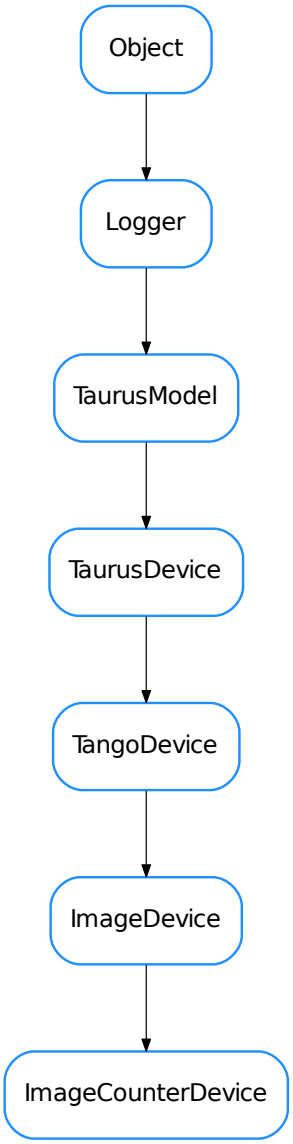
LimaCCDs



class LimaCCDs (*name*, *image_name='video_last_image'*, *image_ct='video_last_image_counter'*, ***kw*)

Bases: `taurus.core.tango.img.img.ImageCounterDevice`

PyImageViewer



PyImageViewer
alias of *ImageCounterDevice*

- *CCDPVCAM*
- *Falcon*
- *ImageCounterDevice*
- *ImageDevice*
- *ImgBeamAnalyzer*
- *ImgGrabber*

- *LimaCCDs*
- *PyImageViewer*

Functions

registerExtensions()
Registers the image extensions in the *taurus.core.tango.TangoFactory*

taurus.core.tango.util

The sardana package. It contains specific part of sardana

Functions

tangoFormatter(*dtype=None*, ***kwargs*)
The tango formatter callable. Returns a format string based on the *format* Tango Attribute configuration (Display.Format in Tango DB)

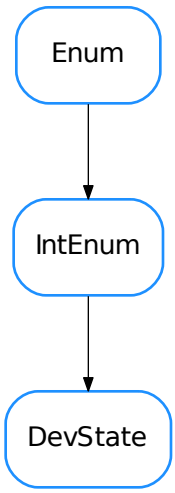
Parameters

- **dtype** (type) – type of the value object
- **kwargs** – other keyword arguments (ignored)

Returns the string formatting

Classes

DevState



class DevState

Bases: enum.IntEnum

This is the taurus.core.tango equivalent to PyTango.DevState. It defines the same members and uses the same numerical values internally, allowing equality comparisons with `PyTango.DevState` (but not identity checks!):

```
from taurus.core.tango import DevState as D1
from PyTango import DevState as D2

D1.OPEN == D2.OPEN           # --> True
D1.OPEN in (D2.ON, D2.OPEN) # --> True
D1.OPEN == 3                 # --> True
D1.OPEN is 3                 # --> False
D1.OPEN is D2.OPEN           # --> False
```

ALARM = 11

CLOSE = 2

DISABLE = 12

EXTRACT = 5

FAULT = 8

INIT = 9

INSERT = 4

MOVING = 6

OFF = 1

ON = 0

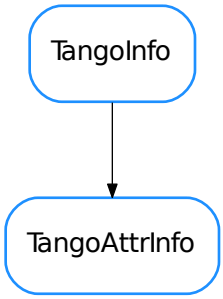
OPEN = 3

RUNNING = 10

STANDBY = 7

UNKNOWN = 13

TangoAttrInfo



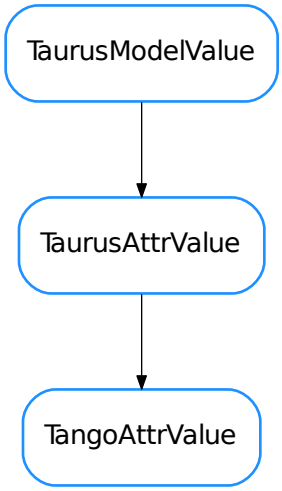
class **TangoAttrInfo**(*container*, *name=None*, *full_name=None*, *device=None*, *info=None*)

Bases: `taurus.core.tango.tangodatabase.TangoInfo`

device()

info()

TangoAttrValue



class **TangoAttrValue**(*attr=None*, *pytango_dev_attr=None*, *config=None*)

Bases: `taurus.core.taurusbasetypes.TaurusAttrValue`

A TaurusAttrValue specialization to decode PyTango.DeviceAttribute objects

has_failed

Deprecated since version 4.0: Use .error instead

value

for backwards compat with taurus < 4

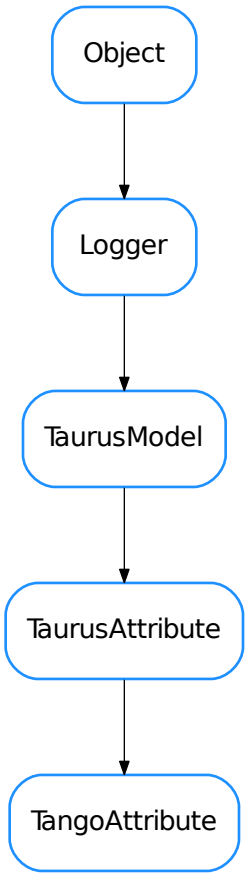
Deprecated since version 4.0: Use .rvalue instead

w_value

for backwards compat with taurus < 4

Deprecated since version 4.0: Use .wvalue instead

TangoAttribute



class **TangoAttribute**(*name*, *parent*, ***kwargs*)

Bases: `taurus.core.taurusattribute.TaurusAttribute`

addListener(*listener*)

Add a TaurusListener object in the listeners list. If it is the first element and Polling is enabled starts the polling mechanism. If the listener is already registered nothing happens.

alarms

calarms

Deprecated since version 4.0: Use getAlarms instead

cleanUp()

climits

Deprecated since version 4.0: Use getRange instead

cranges

Deprecated since version 4.0: Use getRange + getAlarms + getWarnings instead

cwarnings
Deprecated since version 4.0: Use getAlarms instead

decode(*attr_value*)
Decodes a value that was received from PyTango into the expected representation

description

dev_alias

displayValue(**args*, ***kwargs*)
Deprecated since version 4.0.

encode(*value*)
Translates the given value into a tango compatible value according to the attribute data type.

Raises *pint.DimensionalityError* if value is a Quantity and it cannot be expressed in the units of the attribute set in the DB

format
Deprecated since version 4.0.4: Use format_spec or precision instead

getAlarms(*cache=True*)

getAttributeInfoEx()

getAttributeProxy()
Convenience method that creates and returns a PyTango.AttributeProxy object

getCAlarms(**args*, ***kwargs*)
Deprecated since version 4.0: Use getAlarms instead

getCLimits(**args*, ***kwargs*)
Deprecated since version 4.0: Use getRange instead

getCRanges(**args*, ***kwargs*)
Deprecated since version 4.0: Use getRange + getAlarms + getWarnings instead

getCWarnings(**args*, ***kwargs*)
Deprecated since version 4.0: Use getWarnings instead

getConfig(**args*, ***kwargs*)
Returns the current configuration of the attribute.

Deprecated since version 4.0: Use self instead

getDescription(**args*, ***kwargs*)
Deprecated since version 4.0: Use .description instead

getDisplayUnit(**args*, ***kwargs*)
Deprecated since version 4.0: Use .rvalue.units instead

getDisplayValue(**args*, ***kwargs*)
Deprecated since version 4.0: Use getLabel instead

getDisplayWriteValue(**args*, ***kwargs*)
Deprecated since version 4.0.

getFormat(*cache=True*)

getLabel(*cache=True*)

getLimits(*cache=True*)

getMaxAlarm(**args*, ***kwargs*)
Deprecated since version 4.0: Use .alarms[1] instead

getMaxDim(*cache=True*)

getMaxDimX(**args*, ***kwargs*)
Deprecated since version 4.0: Use getMaxDim instead

getMaxDimY(**args*, ***kwargs*)
Deprecated since version 4.0: Use getMaxDim instead

getMaxValue(**args*, ***kwargs*)
Deprecated since version 4.0: Use getMaxRange instead

getMaxWarning(**args*, ***kwargs*)
Deprecated since version 4.0: Use .warnings[1] instead

getMinAlarm(**args*, ***kwargs*)
Deprecated since version 4.0: Use .alarms[0] instead

getMinValue(**args*, ***kwargs*)
Deprecated since version 4.0: Use getMinRange instead

getMinWarning(**args*, ***kwargs*)
Deprecated since version 4.0: Use .warnings[0] instead

getNewOperation(*value*)

getParam(**args*, ***kwargs*)
Get attributes of AttributeInfoEx (PyTango)

Deprecated since version 4.0: Use getAttributeInfoEx instead

getRange(*cache=True*)

getRanges(*cache=True*)

getShape(**args*, ***kwargs*)
Deprecated since version 4.0.

getStandardUnit(**args*, ***kwargs*)
Deprecated since version 4.0: Use .rvalue.units instead

getTangoWritable(*cache=True*)
like TaurusAttribute.isWritable(), but it returns a PyTango.AttrWriteType instead of a bool

getUnit(**args*, ***kwargs*)
Deprecated since version 4.0: Use .rvalue.units instead

getWarnings(*cache=True*)

getWritable(**args*, ***kwargs*)
Deprecated since version 4.0: Use isWritable instead

isBoolean(*inc_array=False*)

isFloat(*inc_array=False*)

isImage(**args*, ***kwargs*)
Deprecated since version 4.0: Use self.data_format instead

isInformDeviceOfErrors(**args*, ***kwargs*)
Deprecated since version 4.0.

isInteger(*inc_array=False*)

isNumeric(*inc_array=False*)

isReadOnly(*cache=True*)

isReadWrite(*cache=True*)

isScalar(**args*, ***kwargs*)
Deprecated since version 4.0: Use self.data_format instead

isSpectrum(**args*, ***kwargs*)
Deprecated since version 4.0: Use self.data_format instead

isState()

isUsingEvents()

isWrite(*cache=True*)

label

max_alarm
Deprecated since version 4.0: Use .alarms[1] instead

max_warning
Deprecated since version 4.0: Use .warnings[1] instead

min_alarm
Deprecated since version 4.0: Use .alarms[0] instead

min_warning
Deprecated since version 4.0: Use .warnings[0] instead

no_abs_change = 'Not specified'

no_archive_abs_change = 'Not specified'

no_archive_period = 'Not specified'

no_archive_rel_change = 'Not specified'

no_cfg_value = '——'

no_delta_t = 'Not specified'

no_delta_val = 'Not specified'

no_description = 'No description'

no_display_unit = 'No display unit'

no_max_alarm = 'Not specified'

no_max_value = 'Not specified'

no_max_warning = 'Not specified'

no_min_alarm = 'Not specified'

no_min_value = 'Not specified'

no_min_warning = 'Not specified'

no_rel_change = 'Not specified'

no_standard_unit = 'No standard unit'

no_unit = 'No unit'

not_specified = 'Not specified'

poll(***kwargs*)
Notify listeners when the attribute has been polled

push_event(*event*)
Method invoked by the PyTango layer when an event occurs. It propagates the event to listeners and delegates other tasks to specific handlers for different event types.

range

read(*cache=True*)
Returns the current value of the attribute. if cache is set to True (default) or the attribute has events active then it will return the local cached value. Otherwise it will read the attribute value from the tango device.

removeListener(*listener*)
Remove a TaurusListener from the listeners list. If polling enabled and it is the last element the stop the polling timer. If the listener is not registered nothing happens.

setAlarms(**limits*)

setConfigEx(*config*)

setDescription(**args*, ***kwargs*)
Deprecated since version 4.0: Use .description instead

setLabel(*lbl*)

setLimits(*low*, *high*)

setMaxAlarm(**args*, ***kwargs*)
Deprecated since version 4.0: Use .alarms instead

setMaxWarning(**args*, ***kwargs*)
Deprecated since version 4.0: Use .warnings instead

setMinAlarm(**args*, ***kwargs*)
Deprecated since version 4.0: Use .alarms instead

setMinWarning(**args*, ***kwargs*)
Deprecated since version 4.0: Use .warnings instead

setParam(**args*, ***kwargs*)
Set attributes of AttributeInfoEx (PyTango)

Deprecated since version 4.0: Use PyTango instead

setRange(**limits*)

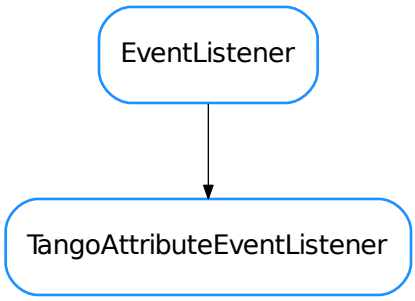
setWarnings(**limits*)

unit
Deprecated since version 4.0: Use .rvalue.units instead

warnings

write(*value*, *with_read=True*)
Write the value in the Tango Device Attribute

TangoAttributeEventListener



class **TangoAttributeEventListener**(*attr*)

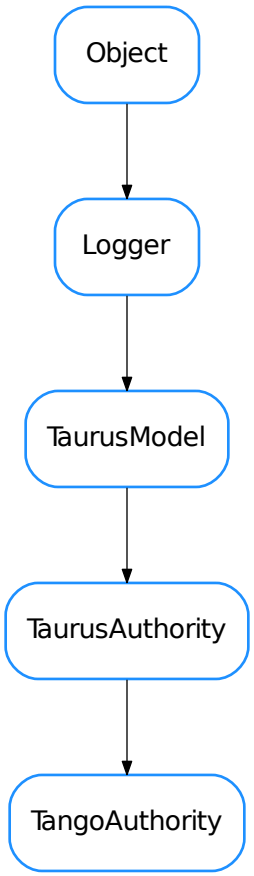
Bases: `taurus.core.util.event.EventListener`

A class that listens for an event with a specific value

Note: Since this class stores for each event value the last timestamp when it occured, it should only be used for events for which the event value domain (possible values) is limited and well known (ex: an enum)

eventReceived(*s*, *t*, *v*)

TangoAuthority



class **TangoAuthority**(*host=None*, *port=None*, *parent=None*)

Bases: `taurus.core.taurusauthority.TaurusAuthority`

addListener(*listener*)

cache()

deviceTree()

Returns a tree container with all devices in three levels : domain, family and member

Return type `TangoDevTree`

Returns a tree containning all devices

getAliasNames()

Returns a list of registered tango device alias

Return type `sequence<str>`

Returns a sequence with all registered tango device alias

getClassNames()
Returns a list of registered tango device classes

Return type sequence<str>

Returns a sequence with all registered tango device classes

getDescription(**args*, ***kwargs*)
Deprecated since version 4.0: Use .description instead

getDevice(*name*)
Reimplemented from TaurusDevice to use cache and return *taurus.core.tango.TangoDevInfo* objects with information about the given device name

Parameters **name** (str) – the device name

Return type *TangoDevInfo*

Returns information about the tango device

getDeviceDomainNames()

getDeviceFamilyNames(*domain*)

getDeviceMemberNames(*domain*, *family*)

getDeviceNames()
Returns a list of registered tango device names

Return type sequence<str>

Returns a sequence with all registered tango device names

getDisplayValue(**args*, ***kwargs*)
Deprecated since version 4.0: Use getFullName instead

getDomainDevices(*domain*)

getElementAlias(*full_name*)
return the alias of an element from its full name

getElementFullName(*alias*)
return the full name of an element from its alias

getFamilyDevices(*domain*, *family*)

getServerNameInstances(*serverName*)

getServerNames()
Returns a list of registered tango device servers in format<name>/<instance>

Return type sequence<str>

Returns a sequence with all registered tango device servers

getTangoDB()

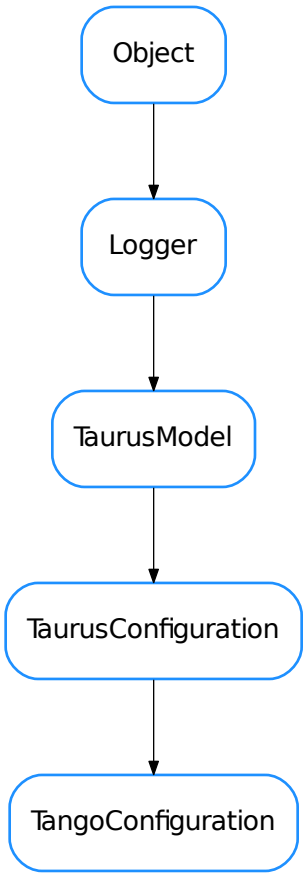
getValueObj(**args*, ***kwargs*)
Deprecated since version 4.0: Use getTangoDB instead

static get_default_tango_host()

get_device_attribute_list(*dev_name*, *wildcard*)

refreshCache()

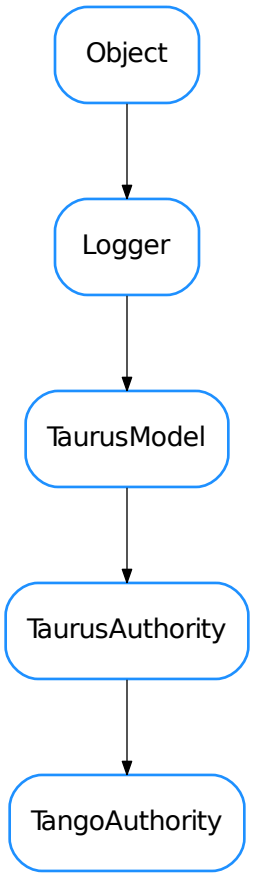
TangoConfiguration



class TangoConfiguration (**args*, ***kwargs*)

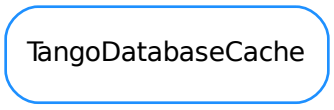
Bases: taurus.core.taurusconfiguration.TaurusConfiguration

TangoDatabase



TangoDatabase
alias of *TangoAuthority*

TangoDatabaseCache



class TangoDatabaseCache(*db*)

Bases: object

db

deviceTree()

Returns a tree container with all devices in three levels: domain, family and member

Return type TangoDevTree

Returns a tree containning all devices

devices()

getAliasNames()

getClassNames()

Returns a list of registered device classes

Return type sequence <str>

Returns a sequence with all registered device classes

getDevice(*name*)

Returns a *TangoDevInfo* object with information about the given device name

Parameters **name** (str) – the device name

Return type *TangoDevInfo*

Returns information about the device

getDeviceDomainNames()

getDeviceFamilyNames(*domain*)

getDeviceMemberNames(*domain*, *family*)

getDeviceNames()

Returns a list of registered device names

Return type sequence <str>

Returns a sequence with all registered device names

getDomainDevices(*domain*)

getFamilyDevices(*domain*, *family*)

getServerNameInstances(*serverName*)

getServerNames()

Returns a list of registered server names

Return type sequence <str>

Returns a sequence with all registered server names

klasses()

refresh()

refreshAttributes(*device*)

serverTree()

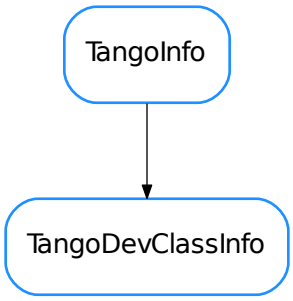
Returns a tree container with all servers in two levels: server name and server instance

rtype TangoServerTree

return a tree containning all servers

servers()

TangoDevClassInfo



class TangoDevClassInfo(*container*, *name=None*, *full_name=None*)

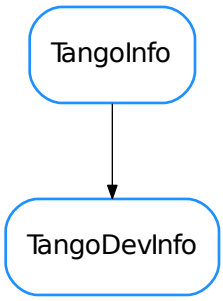
Bases: taurus.core.tango.tangodatabase.TangoInfo

addDevice(*dev*)

devices()

getDeviceNames()

TangoDevInfo



class TangoDevInfo(*container*, *name=None*, *full_name=None*, *alias=None*, *server=None*, *klass=None*, *exported=False*, *host=None*)

Bases: taurus.core.tango.tangodatabase.TangoInfo

alias()

alive()

attributes()

domain()

exported()

family()

getAttribute(*attrname*)

getDeviceProxy()

getHWObj(**args*, ***kwargs*)
Deprecated since version 4.0: Use getDeviceProxy() instead

host()

klass()

member()

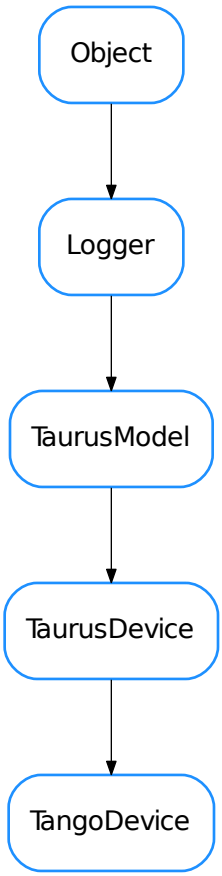
refreshAttributes()

server()

setAttributes(*attributes*)

state()
Overwrite state so it doesn't call 'alive()' since it can take a long time for devices that are declared as exported but are in fact not running (crashed, network error, power cut, etc)

TangoDevice



class TangoDevice (*name*, ***kw*)

Bases: `taurus.core.taurusdevice.TaurusDevice`

A Device object representing an abstraction of the PyTango.DeviceProxy object in the taurus.core.tango scheme

addListener (*listener*)

cleanUp ()

description

eventReceived (*event_src*, *event_type*, *event_value*)

getAttribute (*attrname*)

Returns the attribute object given its name

getDescription (**args*, ***kwargs*)

Deprecated since version 4.0: Use .description instead

getDeviceProxy ()

getDisplayDescrObj(*cache=True*)

getDisplayValue(**args*, ***kwargs*)
Deprecated since version 4.0: Use .state().name instead

getHWObj(**args*, ***kwargs*)
Deprecated since version 4.0: Use getDeviceProxy() instead

getLockInfo(*cache=False*)

getSWState(**args*, ***kwargs*)
Deprecated since version 4.0: Use state instead

getState(**args*, ***kwargs*)
Deprecated since version 4.0: Use .stateObj.read().rvalue [Tango] or .state [agnostic] instead

getStateObj(**args*, ***kwargs*)
Deprecated since version 4.0: Use .stateObj [Tango] or .factory.getAttribute(state_full_name) [agnostic] instead

getValueObj(**args*, ***kwargs*)

Deprecated by TEP14.

..warning:: this bck-compat implementation is not perfect because the rvalue of the returned TangoAttributeValue is now a member of TaurusDevState instead of TaurusSWDevState

Deprecated since version 4.0: Use state [agnostic] or stateObj.read [Tango] instead

isValidDev(**args*, ***kwargs*)
see: `TaurusDevice.isValid()`

Deprecated since version 4.0: Use getDeviceProxy() is not None instead

lock(*force=False*)

poll(*attrs*, *asynch=False*, *req_id=None*)
optimized by reading of multiple attributes in one go

removeListener(*listener*)

state
Reimplemented from `TaurusDevice` to use Tango's state attribute for diagnosis of the current state. It supports a "cache" kwarg

Parameters **cache** (`bool`) – If True (default), cache will be used when reading the state attribute of this device

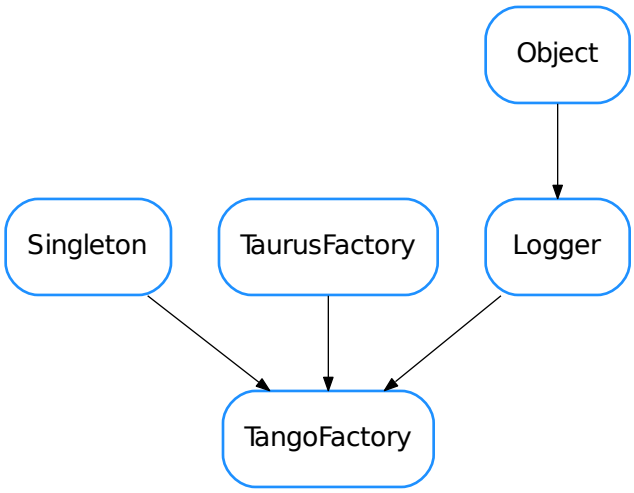
Return type `TaurusDevState`

Returns

stateObj

unlock(*force=False*)

TangoFactory



class TangoFactory

Bases: taurus.core.util.singleton.Singleton, taurus.core.taurusfactory.TaurusFactory, taurus.core.util.log.Logger

A TaurusFactory singleton class to provide Tango-specific Taurus Element objects (TangoAuthority, TangoDevice, TangoAttribute)

Tango model names are URI based See https://tools.ietf.org/html/rfc3986. For example, a TangoAttribute would be:

```
tango://foo.org:1234/a/b/c/d#label
\___/   \_____/ \__/ \_____/ \___/
  |        |     |      |      |
  |    hostname port   attr     |
  |   \_____________/\______/   |
  |         |           |      |
scheme   authority     path  fragment
```

For Tango Elements:

- The 'scheme' must be the string "tango" (lowercase mandatory)
- The 'authority' identifies the Tango database (<hostname> and <port> are mandatory if authority is given)
- The 'path' identifies Tango Device and Attributes. For devices it must have the format _/_/_ or alias For attributes it must have the format _/_/_/_ or devalias/_
- The 'fragment' is optional and it refers to a member of the model object, thus not being part of the model name itself

caseSensitive = False

cleanUp()

Cleanup the singleton instance

disablePolling()
Disable the application tango polling

elementTypesMap = {18: <class 'taurus.core.tango.tangodatabase.TangoAuthority'>, 3: <class 'taurus.core.tango.tango

enablePolling()
Enable the application tango polling

getAttribute(*attr_name*, *create_if_needed=True*, ***kwargs*)
Obtain the object corresponding to the given attribute name. If the corresponding attribute already exists, the existing instance is returned. Otherwise a new instance is stored and returned.

Parameters

- **attr_name** (str) – a valid attribute name URI
- **create_if_needed** (bool) – If True, the Attribute is created if it did not already exist. If False, None is returned if it did not exist

Return type TangoAttribute

Returns attribute object

Raise (TaurusException) if the given alias is invalid.

getAttributeInfo(*full_attr_name*)
Deprecated: Use *taurus.core.tango.TangoFactory.getConfiguration()* instead.

Obtain attribute information corresponding to the given attribute name. If the corresponding attribute info already exists, the existing information is returned. Otherwise a new information instance is stored and returned.

Parameters **full_attr_name** (str) – attribute name in format: <tango device name>'/'<attribute name>

Return type *TangoConfiguration*

Returns configuration object

getAttributeNameValidator()
Return TangoAttributeNameValidator

getAuthority(*name=None*)
Obtain the object corresponding to the given database name or the default database if name is None. If the corresponding authority object already exists, the existing instance is returned. Otherwise a new instance is stored and returned.

Parameters **name** (str) – database name string alias. If None, the default database is used

Return type TangoAuthority

Returns database object

Raise (TaurusException) if the given alias is invalid.

getAuthorityNameValidator()
Return TangoAuthorityNameValidator

getConfiguration(**args*, ***kwargs*)

Obtain the object corresponding to the given attribute or full name. If the corresponding configuration already exists, the existing instance is returned. Otherwise a new instance is stored and returned.

type param TaurusAttribute or str

param param attribute object or full configuration name

rtype *TangoAttribute*

return configuration object

Deprecated since version 4.0: Use getAttribute instead

getDatabase(*name=None*)
Deprecated. Use getAuthority instead

getDatabaseNameValidator()
Deprecated

getDevice(*dev_name*, *create_if_needed=True*, ***kw*)
Obtain the object corresponding to the given tango device name. If the corresponding device already exists, the existing instance is returned. Otherwise a new instance is stored and returned.

Parameters

- **dev_name** (str) – tango device name or tango alias for the device. It must be a valid Tango device URI. If authority is not explicit, the default Tango Database will be used
- **create_if_needed** (bool) – If True, the Device is created if it did not exist previously. If False, it returns None if it did not exist

Return type *TangoDevice*

Returns a device object

Raise (TaurusException) if the given dev_name is invalid.

getDeviceNameValidator()
Return TangoDeviceNameValidator

getExistingAttribute(*attr_name*)
Deprecated: use getAtribute with create_if_needed=False

getExistingAttributes()
Returns a new dictionary will all registered attributes on this factory

Returns dictionary will all registered attributes on this factory

Return type dict

getExistingDatabases()
Returns a new dictionary will all registered databases on this factory

Returns dictionary will all registered databases on this factory

Return type dict

getExistingDevice(*dev_name*)
Deprecated: use getDevice with create_if_needed=False

getExistingDevices()
Returns a new dictionary will all registered devices on this factory

Returns dictionary will all registered devices on this factory

Return type dict

getOperationMode()
Deprecated. Gives the current operation mode.

get_default_tango_host()
Retruns the current default tango host

init(**args*, ***kwargs*)
Singleton instance initialization. **For internal usage only**

isPollingEnabled()
Tells if the local tango polling is enabled

Return type bool

Returns wheter or not the polling is enabled

reInit()
Reinitialize the singleton

registerAttributeClass(*attr_name*, *attr_klass*)
Registers a new attribute class for the attribute name.

Parameters

- **attr_name** (str) – attribute name
- **attr_klass** (*TangoAttribute*) – the new class that will handle the attribute

registerDeviceClass(*dev_klass_name*, *dev_klass*)
Registers a new python class to handle tango devices of the given tango class name

Parameters

- **dev_klass_name** (str) – tango device class name
- **dev_klass** (*TangoDevice*) – the new class that will handle devices of the given tango class name

removeExistingAttribute(*attr_or_attr_name*)
Removes a previously registered attribute.

Parameters **attr_or_attr_name** (str or *TangoAttribute*) – attribute name or attribute object

removeExistingDevice(*dev_or_dev_name*)
Removes a previously registered device.

Parameters **dev_or_dev_name** (str or *TangoDevice*) – device name or device object

schemes = ('tango',)
the list of schemes that this factory supports. For this factory: 'tango' is the only scheme

setOperationMode(*mode*)
Deprecated. setOperationMode(OperationMode mode) -> None Sets the operation mode for the Tango system.

set_default_tango_host(*tango_host*)
Sets the new default tango host. The method will transform the given name to an Authority URI.

Note: Calling this method also clears the device alias cache.

Parameters **tango_host** (str) – the new tango host. It accepts any valid Tango

authority name or None to use the defined by $TANGO_HOST env. var.

unregisterAttributeClass(*attr_name*)
Unregisters the attribute class for the given attribute If no class was registered before for the given attribute, this call as no effect

Parameters **attr_name** (str) – attribute name

unregisterDeviceClass (*dev_klass_name*)
Unregisters the class for the given tango class name If no class was registered before for the given attribute, this call as no effect

Parameters **dev_klass_name** (str) – tango device class name

TangoInfo

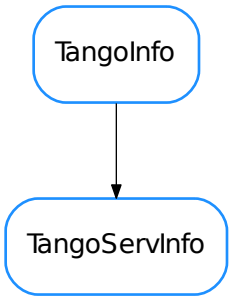
class TangoInfo (*container*, *name=None*, *full_name=None*)
Bases: object

container()

fullName()

name()

TangoServInfo



class TangoServInfo (*container*, *name=None*, *full_name=None*)
Bases: taurus.core.tango.tangodatabase.TangoInfo

addDevice (*dev*)

alive()

devices()

exported()

getClassNames()

getDeviceNames()

host()

serverInstance()

serverName()

state()

- *DevState*
- *TangoAttrInfo*
- *TangoAttrValue*
- *TangoAttribute*
- *TangoAttributeEventListener*
- *TangoAuthority*
- *TangoConfiguration*
- *TangoDatabase*
- *TangoDatabaseCache*
- *TangoDevClassInfo*
- *TangoDevInfo*
- *TangoDevice*
- *TangoFactory*
- *TangoInfo*
- *TangoServInfo*

taurus.core.util

This package consists of a collection of useful classes and functions. Most of the elements are taurus independent and can be used generically.

This module contains a python implementation of `json`. This was done because json only became part of python since version 2.6. The json implementation follows the rule:

1. if python >= 2.6 use standard json from python distribution
2. otherwise use private implementation distributed with taurus

Modules

taurus.core.util.argparse

Helper command line parser for taurus based on `optparse`. Suppose you have an application file:

```
import sys
from PyQt4 import Qt


class GUI(Qt.QMainWindow):
    pass


def main():
```

```
    import taurus.core.util.argparse as argparse

    parser, options, args = argparse.init_taurus_args()

    app = Qt.QApplication(sys.argv)
    w = GUI()
    w.show()
    sys.exit(app.exec_())

if __name__ == "__main__":
    main()
```

The call to *taurus.core.util.argparse.init_taurus_args()* will initialize taurus environment based on the command line options given by the user. Currently, the known options are:

1. `--help` prints the total number of available options
2. `--taurus-log-level` sets the taurus log level
3. `--tango-host` sets the default tango host
4. `--taurus-polling-period` sets the default taurus global polling period (milliseconds)
5. `--taurus-serialization-mode` sets the default taurus serialization mode
6. `--remote-console-port` enables remote debugging using the given port

You can easily extend the taurus options with your application specific options. Suppose you want to add an option like `--model=<model name>`:

```
def main():
    import taurus.core.util.argparse as argparse
    parser = argparse.get_taurus_parser(parser=parser)
    parser.set_usage("%prog [options] <special item>")
    parser.set_description("my own GUI application")
    parser.add_option("--model")

    parser, options, args = argparse.init_taurus_args(parser=parser)

    app = Qt.QApplication(sys.argv)
    w = GUI()
    w.show()
    sys.exit(app.exec_())
```

Functions

get_taurus_parser(*parser=None*)

Returns a optparse.OptionParser initialized with a optparse.OptionGroup containning some taurus options. If a parser is given as parameter then it uses this parser instead of creating a new one.

Parameters **parser** (optparse.OptionParser) – an option parser or None (default) to create a new parser

Returns an option parser or the given parser if it is not None

Return type optparse.OptionParser

init_taurus_args(*parser=None*, *args=None*, *values=None*)

Parses the command line using *taurus.core.util.argparse.parse_taurus_args()*.

After the command line is parsed, actions are taken on each recognized parameter. For example, the taurus log level and the default tango host are set accordingly.

Parameters

- **parser** (optparse.OptionParser) – an option parser or None (default) to create a new parser
- **args** (*seq<str>*) – the list of arguments to process (default is None meaning: sys.argv[1:])
- **values** – a optparse.Values object to store option arguments in (default is None meaning: a new instance of Values) - if you give an existing object, the option defaults will not be initialized on it

Returns a tuple of three elements: parser, options, args

Return type optparse.OptionParser, optparse.Values, seq<str>

parse_taurus_args(*parser=None*, *args=None*, *values=None*)

Parses the command line. If parser is not given, then a new parser is created. In any case, the parser is initialized using the *taurus.core.util.argparse.get_taurus_parser()*. args and values are the optional parameters that will be given when executing optparse.OptionParser.parse_args().

Parameters

- **parser** (optparse.OptionParser) – an option parser or None (default) to create a new parser
- **args** (*seq<str>*) – the list of arguments to process (default is None meaning: sys.argv[1:])
- **values** – a optparse.Values object to store option arguments in (default is None meaning: a new instance of Values) - if you give an existing object, the option defaults will not be initialized on it

Returns a tuple of three elements: parser, options, args

Return type optparse.OptionParser, optparse.Values, seq<str>

split_taurus_args(*parser*, *args=None*)

Splits arguments into valid parser arguments and non valid parser arguments.

Parameters

- **parser** (optparse.OptionParser) – an option parser
- **args** (*seq<str>*) – the list of arguments to process (default is None meaning: sys.argv)

Returns a tuple of two elements: parser args, non parser args

Return type seq<seq<str>, seq<str>>

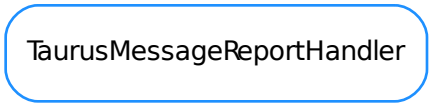
taurus.core.util.decorator

taurus.core.util.report

This module provides a panel to display taurus messages

Classes

TaurusMessageReportHandler



class TaurusMessageReportHandler (*parent*)

Bases: object

Label = 'Default report handler'

report (*message*)

- *TaurusMessageReportHandler*

Classes

ArrayBuffer

class ArrayBuffer (*buffer*, *maxSize=0*)

Bases: object

A data buffer which internally uses a preallocated numpy.array. An ArrayBuffer will only present the actual contents, not the full internal buffer, so when appending or extending, it behaves as if dynamic reallocation was taking place.

The contents of the class:*ArrayBuffer* can be accessed as if it was a numpy array (i.e slicing notation like b[2:3], b[:,2], b[-1],... are all valid).

For retrieving the full contents, see *ArrayBuffer.contents()* and *ArrayBuffer.toArray()*

On creation, a given initial internal buffer and a maximum size are set. If the maximum size is larger than the original internal buffer, this will be automatically grown in geometrical steps if needed to accommodate the contents, up to the maximum size. Once the contents fill the maximum size, appending or extending the contents will result in older contents being discarded (in a FIFO way)

The *append()* and meth:*extend* methods are designed to be cheap (especially if the internal buffer size is already at the maximum size), at the expense of memory usage

append(*x*)
similar to the append method in a list, except that once the maximum buffer size is reached, elements get discarded on the begginning to keep the size within the limit

Parameters **x** (scalar) – element to be appended

See also:

extend()

bufferSize()
Returns the current size of the internal buffer

Return type int

Returns lcurrent length of the internal buffer

See also:

contentsSize(), *maxSize()*

contents()
returns the array of the contents that have already been filled. Note that it does not return the full buffer, only those elements that have been already set.

It is equivalent to b[:]

Return type array

Returns array of contents

See also:

toArray()

contentsSize()
Equivalent to len(b)

Return type int

Returns length of the current contents of the ArrayBuffer (not the maximum size of the buffer)

See also:

maxSize()

extend(*a*)
similar to the extend method of a list, except that once the maximum buffer size is reached, elements get discarded on the begginning to keep the size within the limit

Parameters **a** (array) – array of elements to append

See also:

append(), *extendLeft()*

extendLeft(*a*)
Prepends data to the current contents. Note that, contrary to the extent method, no data will be discarded if the maximum size limit is reached. Instead, an exception will be raised.

Parameters **a** (array) – array of elements to append

See also:

extend()

isFull()
Whether the contents fill the whole of the internal buffer

Return type bool

Returns True if the contents fill the maximum size. False otherwise.

See also:

maxSize()

maxSize()
Returns the maximum size of the internal buffer, beyond which the ArrayBuffer starts discarding elements when appending

Return type int

Returns maximum length of the internal buffer

See also:

contentsSize(), *append()*, *extend()*, *isFull()*

moveLeft(*n*)
discards n elements from the begginning to make space at the end of the buffer. Moves all elements n positions to the left and the contents size gets decreased by n

Note: if n is larger or equal than the maximum buffer size, the whole buffer is wiped

Parameters **n** (int) –

remainingSize()
returns the remaining free space in the internal buffer (e.g., 0 if it is full)

Return type int

Returns length of the unused space in the internal buffer

See also:

contentsSize(), *maxSize()*,

resizeBuffer(*newlen*)
resizes the internal buffer

setMaxSize(*maxSize*)
Sets the maximum size of the internal buffer, beyond which the ArrayBuffer starts discarding elements when appending

Parameters **maxSize** (int) – maximum length of the internal buffer

See also:

contentsSize(), *append()*, *extend()*, *isFull()*

toArray()
returns a copy of the array of the contents. It is equivalent to `b.contents.copy()`

Return type array

Returns copy of array of contents

See also:

contents()

AttributeEventIterator

class AttributeEventIterator (**attrs*)

Bases: object

Internal class. For test purposes

connect (*attrs*)

disconnect ()

eventReceived (*s*, *t*, *v*)

events (*timeout=1*)

fireEvent (*s*, *v*)

lock ()

unlock ()

AttributeEventWait

class AttributeEventWait (*attr=None*)

Bases: object

Class designed to connect to a `taurus.core.taurusattribute.TaurusAttribute` and fire events or wait for a certain event.

clearEventSet ()

Clears the internal event buffer

connect (*attr*)

Connect to the given attribute :param attr: the attribute to connect to :type attr: taurus.core.taurusattribute.TaurusAttribute

disconnect()
Disconnects from the attribute. If not connected nothing happens.

eventReceived(*s*, *t*, *v*)
Event listener method for the underlying attribute. Do not call this method. It will be called internally when the attribute generates an event.

fireEvent(*v*)
Notifies that a given event has arrived This function is protected inside with the object's lock. Do NOT call this function when you have the lock acquired on this object.

Parameters **v** (*object*) – event value

getLastRecordedEvent()
returns the value of the last recorded event or None if no event has been received or the last event was an error event

Returns the last event value to be recorded

Return type object

getRecordedEvent(*v*)
Returns the the recorded local timestamp for the event with the given value or None if no event with the given value has been recorded.

Parameters **v** (*object*) – event value

Returns local timestamp for the event or None if no event has been recorded

Return type float

getRecordedEvents()
Returns a reference to the internal dictionary used to store the internal events. Modify the return dictionary at your own risk!

Returns reference to the internal event dictionary

Return type dict

lock()
Locks this event listener

unlock()
Unocks this event listener

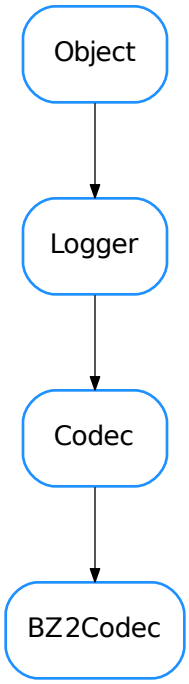
waitEvent(*val*, *after=0*, *equal=True*, *timeout=None*, *retries=-1*, *any=False*)
Wait for an event with the given value.

Parameters

- **val** (*object*) – value to compare
- **after** (*float*) – timestamp. wait for events comming after the given time. default value is 0 meaning any event after Jan 1, 1970
- **equal** (*bool*) – compare for equality. equal=True means an event with the given value, equal=False means any event which as a different value
- **timeout** (*float*) – maximum time to wait (seconds). Default is None meaning wait forever.
- **retries** – number of maximum retries of max timeout to attempts. Default is -1 meaning infinite number of retries. 0 means no wait. Positive number is obvious.

- **any** (*bool*) – if any is True ignore 'val' parameter and accept any event. If False (default),check with given 'val' parameter

BZ2Codec



class BZ2Codec

Bases: `taurus.core.util.codecs.Codec`

A codec able to encode/decode to/from BZ2 format. It uses the `bz2` module

Example:

```
>>> from taurus.core.util.codecs import CodecFactory

>>> # first encode something
>>> data = 100 * "Hello world\n"
>>> cf = CodecFactory()
>>> codec = cf.getCodec('bz2')
>>> format, encoded_data = codec.encode(("", data))
>>> print len(data), len(encoded_data)
1200, 68
>>> format, decoded_data = codec.decode((format, encoded_data))
>>> print decoded_data[20]
'Hello world\nHello wo'
```

decode (*data*, **args*, ***kwargs*)

decodes the given data from a bz2 string.

Parameters **data** (`sequence[str, obj]`) – a sequence of two elements where the first item is the encoding format of the second item object

Return type `sequence[str, obj]`

Returns a sequence of two elements where the first item is the encoding format of the second item object

encode (*data*, **args*, ***kwargs*)

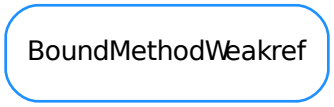
encodes the given data to a bz2 string. The given data **must** be a string

Parameters **data** (`sequence[str, obj]`) – a sequence of two elements where the first item is the encoding format of the second item object

Return type `sequence[str, obj]`

Returns a sequence of two elements where the first item is the encoding format of the second item object

BoundMethodWeakref

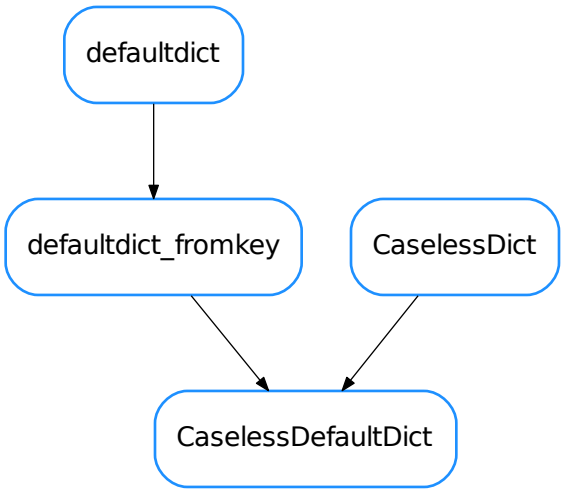


class **BoundMethodWeakref** (*bound_method*, *del_cb=None*)

Bases: `object`

This class represents a weak reference to a method of an object since weak references to methods don't work by themselves

CaselessDefaultDict



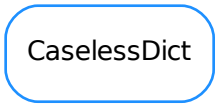
class CaselessDefaultDict

Bases: taurus.core.util.containers.defaultdict_fromkey, taurus.core.util.containers.CaselessDict

a join venture between caseless and default dict This class merges the two previous ones. This declaration equals to:

```
CaselessDefaultDict = type('CaselessDefaultType',(CaselessDict,defaultdict_
↪fromkey),{})
```

CaselessDict



class CaselessDict (*other=None*)

Bases: dict

A case insensitive dictionary. Use this class as a normal dictionary. The keys must be strings

fromkeys (*iterable*, *value=None*)

get (*key*, *def_val=None*)
overwritten from `dict.get()`

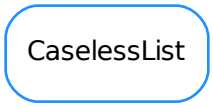
has_key (*key*)
overwritten from `dict.has_key()`

pop (*key*, *def_val=None*)
overwritten from `dict.pop()`

setdefault (*key*, *def_val=None*)
overwritten from `dict.setdefault()`

update (*other*)
overwritten from `dict.update()`

CaselessList



class CaselessList (*inlist=[]*)
Bases: *list*

A case insensitive lists that has some caseless methods. Only allows strings as list members. Most methods that would normally return a list, return a CaselessList. (Except list() and lowercopy()) Sequence Methods implemented are : __contains__, remove, count, index, append, extend, insert, __getitem__, __setitem__, __getslice__, __setslice__ __add__, __radd__, __iadd__, __mul__, __rmul__ Plus Extra methods: findentry, copy , lowercopy, list Inherited methods : __imul__, __len__, __iter__, pop, reverse, sort

append (*item*)
Adds an item to the list and checks it's a string.

copy ()
Return a CaselessList copy of self.

count (*item*)
Counts references to 'item' in a caseless manner. If item is not a string it will always return 0.

extend (*item*)
Extend the list with another list. Each member of the list must be a string.

findentry (*item*)
A caseless way of checking if an item is in the list or not. It returns None or the entry.

index (*item*, *minindex=0*, *maxindex=None*)
Provide an index of first occurence of item in the list. (or raise a ValueError if item not present) If item is not a string, will raise a TypeError. minindex and maxindex are also optional arguments s.index(x[, i[, j]]) return smallest k such that s[k] == x and i <= k < j

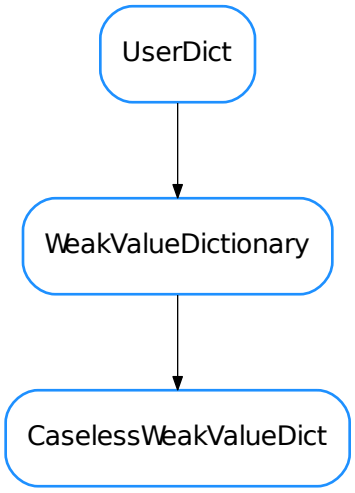
insert (*i*, *x*)
s.insert(i, x) same as s[i:i] = [x] Raises TypeError if x isn't a string.

list()
Return a normal list version of self.

lowercopy()
Return a lowercase (list) copy of self.

remove(*item*)
Remove the first occurence of an item, the caseless way.

CaselessWeakValueDict



class CaselessWeakValueDict(*other=None*)
Bases: weakref.WeakValueDictionary

fromkeys(*iterable*, *value=None*)

get(*key*, *def_val=None*)
overwritten from weakref.WeakValueDictionary.get()

has_key(*key*)
overwritten from weakref.WeakValueDictionary.has_key()

pop(*key*, *def_val=None*)
overwritten from weakref.WeakValueDictionary.pop()

setdefault(*key*, *def_val=None*)
overwritten from weakref.WeakValueDictionary.setdefault()

update(*other*)
overwritten from weakref.WeakValueDictionary.update()

CircBuf

class CircBuf (*leng*)

Bases: object

A circular buffer of Python values.

Examples:

```
>>> cb = CircBuf(3)
>>> cb.is_empty()
1
>>> cb.put('first')
>>> cb.is_empty()
0
>>> cb.put('second')
>>> cb.put('third')
>>> cb.is_full()
1
>>> cb.put('fourth')
>>> cb.get()
'second'
>>> cb.get()
'third'
>>> cb.get()
'fourth'
>>> cb.is_empty()
1
```

get ()

Retrieves an item from a non-empty circular buffer.

is_empty ()

Returns true only if CircBuf has no items.

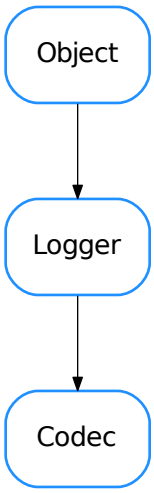
is_full ()

Returns true only if CircBuf has no space.

put (*item*)

Puts an item onto a circular buffer.

Codec



class Codec

Bases: `taurus.core.util.log.Logger`

The base class for all codecs

decode(*data*, **args*, ***kwargs*)

decodes the given data. This method is abstract an therefore must be implemented in the subclass.

Parameters **data** (`sequence[str, obj]`) – a sequence of two elements where the first item is the encoding format of the second item object

Return type `sequence[str, obj]`

Returns a sequence of two elements where the first item is the encoding format of the second item object

Raise NotImplementedError

encode(*data*, **args*, ***kwargs*)

encodes the given data. This method is abstract an therefore must be implemented in the subclass.

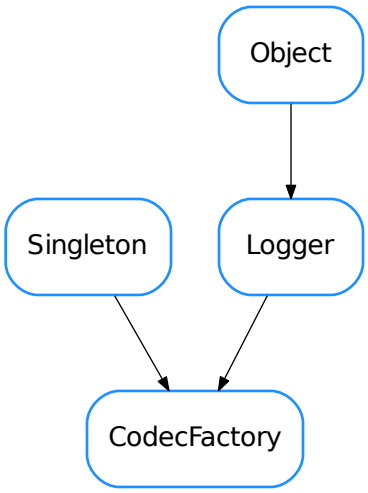
Parameters **data** (`sequence[str, obj]`) – a sequence of two elements where the first item is the encoding format of the second item object

Return type `sequence[str, obj]`

Returns a sequence of two elements where the first item is the encoding format of the second item object

Raise NotImplementedError

CodecFactory



class CodecFactory

Bases: `taurus.core.util.singleton.Singleton`, `taurus.core.util.log.Logger`

The singleton CodecFactory class.

To get the singleton object do:

```
from taurus.core.util.codecs import CodecFactory
f = CodecFactory()
```

The *CodecFactory* class allows you to get a codec object for a given format and also to register new codecs. The *CodecPipeline* is a special codec that is able to code/decode a sequence of codecs. This way you can have codecs 'inside' codecs.

Example:

```
>>> from taurus.core.util.codecs import CodecFactory
>>> cf = CodecFactory()
>>> json_codec = cf.getCodec('json')
>>> bz2_json_codec = cf.getCodec('bz2_json')
>>> data = range(100000)
>>> f1, enc_d1 = json_codec.encode(('', data))
>>> f2, enc_d2 = bz2_json_codec.encode(('', data))
>>> print len(enc_d1), len(enc_d2)
688890 123511
>>>
>>> f1, dec_d1 = json_codec.decode((f1, enc_d1))
>>> f2, dec_d2 = bz2_json_codec.decode((f2, enc_d2))
```

A Taurus related example:

```
>>> # this example shows how to automatically get the data from a DEV_ENCODED␣
↪attribute
>>> import taurus
>>> from taurus.core.util.codecs import CodecFactory
>>> cf = CodecFactory()
>>> devenc_attr = taurus.Attribute('a/b/c/devenc_attr')
>>> v = devenc_attr.read()
>>> codec = CodecFactory().getCodec(v.format)
>>> f, d = codec.decode((v.format, v.value))
```

CODEC_MAP = {'video_image': <class 'taurus.core.util.codecs.VideoImageCodec'>, '': <class 'taurus.core.util.codecs.Null
Default minimum map of registered codecs

decode (*data*, **args*, ***kwargs*)

encode (*format*, *data*, **args*, ***kwargs*)

getCodec (*format*)
Returns the codec object for the given format or None if no suitable codec is found

Parameters **format** (str) – the codec id

Return type *Codec* or None

Returns the codec object for the given format

init (**args*, ***kwargs*)
Singleton instance initialization.

registerCodec (*format*, *klass*)
Registers a new codec. If a codec already exists for the given format it is removed.

Parameters

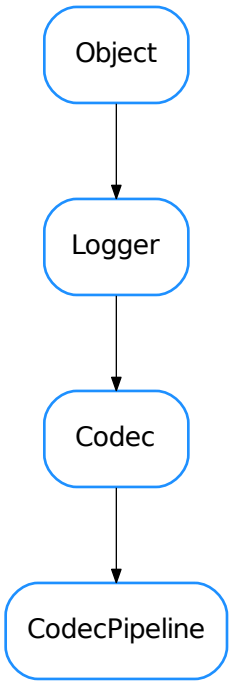
- **format** (str) – the codec id
- **klass** (Codec class) – the class that handles the format

unregisterCodec (*format*)
Unregisters the given format. If the format does not exist an exception is thrown.

Parameters **format** (str) – the codec id

Raise KeyError

CodecPipeline



class CodecPipeline (*format*)

Bases: `taurus.core.util.codecs.Codec`, `list`

The codec class used when encoding/decoding data with multiple encoders

Example usage:

```
>>> from taurus.core.util.codecs import CodecPipeline

>>> data = range(100000)
>>> codec = CodecPipeline('bz2_json')
>>> format, encoded_data = codec.encode(("", data))

# decode it
format, decoded_data = codec.decode((format, encoded_data))
print decoded_data
```

decode (*data*, **args*, ***kwargs*)

decodes the given data.

Parameters **data** (`sequence[str, obj]`) – a sequence of two elements where the first item is the encoding format of the second item object

Return type `sequence[str, obj]`

Returns a sequence of two elements where the first item is the encoding format of the second

item object

encode (*data*, **args*, ***kwargs*)
encodes the given data.

Parameters **data** (`sequence[str, obj]`) – a sequence of two elements where the first item is the encoding format of the second item object

Return type `sequence[str, obj]`

Returns a sequence of two elements where the first item is the encoding format of the second item object

ColorPalette

class ColorPalette (*dat*, *int_decoder_dict=None*)
Bases: `object`

Provides the list of taurus colors equivalent to Tango colors.

format_SimStates (*var='T1'*)

has (*name*)

hex (*stoq*, *fmt='%06x'*, *fg=False*)
Returns the rgb colors as string of lowercase hexadecimal characters

htmlStyle (*htmlTag*, *stoq*)

name (*stoq*, *fg=False*)
Returns the name of the color.

number (*stoq*, *fg=False*)
Returns the colors as a number, suitable for conversion to hexadecimal as argument to QtGui.QColor.

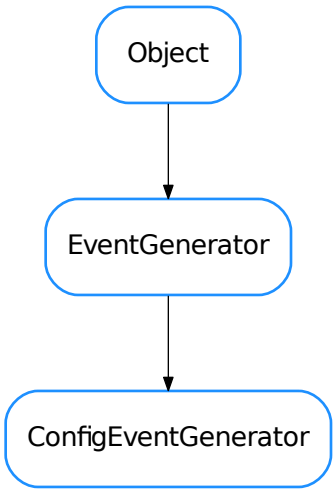
qtStyleSheet (*stoq*)

rgb (*stoq*, *fg=False*)
Returns a triplet of rgb colors in the range of 0 .. 255.

rgb_pair (*stoq*)
Returns pair of foreground and background colors.

size ()

ConfigEventGenerator



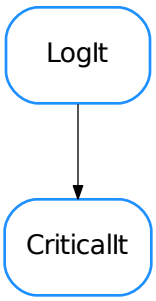
class **ConfigEventGenerator**(*name*, *events_active=True*)

Bases: `taurus.core.util.event.EventGenerator`

Manage configuration events

fireEvent(*val*, *event_val=None*)

CriticalIt



class **CriticalIt**(*showargs=False*, *showret=False*)

Bases: `taurus.core.util.log.LogIt`

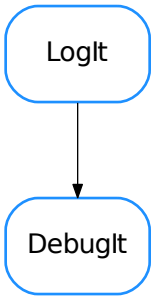
Specialization of LogIt for critical level messages. Example:

```
from taurus.core.util.log import Logger, CriticalIt
class Example(Logger):

    @CriticalIt()
    def go(self):
        print "Hello world"
```

See also:

LogIt

DebugIt



class DebugIt (*showargs=False*, *showret=False*)

Bases: taurus.core.util.log.LogIt

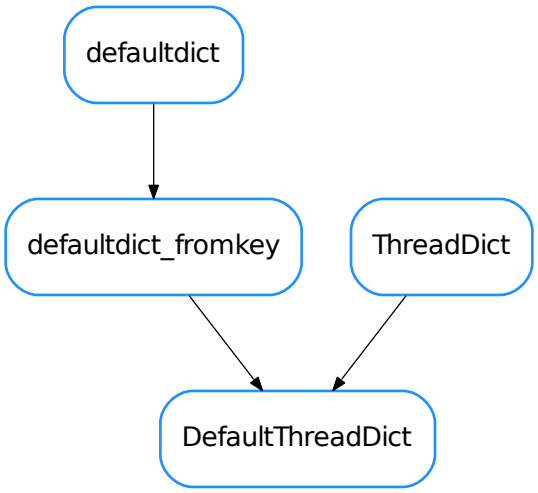
Specialization of LogIt for debug level messages. Example:

```
from taurus.core.util.log import Logger, DebugIt
class Example(Logger):

    @DebugIt()
    def go(self):
        print "Hello world"
```

See also:

LogIt

DefaultThreadDict



class DefaultThreadDict(*other=None*, *default_factory=None*, *read_method=None*, *write_method=None*, *timewait=0.1*, *threaded=True*)

Bases: `taurus.core.util.containers.defaultdict_fromkey`, `taurus.core.util.containers.ThreadDict`

a join venture between thread and default dict This class merges the two previous ones. @todo This two classes are not yet well integrated ... the way a new key is added to the dict must be rewritten explicitly.

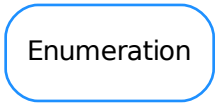
EnumException

class EnumException

Bases: `exceptions.Exception`

Exception thrown by *`Enumeration`* when trying to declare an invalid enumeration.

Enumeration



class Enumeration (*name*, *enumList*, *flaggable=False*, *no_doc=False*)

Bases: object

Enumeration class intended to provide the 'enum' feature present in many programming languages. The elements of the enumeration can be accessed in an "object member way" or as elements of a dictionary. Usage:

```
from taurus.core.util.enumeration import Enumeration

Volkswagen = Enumeration("Volkswagen",
    ["JETTA",
     "RABBIT",
     "BEETLE",
     ("THING", 400),
     "PASSAT",
     "GOLF",
     ("CABRIO", 700),
     "EURO_VAN",
     "CLASSIC_BEETLE",
     "CLASSIC_VAN"
     ])
```

In the command line:

```
>>> my_car = Volkswagen.BEETLE
>>> homer_car = Volkswagen.PASSAT

>>> print Volkswagen.BEETLE
2

>>> print Volkswagen['BEETLE']
2

>>>print Volkswagen.whatis(homer_car)
'PASSAT'
```

get (*i*)

Returns the element for the given key/value

has_key (*key*)

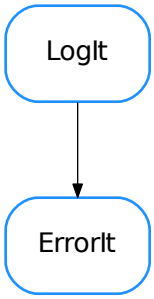
Determines if the enumeration contains the given key :param key: the key :type key: str :return: True if the key is in the enumeration or False otherswise :rtype: bool

keys ()

Returns an iterable containning the valid enumeration keys :return: an interable containning the valid enumeration keys :rtype: iter<str>

whatis (*value*)
Returns a string representation of the value in the enumeration. :param value: a valid enumeration element :return: a string representation of the given enumeration element :rtype: str

ErrorIt



class ErrorIt (*showargs=False*, *showret=False*)
Bases: `taurus.core.util.log.LogIt`

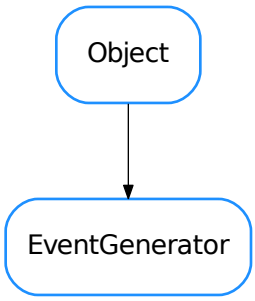
Specialization of LogIt for error level messages. Example:

```
from taurus.core.util.log import Logger, ErrorIt
class Example(Logger):

    @ErrorIt()
    def go(self):
        print "Hello world"
```

See also:

LogIt

EventGenerator



class **EventGenerator**(*name*, *events_active=True*)

Bases: taurus.core.util.object.Object

Base class capable of firing events

WaitTimeout = 0.1

fireEvent(*val*, *event_val=None*)

Fires an event. :param val: event value :type val: object

getEventsActive()

Determines is events are active :return: True if events are active or False otherwise :rtype: bool

isSubscribed(*cb*, *data=None*)

Determines if the given callback is registered for this event.

Parameters

- **cb** (*callable*) – a callable object
- **data** (*object*) – extra data to send each time an event is triggered on the given callback. Default is None

Returns True if callback is registered or False otherwise

Return type bool

lock()

Locks this event generator

read()

Read the last event

Returns the last event value

Return type object

setEventsActive(*events_active*)

(De)activates events on this event generator.

Parameters **events_active** (*bool*) – activate/deactivate events

subscribeEvent(*cb*, *data=None*, *with_first_event=True*)
Subscribes to the event

Parameters

- **cb** (*callable*) – a callable object
- **data** (*boolean*) – extra data to send each time an event is triggered on the given callback. Default is None.
- **with_first_event** – whether call the callback with the first event value (the most recent value) during the subscription process. Default is True.

unlock()
Unlocks this event generator

unsubscribeDeletedEvent(*cb_ref*)
for internal usage only

unsubscribeEvent(*cb*, *data=None*)
Unsubscribes the given callback from the event. If the callback is not a listener for this event a debug message is generated an nothing happens.

Parameters

- **cb** (*callable*) – a callable object
- **data** (*object*) – extra data to send each time an event is triggered on the given callback. Default is None

waitEvent(*val=None*, *equal=True*, *any=False*, *timeout=None*, *stack=None*)
Waits for an event to occur

Parameters

- **val** (*object*) – event value
- **equal** (*bool*) – check for equality. Default is True
- **any** (*bool*) – if True unblock after first event, not matter what value it has. Default is False.
- **timeout** (*float*) – maximum time to wait (seconds). Default is None meaning wait forever.
- **stack** – For internal usage only.

Returns the value of the event that unblocked the wait

Return type object

EventListener

class EventListener

Bases: object

A class that listens for an event with a specific value

Note: Since this class stores for each event value the last timestamp when it occured, it should only be used for events for which the event value domain (possible values) is limited and well known (ex: an enum)

clearEventSet()

Clears the internal event buffer

fireEvent(*v*)

Notifies that a given event has arrived This function is protected inside with the object's lock. Do NOT call this function when you have the lock acquired on this object.

Parameters **v** (*object*) – event value

lock()

Locks this event listener

unlock()

Unlocks this event listener

waitEvent(*val*, *after=0*, *equal=True*)

Wait for an event with the given value. You MUST protect this function with this object's lock before calling this method and always unlock it afterward, of course:

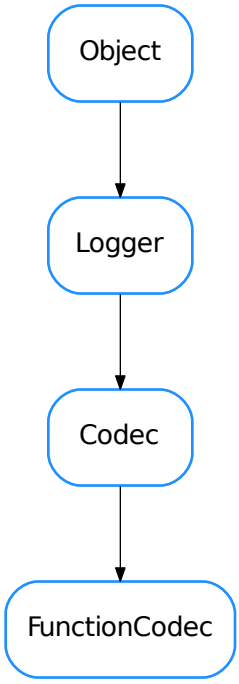
```
from taurus.core.util.event import EventListener

class MyEvtListener(EventListener):
    # Your specific listener code here
    pass

evt_listener = EventListener()
try:
    evt_listener.lock()
    t = time.time()
    go()
    evt_listener.waitEvent(Stop, t)
finally:
    evt_listener.unlock()
```

Parameters

- **val** (*object*) – value to compare

- **after** (*float*) – timestamp. wait for events comming after the given time. default value is 0 meaning any event after Jan 1, 1970
- **equal** (*bool*) – compare for equality. equal=True means an event with the given value, equal=False means any event which as a different value

FunctionCodec



class FunctionCodec (*func_name*)

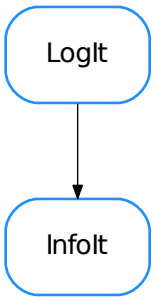
Bases: `taurus.core.util.codecs.Codec`

A generic function codec

decode (*data*, **args*, ***kwargs*)

encode (*data*, **args*, ***kwargs*)

InfoIt



class **InfoIt**(*showargs=False*, *showret=False*)

Bases: `taurus.core.util.log.LogIt`

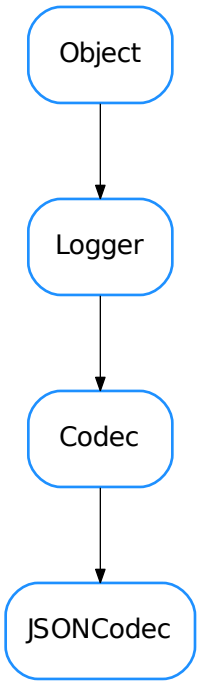
Specialization of LogIt for info level messages. Example:

```
from taurus.core.util.log import Logger, InfoIt
class Example(Logger):

    @InfoIt()
    def go(self):
        print "Hello world"
```

See also:

LogIt

JSONCodec



class JSONCodec

Bases: `taurus.core.util.codecs.Codec`

A codec able to encode/decode to/from json format. It uses the `json` module.

Example:

```
>>> from taurus.core.util.codecs import CodecFactory

>>> cf = CodecFactory()
>>> codec = cf.getCodec('json')
>>>
>>> # first encode something
>>> data = { 'hello' : 'world', 'goodbye' : 1000 }
>>> format, encoded_data = codec.encode(("", data))
>>> print encoded_data
'{"hello": "world", "goodbye": 1000}'
>>>
>>> # now decode it
>>> format, decoded_data = codec.decode((format, encoded_data))
>>> print decoded_data
{'hello': 'world', 'goodbye': 1000}
```

decode (*data*, **args*, ***kwargs*)

decodes the given data from a json string.

Parameters **data** (sequence[str, obj]) – a sequence of two elements where the first item is the encoding format of the second item object

Return type sequence[str, obj]

Returns a sequence of two elements where the first item is the encoding format of the second item object

encode(*data*, **args*, ***kwargs*)

encodes the given data to a json string. The given data **must** be a python object that json is able to convert.

Parameters **data** (sequence[str, obj]) – a sequence of two elements where the first item is the encoding format of the second item object

Return type sequence[str, obj]

Returns a sequence of two elements where the first item is the encoding format of the second item object

LIFO

class **LIFO**(*max=10*)

Bases: object

append(*elem*)

clear()

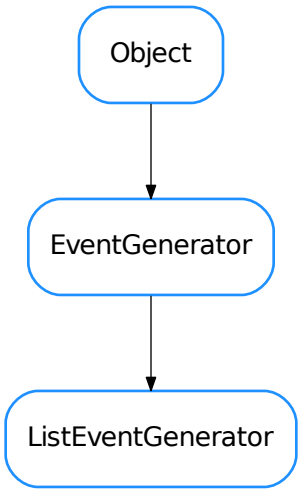
extend(*lst*)

get()

getCopy()

pop(*index=0*)

ListEventGenerator



class **ListEventGenerator**(*name*, *events_active=True*)

Bases: `taurus.core.util.event.EventGenerator`

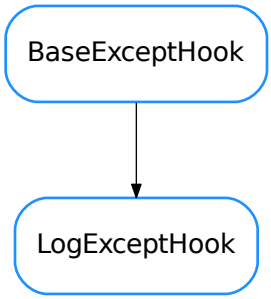
Manage list events, detecting changes in the list

fireEvent(*val*)

Notifies that a given event has arrived This function is protected inside with the object's lock. Do NOT call this function when you have the lock acquired on this object.

Parameters **val** (*object*) – event value

LogExceptHook



class LogExceptHook (*hook_to=None*, *name=None*, *level=40*)

Bases: taurus.core.util.excepthook.BaseExceptHook

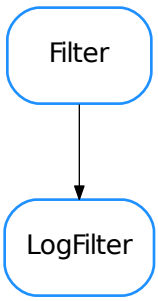
A callable class that acts as an excepthook that logs the exception in the python logging system.

Parameters

- **hook_to** (*callable*) – callable excepthook that will be called at the end of this hook handling [default: None]
- **name** (*str*) – logger name [default: None meaning use class name]
- **level** (*int*) – log level [default: logging.ERROR]

report (**exc_info*)

LogFilter



class LogFilter (*level*)

Bases: logging.Filter

Experimental log filter

filter (*record*)

LogIt

class LogIt (*level=10*, *showargs=False*, *showret=False*, *col_limit=0*)

Bases: object

A function designed to be a decorator of any method of a Logger subclass. The idea is to log the entrance and exit of any decorated method of a Logger subclass. Example:

```
from taurus.core.util.log import Logger, LogIt

class Example(Logger):

    @LogIt(Logger.Debug)
    def go(self):
        print "Hello world "
```

This will generate two log messages of Debug level, one before the function go is called and another when go finishes. Example output:

```
MainThread     DEBUG    2010-11-15 15:36:11,440 Example: -> go
Hello world of mine
MainThread     DEBUG    2010-11-15 15:36:11,441 Example: <- go
```

This decorator can receive two optional arguments **showargs** and **showret** which are set to False by default. Enabling them will had verbose infomation about the parameters and return value. The following example:

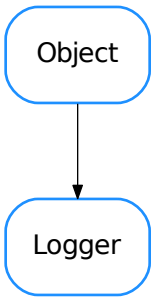
```
from taurus.core.uti.log import Logger, LogIt

class Example(Logger):

    @LogIt(Logger.Debug, showargs=True, showret=True)
    def go(self, msg):
        msg = "Hello world",msg
        print msg
        return msg
```

would generate an output like:

```
MainThread     DEBUG    2010-11-15 15:42:02,353 Example: -> go('of mine',)
Hello world of mine
MainThread     DEBUG    2010-11-15 15:42:02,353 Example: <- go = Hello world of␣
↪mine
```

Note: it may happen that in these examples that the output of the method appears before or after the log messages. This is because log messages are, by default, written to the *stardard error* while the print message inside the go method outputs to the *standard ouput*. On many systems these two targets are not synchronized.

Logger



class Logger (*name=''*, *parent=None*, *format=None*)
Bases: taurus.core.util.object.Object

The taurus logger class. All taurus pertinent classes should inherit directly or indirectly from this class if they need taurus logging facilities.

Critical = 50
Critical message level (constant)

Debug = 10
Debug message level (constant)

DftLogFormat = <logging.Formatter object>
Default log format (constant)

DftLogLevel = 20
Default log level (constant)

DftLogMessageFormat = '%(threadName)-14s %(levelname)-8s %(asctime)s %(name)s: %(message)s'
Default log message format (constant)

Error = 40
Error message level (constant)

Fatal = 50
Fatal message level (constant)

Info = 20
Info message level (constant)

Trace = 5
Trace message level (constant)

Warning = 30
Warning message level (constant)

addChild (*child*)
Adds a new logging child

Parameters **child** (Logger) – the new child

classmethod addLevelName(*level_no*, *level_name*)
Registers a new log level

Parameters

- **level_no** (int) – the level number
- **level_name** (str) – the corresponding name

addLogHandler(*handler*)
Registers a new handler in this object's logger

Parameters **handler** (Handler) – the new handler to be added

classmethod addRootLogHandler(*h*)
Adds a new handler to the root logger

Parameters **h** (Handler) – the new log handler

changeLogName(*name*)
Change the log name for this object.

Parameters **name** (str) – the new log name

cleanUp()
The cleanUp. Default implementation does nothing Overwrite when necessary

copyLogHandlers(*other*)
Copies the log handlers of other object to this object

Parameters **other** (object) – object which contains 'log_handlers'

critical(*msg*, **args*, ***kw*)
Record a critical message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.critical().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

debug(*msg*, **args*, ***kw*)
Record a debug message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.debug().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

deprecated(*msg=None*, *dep=None*, *alt=None*, *rel=None*, *dbg_msg=None*, *_callerinfo=None*, ***kw*)
Record a deprecated warning message in this object's logger. If message is not passed, a estandard deprecation message is constructued using dep, alt, rel arguments. Also, an extra debug message can be recorded, followed by traceback info.

Parameters

- **msg** (str) – the message to be recorded (if None passed, it will be constructed using dep (and, optionally, alt and rel)
- **dep** (str) – name of deprecated feature (in case msg is None)

- **alt** (str) – name of alternative feature (in case msg is None)
- **rel** (str) – name of release from which the feature was deprecated (in case msg is None)
- **dbg_msg** (str) – msg for debug (or None to log only the warning)
- **_callerinfo** – for internal use only. Do not use this argument.
- **kw** – any additional keyword arguments, are passed to logging.Logger.warning()

classmethod disableLogOutput()
Disables the logging.StreamHandler which dumps log records, by default, to the stderr.

classmethod enableLogOutput()
Enables the logging.StreamHandler which dumps log records, by default, to the stderr.

error(*msg*, **args*, ***kw*)
Record an error message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.error().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

exception(*msg*, **args*)
Log a message with severity 'ERROR' on the root logger, with exception information.. Accepted *args* are the same as logging.Logger.exception().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments

fatal(*msg*, **args*, ***kw*)
Record a fatal message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.fatal().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

flushOutput()
Flushes the log output

getChildren()
Returns the log children for this object

Return type Logger

Returns the list of log children

classmethod getLogFormat()
Retuns the current log message format (the root log format)

Return type str

Returns the log message format

getLogFullName()
Gets the full log name for this object

Return type str

Returns the full log name

classmethod getLogLevel()
Retuns the current log level (the root log level)

Return type int

Returns a number representing the log level

getLogName()
Gets the log name for this object

Return type str

Returns the log name

getLogObj()
Returns the log object for this object

Return type Logger

Returns the log object

classmethod getLogger(*name=None*)

getParent()
Returns the log parent for this object or None if no parent exists

Return type Logger or None

Returns the log parent for this object

classmethod getRootLog()
Retuns the root logger

Return type Logger

Returns the root logger

info(*msg*, **args*, ***kw*)
Record an info message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.info().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

classmethod initRoot()
Class method to initialize the root logger. Do **NOT** call this method directly in your code

log(*level*, *msg*, **args*, ***kw*)
Record a log message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.log().

Parameters

- **level** (int) – the record level
- **msg** (str) – the message to be recorded

- **args** – list of arguments
- **kw** – list of keyword arguments

log_format = <logging.Formatter object>
Default log message format

log_level = 20
Current global log level

classmethod removeRootLogHandler(*h*)
Removes the given handler from the root logger

Parameters **h** (Handler) – the handler to be removed

classmethod resetLogFormat()
Resets the log message format (the root log format)

classmethod resetLogLevel()
Resets the log level (the root log level)

root_init_lock = <thread.lock object>
Internal usage

root_inited = True
Internal usage

classmethod setLogFormat(*format*)
sets the new log message format

Parameters **level** (str) – the new log message format

classmethod setLogLevel(*level*)
sets the new log level (the root log level)

Parameters **level** (int) – the new log level

stack(*target=5*)
Log the usual stack information, followed by a listing of all the local variables in each frame.

Parameters **target** (int) – the log level assigned to the record

Return type str

Returns The stack string representation

stream_handler = <logging.StreamHandler object>
the main stream handler

syncLog()
Synchronises the log output

trace(*msg*, **args*, ***kw*)
Record a trace message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.log().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

traceback(*level=5*, *extended=True*)
Log the usual traceback information, followed by a listing of all the local variables in each frame.

Parameters

- **level** (int) – the log level assigned to the traceback record
- **extended** (bool) – if True, the log record message will have multiple lines

Return type str

Returns The traceback string representation

warning(*msg*, **args*, ***kw*)

Record a warning message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.warning().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

LoopList

class LoopList(*itemlist=[]*)

Bases: object

this class provides an effectively cyclic list. It can be used, e.g., for storing colors or pen properties to be changed automatically in a plot

A LoopList stores an internal index to remember the last accessed item in the list It provides previous(), current() and next() methods that return the previous,current and next items in the list. The method allItems() returns a copy of all items contained in the list. The index can be accessed by setCurrentIndex() and getCurrentIndex() (setCurrentIndex(i) additionally returns new current item) Items can be accessed ***without modifying the current index*** by using llist[i] and llist[i]=x syntax len(llist) returns the **period** of the list.

Note: only basic methods of lists are implemented for llists. In particular, the following are **not** implemented:

•slicing

•resizing (append, insert, del,...)

•binary operators (+,*,...)

..note:

```
it can be used for loops, but the loop will be infinite unless other
condition is used for exiting it:
```

```
    - for item in llist: print item  # This is a infinite loop!!
    - for i in range(len(llist)):print  llist[i]  #This is not infinite
      since len(llist) returns the period of the list
```

allItems()
returns the items list (one period)

current()
returns current item

getCurrentIndex()
returns the current index

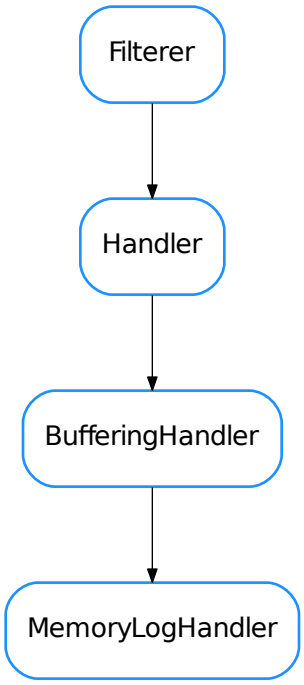
next()
advances one item in the list and returns it

previous()
goes one item back in the list and returns it

setCurrentIndex(*index*)
sets current index (and returns the corresponding item)

setItemList(*itemlist*)
sets the item list

MemoryLogHandler



class **MemoryLogHandler** (*capacity=1000*)

Bases: `list`, `logging.handlers.BufferingHandler`

An experimental log handler that stores temporary records in memory. When flushed it passes the records to another handler

close ()

Closes this handler

flush ()

Flushes this handler

shouldFlush (*record*)

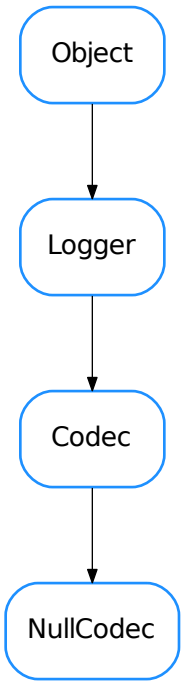
Determines if the given record should trigger the flush

Parameters **record** (`LogRecord`) – a log record

Return type `bool`

Returns wheter or not the handler should be flushed

NullCodec



class **NullCodec**

Bases: `taurus.core.util.codecs.Codec`

decode (*data*, **args*, ***kwargs*)

decodes with Null encoder. Just returns the given data

Parameters **data** (sequence[str, obj]) – a sequence of two elements where the first item is the encoding format of the second item object

Return type sequence[str, obj]

Returns a sequence of two elements where the first item is the encoding format of the second item object

encode(*data*, **args*, ***kwargs*)
encodes with Null encoder. Just returns the given data

Parameters **data** (sequence[str, obj]) – a sequence of two elements where the first item is the encoding format of the second item object

Return type sequence[str, obj]

Returns a sequence of two elements where the first item is the encoding format of the second item object

Object

class Object
Bases: object

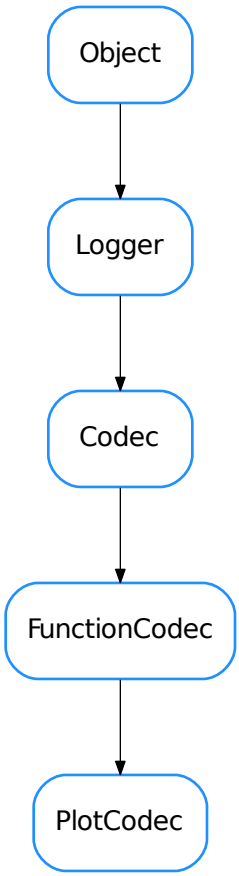
call__init__(*klass*, **args*, ***kw*)
Method to be called from subclasses to call superclass corresponding __init__ method. This method ensures that classes from diamond like class hierarquies don't call their super classes __init__ more than once.

call__init__wo_kw(*klass*, **args*)
Same as call__init__ but without keyword arguments because PyQT does not support them.

getAttrDict()

updateAttrDict(*other*)

PlotCodec



class PlotCodec

Bases: taurus.core.util.codecs.FunctionCodec

A specialization of the *FunctionCodec* for plot function

SafeEvaluator

class SafeEvaluator(*safedict=None*, *defaultSafe=True*)

Bases: object

This class provides a safe eval replacement.

The method eval() will only evaluate the expressions considered safe (whitelisted). By default it has a whitelist of mathematical expressions that can be turn off using defaultSafe=False at init

The user can add more safe functions passing a safedict to the addSafe() or init methods.

Functions can be removed by name using removeSafe()

Note: In order to use variables defined outside, the user must explicitly declare them safe.

addSafe(*safedict*, *permanent=False*)

The values in safedict will be evaluable (whitelisted) The safedict is as follows: {"eval_name":object, ...}. The evaluator will interpret eval_name as object.

eval(*expr*)

safe eval

getSafe()

returns the currently whitelisted expressions

removeSafe(*name*, *permanent=False*)

Removes an object from the whitelist

resetSafe()

restores the safe dict with wich the evaluator was instantiated

Singleton

class Singleton

Bases: object

This class allows Singleton objects The __new__ method is overriden to force Singleton behaviour. The Singleton is created for the lowest subclass. Usage:

```
from taurus.core.util.singleton import Singleton

class MyManager(Singleton):

    def init(self, *args, **kwargs):
        print "Singleton initialization"
```

command line:

```
>>> manager1 = MyManager()
Singleton initialization

>>> manager2 = MyManager()

>>> print(manager1,manager2)
<__main__.MyManager object at 0x9c2a0ec>
<__main__.MyManager object at 0x9c2a0ec>
```

Notice that the two instances of manager point to the same object even though you *tried* to construct two instances of MyManager.

Warning: although __new__ is overriden __init__ is still being called for each instance=Singleton()

init (**p*, ***k*)

ThreadDict

class ThreadDict (*other=None*, *read_method=None*, *write_method=None*, *timewait=0.1*, *threaded=True*)

Bases: dict

Thread safe dictionary with redefinable read/write methods and a background thread for hardware update. All methods are thread-safe using *@self_lock* decorator.

Note: any method decorated in this way CANNOT call other decorated methods! All values of the dictionary will be automatically updated in a separate Thread using read_method provided. Any value overwritten in the dict should launch the write_method.

Briefing:

```
a[2] equals to a[2]=read_method(2)
a[2]=1 equals to a[2]=write_method(2,1)
```

alive()

append(**args*, ***kwargs*)

copy(**args*, ***kwargs*)

get(**args*, ***kwargs*)

get_last_cycle_start(**args*, ***kwargs*)

get_last_update(**args*, ***kwargs*)

get_timewait(**args*, ***kwargs*)

has_key(**args*, ***kwargs*)

items(**args*, ***kwargs*)

iteritems(**args*, ***kwargs*)

iterkeys(**args*, ***kwargs*)

itervalues(**args*, ***kwargs*)

keys(**args*, ***kwargs*)

pop(**args*, ***kwargs*)

run()

set_last_cycle_start(**args*, ***kwargs*)

set_last_update(**args*, ***kwargs*)

set_timewait(**args*, ***kwargs*)

start()

stop()

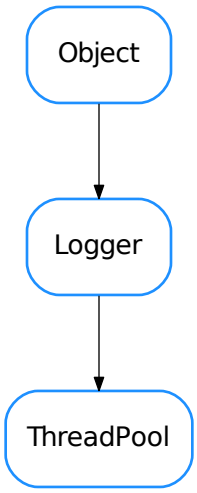
threadkeys(**args*, ***kwargs*)

tracer(*text*)

update(**args*, ***kwargs*)

values(**args*, ***kwargs*)

ThreadPool



class ThreadPool (*name=None*, *parent=None*, *Psize=20*, *Qsize=20*, *daemons=True*)

Bases: `taurus.core.util.log.Logger`

NoJob = (None, None, None, None, None, None)

add (*job*, *callback=None*, **args*, ***kw*)

getNumOfBusyWorkers ()

Get the number of workers that are in busy mode.

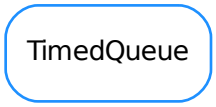
join ()

qsize

size

number of threads

TimedQueue



class `TimedQueue`(*arg=None*)

Bases: `list`

A FIFO that keeps all the values introduced at least for a given time. Applied to some device servers, to force States to be kept at least a minimum time. Previously named as PyTango_utils.device.StateQueue pop(): The value is removed only if delete_time has been reached. at least 1 value is always kept in the list

append(*obj*, *keep=15*)

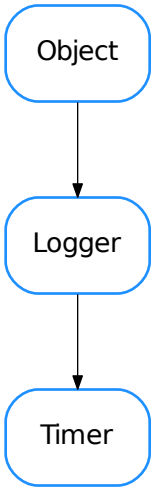
Inserts a tuple with (value,insert_time,delete_time=now+keep)

index(*obj*)

pop(*index=0*)

Returns the indicated value, or the first one; but removes only if delete_time has been reached. All values are returned at least once. When the queue has only a value, it is not deleted.

`Timer`



class `Timer`(*interval*, *function*, *parent*, *strict_timing=True*, **args*, ***kwargs*)

Bases: `taurus.core.util.log.Logger`

Timer Object.

Interval in seconds (The argument may be a floating point number for subsecond precision). If strict_timing is True, the timer will try to compensate for drifting due to the time it takes to execute function in each loop.

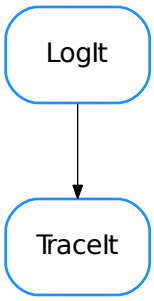
start()

Start Timer Object

stop()

Stop Timer Object

TraceIt



class TraceIt (*showargs=False*, *showret=False*)

Bases: taurus.core.util.log.LogIt

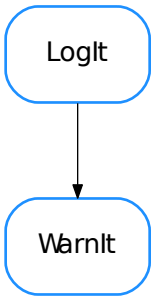
Specialization of LogIt for trace level messages. Example:

```
from taurus.core.util.log import Logger, TraceIt
class Example(Logger):

    @TraceIt()
    def go(self):
        print "Hello world"
```

See also:

LogIt

WarnIt



class **WarnIt** (*showargs=False*, *showret=False*)

Bases: taurus.core.util.log.LogIt

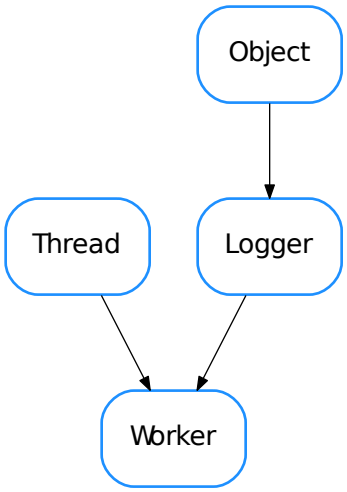
Specialization of LogIt for warn level messages. Example:

```
from taurus.core.util.log import Logger, WarnIt
class Example(Logger):

    @WarnIt()
    def go(self):
        print "Hello world"
```

See also:

LogIt

Worker



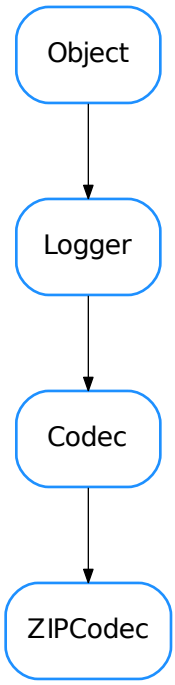
class **Worker** (*pool*, *name=None*, *daemon=True*)

Bases: threading.Thread, taurus.core.util.log.Logger

isBusy()

run()

ZIPCodec



class ZIPCodec

Bases: `taurus.core.util.codecs.Codec`

A codec able to encode/decode to/from gzip format. It uses the `zlib` module

Example:

```
>>> from taurus.core.util.codecs import CodecFactory

>>> # first encode something
>>> data = 100 * "Hello world\n"
>>> cf = CodecFactory()
>>> codec = cf.getCodec('zip')
>>> format, encoded_data = codec.encode(("", data))
>>> print len(data), len(encoded_data)
1200, 31
>>> format, decoded_data = codec.decode((format, encoded_data))
>>> print decoded_data[20]
'Hello world\nHello wo'
```

decode (*data*, **args*, ***kwargs*)

decodes the given data from a gzip string.

Parameters **data** (`sequence[str, obj]`) – a sequence of two elements where the first item is the encoding format of the second item object

Return type `sequence[str, obj]`

Returns a sequence of two elements where the first item is the encoding format of the second item object

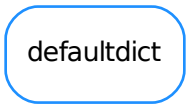
encode(*data*, **args*, ***kwargs*)
encodes the given data to a gzip string. The given data **must** be a string

Parameters **data** (`sequence[str, obj]`) – a sequence of two elements where the first item is the encoding format of the second item object

Return type `sequence[str, obj]`

Returns a sequence of two elements where the first item is the encoding format of the second item object

defaultdict



class defaultdict
Bases: `dict`

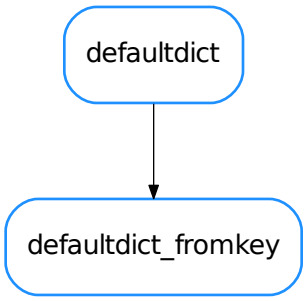
defaultdict(default_factory[, ...]) –> dict with default factory

The default factory is called without arguments to produce a new value when a key is not present, in __getitem__ only. A defaultdict compares equal to a dict with the same items. All remaining arguments are treated the same as if they were passed to the dict constructor, including keyword arguments.

copy() → a shallow copy of D.

default_factory
Factory for default value called by __missing__().

defaultdict_fromkey



class defaultdict_fromkey

Bases: collections.defaultdict

Creates a dictionary with a default_factory function that creates new elements using key as argument. Usage : new_dict = defaultdict_fromkey(method); where method like (lambda key: return new_obj(key)) Each time that new_dict[key] is called with a key that doesn't exist, method(key) is used to create the value Copied from PyAlarm device server

- *ArrayBuffer*
- *AttributeEventIterator*
- *AttributeEventWait*
- *BZ2Codec*
- *BoundMethodWeakref*
- *CaselessDefaultDict*
- *CaselessDict*
- *CaselessList*
- *CaselessWeakValueDict*
- *CircBuf*
- *Codec*
- *CodecFactory*
- *CodecPipeline*
- *ColorPalette*
- *ConfigEventGenerator*
- *CriticalIt*
- *DebugIt*
- *DefaultThreadDict*
- *EnumException*
- *Enumeration*
- *ErrorIt*
- *EventGenerator*
- *EventListener*
- *FunctionCodec*
- *InfoIt*
- *JSONCodec*
- *LIFO*
- *ListEventGenerator*

- *LogExceptHook*
- *LogFilter*
- *LogIt*
- *Logger*
- *LoopList*
- *MemoryLogHandler*
- *NullCodec*
- *Object*
- *PlotCodec*
- *SafeEvaluator*
- *Singleton*
- *ThreadDict*
- *ThreadPool*
- *TimedQueue*
- *Timer*
- *TraceIt*
- *WarnIt*
- *Worker*
- *ZIPCodec*
- *defaultdict*
- *defaultdict_fromkey*

Functions

CallableRef(*object*, *del_cb=None*)
This function returns a callable weak reference to a callable object. Object can be a callable object, a function or a method.

Parameters

- **object** (*callable object*) – a callable object
- **del_cb** (*callable object or None*) – calback function. Default is None meaning to callback.

Returns a weak reference for the given callable

Return type *BoundMethodWeakref* or weakref.ref

critical(*msg*, **args*, ***kw*)

debug(*msg*, **args*, ***kw*)

deprecated(**args*, ***kw*)

deprecation_decorator(*func=None*, *alt=None*, *rel=None*, *dbg_msg=None*)
decorator to mark methods as deprecated

dictFromSequence(*seq*)
Translates a sequence into a dictionary by converting each to elements of the sequence (k,v) into a k:v pair in the dictionary

Parameters **seq** (sequence) – any sequence object

Return type dict

Returns dictionary built from the given sequence

error(*msg*, **args*, ***kw*)

fatal(*msg*, **args*, ***kw*)

getDictAsTree(*dct*)

This method will print a recursive dict in a tree-like shape:

```
>>> print getDictAsTree({'A':{'B':[1,2],'C':[3]}})
```

getSystemUserName()

Finds out user inf (currently just the logged user name) for Windows and Posix machines. sets a USER_NAME variable containing the logged user name defines a UNKNOWN_USER variable to which username falls back.

Return type str

Returns current user name

info(*msg*, **args*, ***kw*)

propertx(*fct*)

Decorator to simplify the use of property. Like @property for attrs who need more than a getter. For getter only property use @property.

adapted from http://code.activestate.com/recipes/502243/

self_locked(*func*, *reentrant=True*)

Decorator to make thread-safe class members Decorator to create thread-safe objects.

Warning:

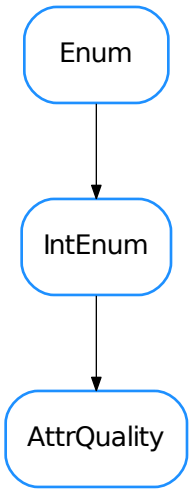
- With Lock() this decorator should not be used to decorate nested functions; it will cause Deadlock!
- With RLock this problem is avoided ... but you should rely more on python threading

trace(*msg*, **args*, ***kw*)

warning(*msg*, **args*, ***kw*)

Classes

AttrQuality



class AttrQuality

Bases: enum.IntEnum

Enumeration of quality states for Taurus attributes. based on This is the Taurus equivalent to PyTango.AttrQuality. The members present in PyTango are also defined here with the same values, allowing equality comparisons with `PyTango.AttrQuality` (but not identity checks!):

```
from taurus.core import AttrQuality as Q1
from PyTango import AttrQuality as Q2

Q1.ATTR_ALARM == Q2.ATTR_ALARM                    # --> True
Q1.ATTR_ALARM in (Q2.ATTR_ALARM, Q2.ATTR_ALARM)  # --> True
Q1.ATTR_ALARM == 2                                # --> True
Q1.ATTR_ALARM is 2                                # --> False
Q1.ATTR_ALARM is Q2.ATTR_ALARM                    # --> False
```

ATTR_ALARM = 2

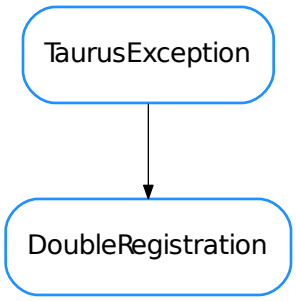
ATTR_CHANGING = 3

ATTR_INVALID = 1

ATTR_VALID = 0

ATTR_WARNING = 4

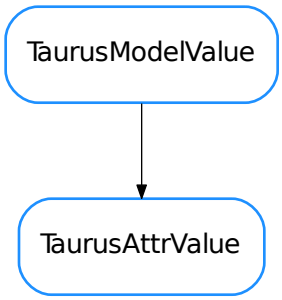
DoubleRegistration



class DoubleRegistration (*description*, *code=None*)

Bases: `taurus.core.taurusexception.TaurusException`

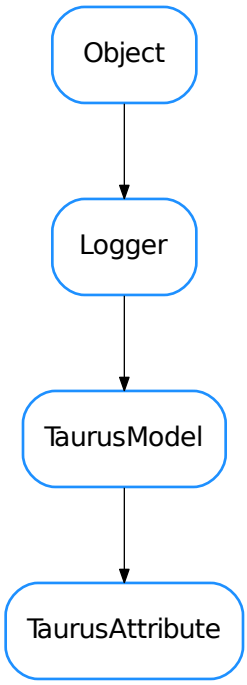
TaurusAttrValue



class TaurusAttrValue

Bases: `taurus.core.taurusbasetypes.TaurusModelValue`

TaurusAttribute



class **TaurusAttribute**(*name*, *parent*, ***kwargs*)

Bases: taurus.core.taurusmodel.TaurusModel

DftTimeToLive = 10000

activatePolling(*period*, *unsubscribe_evts=False*, *force=False*)

activate polling for attribute.

Parameters **period** (*int*) – polling period (in miliseconds)

alarms

areStrValuesEqual(*v1*, *v2*)

classmethod **buildModelName**(*parent_model*, *relative_name*)

build an 'absolute' model name from the parent model and the 'relative' name. - If parent_model is a TaurusDevice, the return is a composition of the database model name and its device name - If parent_model is a TaurusAttribute, the relative name is ignored and the parent name is returned

Note: This is a basic implementation. You may need to reimplement this for a specific scheme if it supports "useParentModel".

changePollingPeriod(*period*)

change polling period to period miliseconds

cleanUp()

deactivatePolling(*maintain_enabled=False*)
unregister attribute from polling

decode(*attr_value*)

defaultFragmentName = 'rvalue'

description

disablePolling()
Disable polling and if polling is active also deactivate it. See *isPollingEnabled()* for clarification of what enabled polling means.

enablePolling(*force=False*)
Enable polling. See *isPollingEnabled()* for clarification of what enabled polling means.

Parameters **force** (bool) – True also activates polling (see: *activatePolling()*)

encode(*value*)

getAlarms(*cache=True*)

getDataFormat(*cache=True*)

getDisplayDescrObj(*cache=True*)

getDisplayDescription(*cache=True*)

getLabel(*cache=True*)

getMaxAlarm(*cache=True*)

getMaxRange(*cache=True*)

getMaxWarning(*cache=True*)

getMinAlarm(*cache=True*)

getMinRange(*cache=True*)

getMinWarning(*cache=True*)

classmethod getNameValidator()

getPollingPeriod()
returns the polling period

getRange(*cache=True*)

classmethod getTaurusElementType()

getType(*cache=True*)

getValueObj(*cache=True*)

getWarnings(*cache=True*)

hasEvents()

isBoolean(*cache=True*)

isNumeric()

isPolled()

isPollingActive()
Indicate whether polling is active. Active polling means that a periodic timer poll the attribute. By default the attribute creation does not activate polling.

Return type bool

Returns whether polling is active

See *activatePolling()*, *disablePolling()*

isPollingEnabled()
Indicate whether polling was activated/deactivated by user. Enabled polling does not mean that it is active - periodically poll the attribute. By default the attribute creation enables polling.

Return type bool

Returns whether polling is enabled

See *enablePolling()*, *disablePolling()*

isPollingForced()

isState(**args*, ***kwargs*)
Deprecated since version >4.0.1: Use .type==DataType.DevState instead

isUsingEvents()

isWritable(*cache=True*)

label

poll()

quality

range

read(*cache=True*)

rvalue

setAlarms(**limits*)

setLabel(*lbl*)

setRange(**limits*)

setWarnings(**limits*)

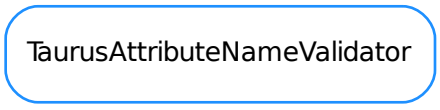
time

warnings

write(*value*, *with_read=True*)

wvalue

TaurusAttributeNameValidator



class TaurusAttributeNameValidator

Bases: taurus.core.taurusvalidator._TaurusBaseValidator

Base class for Attribute name validators. The namePattern will be composed from URI segments as follows:

<scheme>:[<authority>/]<path>[?<query>][#<fragment>]

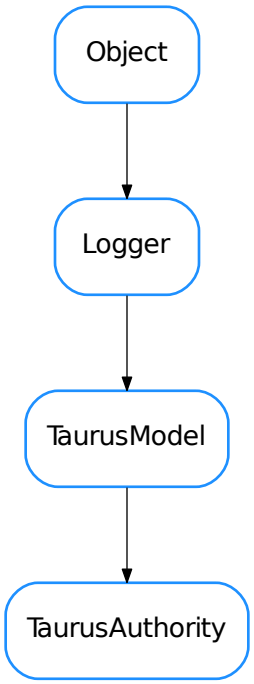
Derived classes must provide attributes defining a regexp string for each URI segment (they can be empty strings):

- scheme
- authority
- path
- query
- fragment

Additionally, the namePattern resulting from composing the above segments must contain a named group called "attrname" (normally within the path segment).

pattern = '^(?P<scheme>%(scheme)s):((?P<authority>%(authority)s)($|(?=[/#?])))?(?P<path>%(path)s)(\\?(?P<query

TaurusAuthority



class TaurusAuthority(*complete_name*, *parent=None*)

Bases: taurus.core.taurusmodel.TaurusModel

classmethod buildModelName(*parent_model*, *relative_name*)
build an 'absolute' model name from the parent name and the 'relative' name. parent_model is ignored since there is nothing above the Authority object

Note: This is a basic implementation. You may need to reimplement this for a specific scheme if it supports "useParentModel".

cleanUp()

description

getChildObj(*child_name*)

getDevice(*devname*)
Returns the device object given its name

getDisplayDescrObj(*cache=True*)

getDisplayDescription(*cache=True*)

classmethod getNameValidator()

classmethod getTaurusElementType()

TaurusAuthorityNameValidator

class TaurusAuthorityNameValidator
Bases: taurus.core.taurusvalidator._TaurusBaseValidator

Base class for Authority name validators. The namePattern will be composed from URI segments as follows:

<scheme>:<authority>[/<path>][?<query>][#<fragment>]

Derived classes must provide attributes defining a regexp string for each URI segment (they can be empty strings):

- scheme
- authority
- path
- query
- fragment

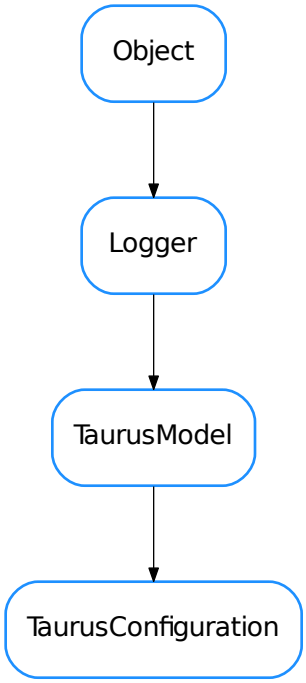
getNames(*name*, *factory=None*)
basic implementation for getNames for authorities. You may reimplement it in your scheme if required

pattern = '^(?P<scheme>%(scheme)s):(?P<authority>%(authority)s)((?=/)(?P<path>%(path)s))?(\\?(?P<query>%(qu

TaurusConfigValue

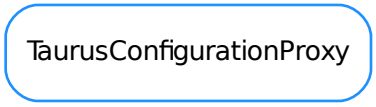
class TaurusConfigValue (**args*, ***kwargs*)
Bases: `object`

TaurusConfiguration



class TaurusConfiguration (**args*, ***kwargs*)
Bases: `taurus.core.taurusmodel.TaurusModel`

TaurusConfigurationProxy



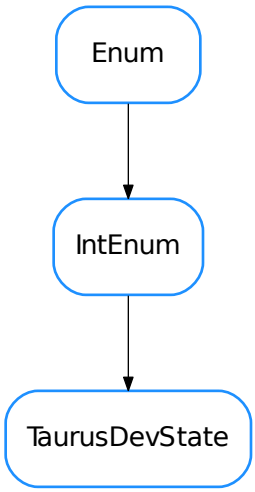
class TaurusConfigurationProxy (**args*, ***kwargs*)

Bases: object

TaurusAttribute has a reference to TaurusConfiguration and it should also have a reference to TaurusAttribute. To solve this cyclic dependency, TaurusConfiguration has a weak reference to TaurusAttribute. But then we must be sure that no other references to TaurusConfiguration exist so that no one tries to use it after its TaurusAttribute has disappeared. That's why to the outside world we don't give access to it directly but to objects of this new TaurusConfigurationProxy class.

getRealConfigClass()

TaurusDevState



class TaurusDevState

Bases: enum.IntEnum

Enumeration of possible states of *taurus.core.TaurusDevice* objects. This is returned, e.g. by *TaurusDevice.state()*.

The description of the values of this enumeration is:

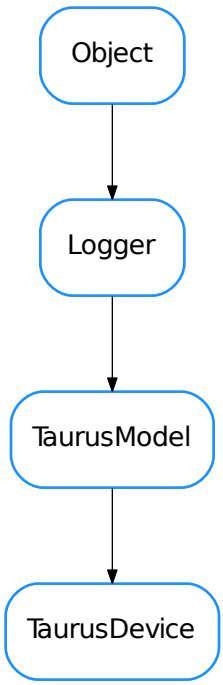
- Ready: the device can be operated by the user and could even be involved in some operation.
- NotReady: the device can not be operated by the user (e.g. due to still being initialized, or due to a device failure,...)
- Undefined: it is not possible to retrieve a coherent state from the device (e.g. due to communication, or to contradictory internal states, ...)

NotReady = 2

Ready = 1

Undefined = 4

TaurusDevice



class TaurusDevice (*name*, ***kw*)

Bases: `taurus.core.taurusmodel.TaurusModel`

A Device object. Different schemes may assign different roles, but in general it is a parent of Taurus Attribute objects and a child of a Taurus Authority

classmethod buildModelName (*parent_model*, *relative_name*)

build an 'absolute' model name from the parent model and the 'relative' name. - If parent_model is a TaurusAuthority, the return is a composition of the authority model name and the device name - If parent_model is a TaurusDevice, the relative name is ignored and the parent name is returned

Note: This is a basic implementation. You may need to reimplement this for a specific scheme if it supports "useParentModel".

description

getChildObj(*child_name*)

getDisplayDescrObj(*cache=True*)

classmethod getNameValidator()

classmethod getTaurusElementType()

poll(*attrs*, *asynch=False*, *req_id=None*)
Polling certain attributes of the device. This default implementation simply polls each attribute one by one

state
Returns a scheme-agnostic representation of the state of a Taurus device. This default implementation always returns *TaurusDevState.Ready*

Subclasses of TaurusDevice may reimplement it to return other *taurus.core.TaurusDevState* enumeration values.

Return type *TaurusDevState*

Returns *TaurusDevState.Ready*

TaurusDeviceNameValidator

class TaurusDeviceNameValidator
Bases: taurus.core.taurusvalidator._TaurusBaseValidator

Base class for Device name validators. The namePattern will be composed from URI segments as follows:

<scheme>:[<authority>/]<path>[?<query>][#<fragment>]

Derived classes must provide attributes defining a regexp string for each URI segment (they can be empty strings):

- scheme
- authority
- path
- query
- fragment

Additionally, the namePattern resulting from composing the above segments must contain a named group called "devname" (normally within the path segment).

pattern = '^(?P<scheme>%(scheme)s):((?P<authority>%(authority)s)($|(?=[/#?])))?(?P<path>%(path)s)(\\?(?P<query

TaurusException

class **TaurusException**(*description*, *code=None*)
Bases: exceptions.Exception

TaurusExceptionListener

class **TaurusExceptionListener**
Bases: object

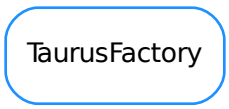
Class for handling ConnectionFailed, DevFailed and TaurusException exceptions.

connectionFailed(*exception*)

devFailed(*exception*)

exceptionReceived(*exception*)

TaurusFactory



class TaurusFactory

Bases: object

The base class for valid Factories in Taurus.

DefaultPollingPeriod = 3000

addAttributeToPolling(*attribute*, *period*, *unsubscribe_evts=False*)

Activates the polling (client side) for the given attribute with the given period (seconds).

Parameters

- **attribute** (*TangoAttribute*) – attribute name.
- **period** (float) – polling period (in seconds)
- **unsubscribe_evts** (bool) – whether or not to unsubscribe from events

caseSensitive = True

changeDefaultPollingPeriod(*period*)

cleanUp()

Reimplement if you need to execute code on program execution exit. Default implementation does nothing.

disablePolling()

Disable the application tango polling

elementTypesMap = None

enablePolling()

Enable the application tango polling

findObject(*absolute_name*)

Must give an absolute name

findObjectClass(*absolute_name*)

Obtain the class object corresponding to the given name.

Note, this generic implementation expects that derived classes provide a an attribute called elementTypesMap consisting in a dictionary whose keys are TaurusElementTypes and whose values are the corresponding specific object classes. e.g., the FooFactory should provide:

```
class FooFactory(TaurusFactory):
    elementTypesMap = {TaurusElementType.Authority: FooAuthority,
                       TaurusElementType.Device: FooDevice,
                       TaurusElementType.Attribute: FooAttribute,
                       }
    (...)
```

Parameters **absolute_name** (str) – the object absolute name string

Return type TaurusModel or None

Returns a TaurusModel class derived type or None if the name is not valid

getAttribute(*name*)

Obtain the model object corresponding to the given attribute name. If the corresponding attribute already exists, the existing instance is returned. Otherwise a new instance is stored and returned.

Parameters **name** (str) – attribute name

Returns a taurus.core.taurusattribute.TaurusAttribute object

Raise

TaurusException if the given name is invalid.

getAttributeNameValidator()

getAuthority(*name=None*)
Obtain the model object corresponding to the given authority name. If the corresponding authority already exists, the existing instance is returned. Otherwise a new instance is stored and returned.

Parameters **name** (str) – authority name

Returns a taurus.core.taurusauthority.TaurusAuthority object

Raise

TaurusException if the given name is invalid.

getAuthorityNameValidator()

getDefaultPollingPeriod()

getDevice(*name*, ***kw*)
Obtain the model object corresponding to the given device name. If the corresponding device already exists, the existing instance is returned. Otherwise a new instance is stored and returned.

Parameters **name** (str) – device name

Returns a taurus.core.taurusdevice.TaurusDevice object

Raise

TaurusException if the given name is invalid.

getDeviceNameValidator()

getObject(*cls*, *name*)

getSerializationMode()
Gives the serialization operation mode.

Return type TaurusSerializationMode

Returns the current serialization mode

getValidTypesForName(*name*, *strict=None*)
Returns a list of all Taurus element types for which *name* is a valid model name (while in many cases a name may only be valid for one element type, this is not necessarily true in general)

In this base implementation, name is checked first for Attribute, then for Device and finally for Authority, and the return value is sorted in that same order.

If a given schema requires a different ordering, reimplement this method

Parameters **name** (str) – taurus model name

Return type list <element>

Returns where element can be one of: *Attribute*, *Device* or *Authority*

isPollingEnabled()
Tells if the Taurus polling is enabled

Return type bool

Returns whether or not the polling is enabled

registerAttributeClass(*attr_name*, *attr_klass*)

registerDeviceClass(*dev_klass_name*, *dev_klass*)

removeAttributeFromPolling(*attribute*)
Deactivate the polling (client side) for the given attribute. If the polling of the attribute was not previously enabled, nothing happens.

Parameters **attribute** (str) – attribute name.

schemes = ()

setSerializationMode(*mode*)
Sets the serialization mode for the system.

Parameters **mode** (TaurusSerializationMode) – the new serialization mode

supportsScheme(*scheme*)
Returns whether the given scheme is supported by this factory

Parameters **scheme** (str) – the name of the schem to be checked

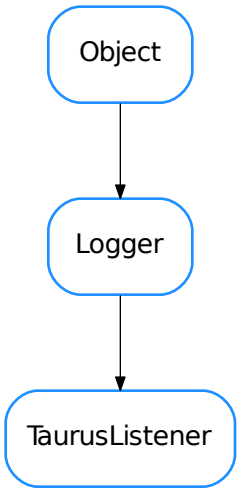
Return type bool

Returns True if the scheme is supported (False otherwise)

unregisterAttributeClass(*attr_name*)

unregisterDeviceClass(*dev_klass_name*)

TaurusListener



class TaurusListener(*name*, *parent=None*)
Bases: taurus.core.util.log.Logger

TaurusListener Interface

attributeList()
Method to return the attributes of the widget

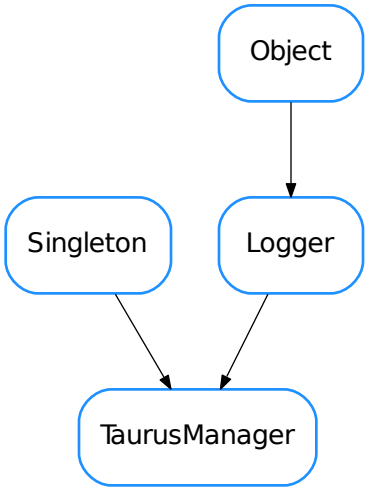
eventReceived(*src*, *type*, *evt_value*)
Method to implement the event notification

TaurusLockInfo

class TaurusLockInfo
Bases: object

LOCK_STATUS_UNKNOWN = 'Lock status unknown'

TaurusManager



class TaurusManager
Bases: taurus.core.util.singleton.Singleton, taurus.core.util.log.Logger

A taurus.core.util.singleton.Singleton class designed to provide Taurus management.

Example:

```
>>> import taurus.core.taurusmanager
>>> manager = taurus.core.taurusmanager.TaurusManager()
>>> print manager == taurus.core.taurusmanager.TaurusManager()
True
```

DefaultSerializationMode = 1

PLUGIN_KEY = '__taurus_plugin__'

addJob(*job*, *callback=None*, **args*, ***kw*)
Add a new job (callable) to the queue. The new job will be processed by a separate thread

Parameters

- **job** (callable) – a callable object
- **callback** (callable) – called after the job has been processed
- **args** (list) – list of arguments passed to the job
- **kw** (dict) – keyword arguments passed to the job

applyPendingOperations(*ops*)
Executes the given operations

Parameters **ops** (*sequence<taurus.core.taurusoperation.TaurusOperation>*) – the sequence of operations

buildPlugins()
Returns the current valid plugins

Return type dic

Returns plugins

changeDefaultPollingPeriod(*period*)

cleanUp()
Cleanup

default_scheme = 'tango'

findObject(*absolute_name*)
Finds the object with the given name

Parameters **absolute_name** (str) – the object name

Return type TaurusModel or None

Returns the taurus model object or None if no suitable name found

findObjectClass(*absolute_name*)
Finds the object class for the given object name

Parameters **absolute_name** (str) – the object name

Return type TaurusModel or None

Returns the taurus model class object or None if no suitable name found

getAttribute(*name*)
Returns a attribute object for the given name

Parameters **name** (str) – attribute name

Return type TaurusAttribute

Returns the attribute for the given name

getAuthority(*name*)
Returns a database object for the given name

Parameters **name** (str) – database name

Return type TaurusAuthority

Returns the authority for the given name

getConfiguration(**args*, ***kwargs*)
Returns a configuration object for the given name

type name str

param name configuration name

rtype TaurusConfiguration

return the configuration for the given name

Deprecated since version 4.0: Use getAttribute instead

getDatabase(*name*)
Deprecated. Use getAuthority instead

getDefaultFactory()
Gives the default factory.

Return type TaurusFactory

Returns the default taurus factory

getDevice(*name*)
Returns a device object for the given name

Parameters **name** (str) – device name

Return type TaurusDevice

Returns the device for the given name

getFactory(*scheme=None*)
Gives the factory class object supporting the given scheme

Parameters **scheme** (str or None) – the scheme. If None the default scheme is used

Return type TaurusFactory or None

Returns the factory class object for the given scheme or None if a proper factory is not found

getObject(*cls*, *name*)
Gives the object for the given class with the given name

Parameters

- **cls** (TaurusModel) – object class
- **name** (str) – the object name

Return type TaurusModel or None

Returns a taurus model object

getOperationMode()
Deprecated. Gives the current operation mode.

Return type OperationMode

Returns the current operation mode

getPlugins()
Gives the information about the existing plugins

Return type dict <str, TaurusFactory>

Returns the list of plugins

getScheme(*name*)
Returns the scheme name for a given model name

Parameters **name** (str) – model name

Return type str

Returns scheme name

getSerializationMode()
Gives the serialization operation mode.

Return type TaurusSerializationMode

Returns the current serialization mode

init(**args*, ***kwargs*)
Singleton instance initialization. For internal usage only. Do **NOT** call this method directly

reInit()
Reinitialization

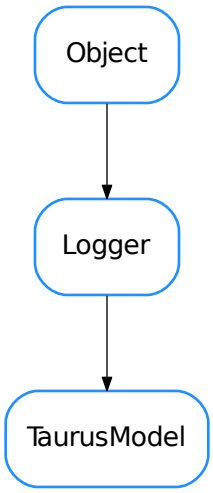
setOperationMode(*mode*)
Deprecated. Sets the operation mode for the system.

Parameters **mode** (OperationMode) – the new operation mode

setSerializationMode(*mode*)
Sets the serialization mode for the system.

Parameters **mode** (TaurusSerializationMode) – the new serialization mode

TaurusModel



class TaurusModel (*full_name*, *parent*, *serializationMode=None*)

Bases: `taurus.core.util.log.Logger`

RegularEvent = (0, 1, 2)

addListener (*listener*)

classmethod buildModelName (*parent_model*, *relative_name*)

cleanUp ()

deleteListener (*listener*)

classmethod factory ()

fireEvent (*event_type*, *event_value*, *listeners=None*)

sends an event to all listeners or a specific one

forceListening ()

fullname

getChildObj (*child_name*)

getDisplayDescrObj (*cache=True*)

A brief description of the model. Can be used as tooltip, for example

getDisplayName (*cache=True*, *complete=True*)

getFragmentObj (*fragmentName=None*)

Returns a fragment object of the model. A fragment of a model is a python attribute of the model object.

Fragment names including dots will be used to recursively get fragments of fragments.

For a simple fragmentName (no dots), this is roughly equivalent to getattr(self, fragmentName)

If the model does not have the fragment, `AttributeError` is raised

Parameters **fragmentName** (str or None) – the returned value will correspond to the given fragmentName. If None is passed the defaultFragmentName will be used instead.

Return type obj

Returns the member of the modelObj referred by the fragment.

getFullName()

classmethod getNameValidator()

getNormalName()

getParentObj()

getSerializationMode()
Gives the serialization operation mode.

Return type TaurusSerializationMode

Returns the current serialization mode

getSimpleName()

classmethod getTaurusElementType()

hasListeners()
returns True if anybody is listening to events from this attribute

classmethod isValid(**args*, ***kwargs*)

isWritable()

name

parentObj

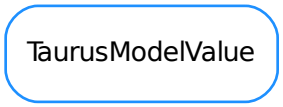
removeListener(*listener*)

setSerializationMode(*mode*)
Sets the serialization mode for the system.

Parameters **mode** (TaurusSerializationMode) – the new serialization mode

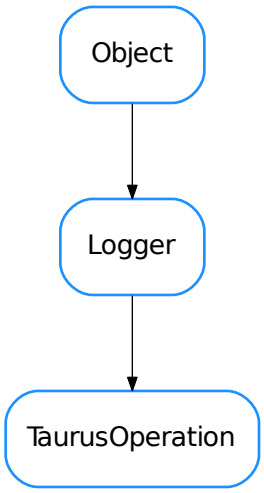
unforceListening()

TaurusModelValue



class TaurusModelValue
Bases: object

TaurusOperation



class **TaurusOperation**(*name='TaurusOperation'*, *parent=None*, *callbacks=None*)

Bases: taurus.core.util.log.Logger

execute()

getCallbacks()

getDangerMessage()

getDevice()

isDangerous()

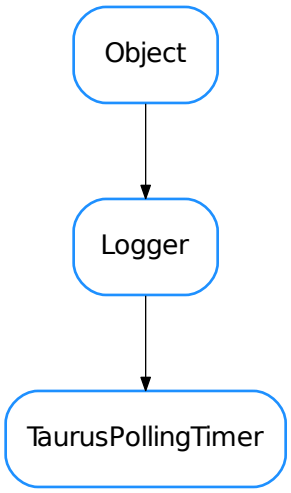
resetDangerMessage()

setCallbacks(*callbacks*)

setDangerMessage(*dangerMessage=None*)

if dangerMessage is None, the operation is considered safe

TaurusPollingTimer



class **TaurusPollingTimer**(*period*, *parent=None*)

Bases: taurus.core.util.log.Logger

Polling timer manages a list of attributes that have to be polled in the same period

addAttribute(*attribute*, *auto_start=True*)

Registers the attribute in this polling.

Parameters

- **attribute** (TaurusAttribute) – the attribute to be added
- **auto_start** (bool) – if True (default) it tells the polling timer that it should startup as soon as there is at least one attribute registered.

containsAttribute(*attribute*)

Determines if the polling timer already contains this attribute

Parameters **attribute** (TaurusAttribute) – the attribute

Return type bool

Returns True if the attribute is registered for polling or False otherwise

getAttributeCount()

Returns the number of attributes registered for polling

Return type int

Returns the number of attributes registered for polling

removeAttribute(*attribute*)

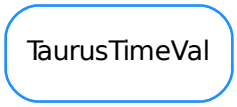
Unregisters the attribute from this polling. If the number of registered attributes decreses to 0 the polling is stopped automatically in order to save resources.

Parameters **attribute** (TaurusAttribute) – the attribute to be added

start()
Starts the polling timer

stop()
Stop the polling timer

TaurusTimeVal



class **TaurusTimeVal**
Bases: object

static **fromdatetime**(*v*)

static **fromtimestamp**(*v*)

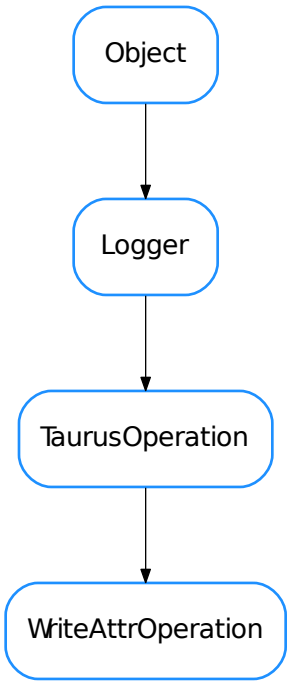
isoformat()

static **now**()

todatetime()

totime()

WriteAttrOperation



class **WriteAttrOperation**(*attr*, *value*, *callbacks=None*)

Bases: taurus.core.taurusoperation.TaurusOperation

execute()

- *AttrQuality*
- *DoubleRegistration*
- *TaurusAttrValue*
- *TaurusAttribute*
- *TaurusAttributeNameValidator*
- *TaurusAuthority*
- *TaurusAuthorityNameValidator*
- *TaurusConfigValue*
- *TaurusConfiguration*
- *TaurusConfigurationProxy*
- *TaurusDevState*
- *TaurusDevice*
- *TaurusDeviceNameValidator*
- *TaurusException*
- *TaurusExceptionListener*
- *TaurusFactory*
- *TaurusListener*
- *TaurusLockInfo*
- *TaurusManager*

- *TaurusModel*
- *TaurusModelValue*
- *TaurusOperation*
- *TaurusPollingTimer*
- *TaurusTimeVal*
- *WriteAttrOperation*

taurus.qt

The taurus.qt submodule. It contains qt-specific part of taurus

Modules

taurus.qt.qtcore

The taurus.qt.qtcore submodule. It contains non-gui components of taurus.qt

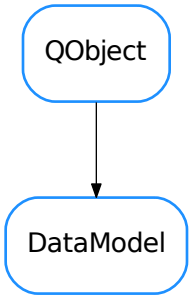
Modules

taurus.qt.qtcore.communication

The taurus.qt.qtcore.communication submodule. It contains non-gui components of taurus.qt

Classes

DataModel



class DataModel (*parent*, *dataUID*, *defaultData=None*)

Bases: `PyQt4.QtCore.QObject`

An object containing one piece of data which is intended to be shared. The data will be identified by its UID (a unique identifier known to objects that intend to access the data)

In general, you are not supposed to instantiate objects of this class directly. Instead, you should interact via the *SharedDataManager*, which uses `SharedDataManager.__getDataModel()` to ensure that the DataModels are singletons.

connectReader(*slot*, *readOnConnect=True*)
Registers the given slot method to receive notifications whenever the data is changed.

Parameters

- **slot** (callable) – a method that will be called when the data changes. This slot will be the receiver of a signal which has the data as its first argument.
- **readOnConnect** (bool) – if True (default) the slot will be called immediately with the current value of the data if the data has been already initialized

See also:

connectWriter(), *getData()*

connectWriter(*writer*, *signalname*)
Registers the given writer object as a writer of the data. The writer is then expected to emit a *QtCore.SIGNAL(signalname)* with the new data as the first parameter.

Parameters

- **writer** (QObject) – object that will change the data
- **signalname** (str) – the signal name that will notify changes of the data

See also:

connectReader(), *setData()*

dataChanged

dataUID()
returns the data unique identifier

Return type str

Returns

disconnectReader(*slot*)
unregister a reader

Parameters **slot** (callable) – the slot to which this was connected

See also:

SharedDataManager.disconnectReader(), *getData()*

disconnectWriter(*writer*, *signalname*)
unregister a writer from this data model

Parameters

- **writer** (QObject) – object to unregister
- **signalname** (str) – the signal that was registered

See also:

SharedDataManager.disconnectWriter()

getData()
Returns the data object.

Return type object

Returns the data object

info()

isDataSet()
Whether the data has been set at least once or if it is uninitialized

Return type bool

Returns True if the data has been set. False it is uninitialized

readerCount()
returns the number of currently registered readers of this model

Return type int

Returns

setData(*data*)
sets the data object and emits a "dataChanged" signal with the data as the parameter

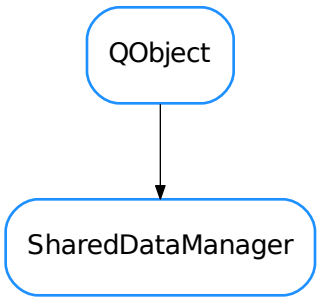
Parameters **data** (object) – the new value for the Model's data

writerCount()
returns the number of currently registered writers of this model

Return type int

Returns

SharedDataManager



class SharedDataManager(*parent*)
Bases: PyQt4.QtCore.QObject

A Factory of *DataModel* objects. The __getDataModel() method ensures that the created DataModels are singletons. DataModels are not kept alive unless there at least some Reader or Writer registered to it (or another object referencing them)

activeDataUIDs()
Returns a list of currently shared data. Note that this list only reflects the situation at the moment of calling this method: a given DataModel may die at any moment if there are no references to it.

Return type list <str>

Returns UIDs of currently shared data.

connectReader(*dataUID*, *slot*, *readOnConnect=True*)
Registers the given slot method to receive notifications whenever the data identified by dataUID is changed.

Note that it returns the *DataModel.getData()* method for the given data UID, which can be used for reading the data at any moment.

Parameters

- **dataUID** (str) – the unique identifier of the data
- **slot** (callable) – a method that will be called when the data changes this slot will be the receiver of a signal which has the data as its first argument.
- **readOnConnect** (bool) – if True (default) the slot will be called immediately with the current value of the data if the data has been already initialized

Return type callable

Returns a callable that can be used for reading the data

See also:

connectWriter(), __getDataModel()

connectWriter(*dataUID*, *writer*, *signalname*)
Registers the given writer object as a changer of the shared data identified by dataUID. The writer is then expected to emit a *QtCore.SIGNAL(signalname)* with the new data as the first parameter

Note that it returns the *DataModel.setData()* method for the given data UID, which can be used for changing the data at any moment.

Parameters

- **dataUID** (str) – the unique identifier of the data
- **writer** (QObject) – object that will change the data
- **signalname** (str) – the signal name that will notify changes of the data

Return type callable

Returns a callable that can be used for setting the data. When using it, one parameter has to be passed containing the new data

See also:

connectWriter(), __getDataModel()

debugReader(*data*)
A slot which you can connect as a reader for debugging. It will print info to the stdout

disconnectReader(*dataUID*, *slot*)
Unregister the given method as data receiver

Parameters

- **dataUID** (str) – the unique identifier of the data
- **slot** (str) – the slot that was registered

See also:

DataModel.disconnectReader()

disconnectWriter(*dataUID*, *writer*, *signalname*)
Unregister the given object as writer of the shared data

Parameters

- **dataUID** (str) – the unique identifier of the data
- **writer** (QObject) – object to unregister
- **signalname** (str) – the signal that was registered

See also:

DataModel.disconnectWriter()

getDataModelProxy(*dataUID*, *callback=None*)
Returns a weakref.proxy to a *DataModel* object for the given data UID or None if the UID is not registered.

Note: The underlying *DataModel* object may cease to exist if all its readers and writers are unregistered.

Parameters

- **dataUID** (str) – the unique identifier of the data
- **callback** (callable) – same as in weakref.ref callback parameter

Return type proxy or None

Returns

See also:

connectReader(), *connectWriter()*, *DataModel*

info()

- *DataModel*
- *SharedDataManager*

Functions

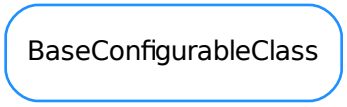
get_signal(*obj*, *signalname*)
Return signal from object and signal name.

taurus.qt.qtcore.configuration

This module provides the set of base classes designed to provide configuration features to the classes that inherit from them

Classes

BaseConfigurableClass



class BaseConfigurableClass

A base class defining the API for configurable objects.

Note: One implicit requisite is that a configurable object must also provide a *meth:'objectName* method which returns the object name. This is typically fulfilled by inheriting from QObject.

Using objects that inherit from *BaseConfigurableClass* automates saving and restoring of application settings and also enables the use of perspectives in Taurus GUIs.

The basic idea is that each object/widget in your application is responsible for providing a dictionary containing information on its properties (see *createConfig()*). The same object/widget is also responsible for restoring such properties when provided with a configuration dictionary (see *applyConfig()*).

For a certain property to be saved/restored it is usually enough to *register* it using *registerConfigProperty()*. When the objects are structured in a hierarchical way (e.g. as the widgets in a Qt application), the parent widget can (should) delegate the save/restore of its children to the children themselves. This delegation is done by registering the children using *registerConfigDelegate()*.

Consider the following example: I am creating a groupbox container which contains a `TaurusForm` and I want to save/restore the state of the checkbox and the properties of the form:

```
#The class looks like this:
class MyBox(Qt.QGroupBox, BaseConfigurableClass):
    def __init__(self):
        ...
        self.form = TaurusForm()
        ...
        self.registerConfigProperty(self.isChecked, self.setChecked, 'checked')
        self.registerConfigDelegate(self.form)   #the TaurusForm already handles␣
↪its own configuration!
        ...

#and we can retrieve the configuration doing:
b1 = MyBox()
b1.setChecked(True)  #checked is a registered property of MyBox class
b1.form.setModifiableByUser(True)  #modifiableByUser is a registered property of␣
↪a TaurusForm
cfg = b1.createConfig()  #we get the configuration as a dictionary
...
b2 = MyBox()
b2.applyConfig(cfg)  #now b2 has the same configuration as b1 when cfg was created
```

createConfig() and *applyConfig()* methods use a dictionary for passing the configuration,

but *BaseConfigurableClass* also provides some other convenience methods for working with files (*saveConfigFile()* and *loadConfigFile()*) or as QByteArrays (*createQConfig()* and *applyQConfig()*)

Finally, we recommend to use `TaurusMainWindow` for all Taurus GUIs since it automates all the steps for *saving properties when closing* and *restoring the settings on startup*. It also provides a mechanism for implementing "perspectives" in your application.

applyConfig(*configdict*, *depth=None*)

applies the settings stored in a configdict to the current object.

In most usual situations, using *registerConfigProperty()* and *registerConfigDelegate()*, should be enough to cover all needs using this method, although it can be reimplemented in children classes to support very specific configurations.

Parameters

- **configdict** (`dict`) –
- **depth** (`int`) – If depth = 0, applyConfig will only be called for this object, and not for any other object registered via *registerConfigurableItem()*. If depth > 0, applyConfig will be called recursively as many times as the depth value. If depth < 0 (default, see note), no limit is imposed to recursion (i.e., it will recurse for as deep as there are registered items).

Note: the default recursion depth can be tweaked in derived classes by changing the class property *defaultConfigRecursionDepth*

See also:

createConfig()

applyQConfig(*qstate*)

restores the configuration from a qstate generated by `getQState()`.

Parameters **qstate** (`QByteArray`) –

See also:

createQConfig()

checkConfigVersion(*configdict*, *showDialog=False*, *supportedVersions=None*)

Check if the version of configdict is supported. By default, the BaseConfigurableClass objects have ["__UNVERSIONED__"] as their list of supported versions, so unversioned config dicts will be accepted.

Parameters

- **configdict** (`dict`) – configuration dictionary to check
- **showDialog** (`bool`) – whether to show a QtWarning dialog if check failed (false by default)
- **supportedVersions** (`sequence <str>` or `None`) – supported version numbers, if None given, the versions supported by this widget will be used (i.e., those defined in self._supportedConfigVersions)

Return type `bool`

Returns returns True if the configdict is of the right version

createConfig(*allowUnpickable=False*)

Returns a dictionary containing configuration information about the current state of the object.

In most usual situations, using *registerConfigProperty()* and *registerConfigDelegate()*, should be enough to cover all needs using this method, although it can be reimplemented in children classes to support very specific configurations.

By default, meth:*createQConfig* and meth:*saveConfigFile* call to this method for obtaining the data.

Hint: The following code allows you to serialize the configuration dictionary as a string (which you can store as a QSetting, or as a Tango Attribute, provided that allowUnpickable==False):

```
import pickle
s = pickle.dumps(widget.createConfig())  #s is a string that can be stored
```

Parameters **alllowUnpickable** (bool) – if False the returned dict is guaranteed to be a pickable object. This is the default and preferred option because it allows the serialization as a string that can be directly stored in a QSetting. If True, this limitation is not enforced, which allows to use more complex objects as values (but limits its persistence).

Return type dict <str, object>

Returns configurations (which can be loaded with *applyConfig()*).

createQConfig()
returns the current configuration status encoded as a QByteArray. This state can therefore be easily stored using QSettings

Return type QByteArray

Returns (in the current implementation this is just a pickled configdict encoded as a QByteArray

See also:

restoreQConfig()

defaultConfigRecursionDepth = -1

getConfigurableItemNames()
returns an ordered list of the names of currently registered configuration items (delegates and properties)

Return type list <unicode>

Returns

static **isTaurusConfig**(*x*)
Checks if the given argument has the structure of a configdict

Parameters **x** (object) – object to test

Return type bool

Returns True if it is a configdict, False otherwise.

loadConfigFile(*ifile=None*)
Reads a file stored by saveConfig() and applies the settings

Parameters **ifile** (file or string) – file or filename from where to read the configuration

Return type str

Returns file name used

registerConfigDelegate(*delegate*, *name=None*)
Registers the given object as a delegate for configuration. Delegates are typically other objects inheriting from BaseConfigurableClass (or at least they must provide the following methods:

•*createConfig* (as provided by, e.g., BaseConfigurableClass)

•*applyConfig* (as provided by, e.g., BaseConfigurableClass)

•*objectName* (as provided by, e.g., QObject)

Parameters

- **delegate** (*BaseConfigurableClass*) – The delegate object to be registered.
- **name** (str) – The name to be used as a key for this item in the configuration dictionary. If None given, the object name is used by default.

Note: the registration order will be used when restoring configurations

---

See also:

unregisterConfigurableItem(), *registerConfigProperty()*, *createConfig()*

registerConfigProperty(*fget*, *fset*, *name*)
Registers a certain property to be included in the config dictionary.

In this context a "property" is some named value that can be obtained via a getter method and can be set via a setter method.

Parameters

- **fget** (method or str) – method (or name of a method) that gets no arguments and returns the value of a property.
- **fset** (method or str) – method (or name of a method) that gets as an argument the value of a property, and sets it
- **name** (str) – The name to be used as a key for this property in the configuration dictionary

---

Note: the registration order will be used when restoring configurations

---

See also:

unregisterConfigurableItem(), *registerConfigDelegate()*, *createConfig()*

registerConfigurableItem(*item*, *name=None*)

resetConfigurableItems()
clears the record of configurable items depending of this object

See also:

registerConfigurableItem()

saveConfigFile(*ofile=None*)
Stores the current configuration on a file

Parameters **ofile** (file or string) – file or filename to store the configuration

Return type str

Returns file name used

unregisterConfigurableItem(*item*, *raiseOnError=True*)
unregisters the given item (either a delegate or a property) from the configurable items record. It raises an exception if the item is not registered

Parameters

- **item** (object or str) – The object that should be unregistered. Alternatively, the name under which the object was registered can be passed as a python string.
- **raiseOnError** (bool) – If True (default), it raises a KeyError exception if item was not registered. If False, it just logs a debug message

See also:

registerConfigProperty(), *registerConfigDelegate()*

configurableProperty

class configurableProperty(*name*, *fget*, *fset*, *obj=None*)

A dummy class used to handle properties with the configuration API

> **Warning:** this class is intended for internal use by the configuration package. Do not instantiate it directly in your code. Use *BaseConfigurableClass.registerConfigProperty()* instead.

applyConfig(*value*, *depth=-1*)

calls the fset function for this property with the given value. The depth parameter is ignored

createConfig(*allowUnpickable=False*)

returns value returned by the fget function of this property. the allowUnpickable parameter is ignored

objectName()

returns the name of this property

- *BaseConfigurableClass*
- *configurableProperty*

taurus.qt.qtcore.mimetypes

contains MIME type definitions used by taurus widgets

taurus.qt.qtcore.model

This package provides taurus Qt models

Pure PyQt view based widgets can be used to display the contents of the several model classes provided here.

Displaying the device list in a `PyQt4.QtGui.QTreeView`:

```
view = Qt.QTreeView()
db = taurus.Database()
model = taurus.qt.qtcore.TaurusDbDeviceModel()
model.setDataSource(db)
view.setModel(model)
```

Same example but in a `PyQt4.QtGui.QTableView`:

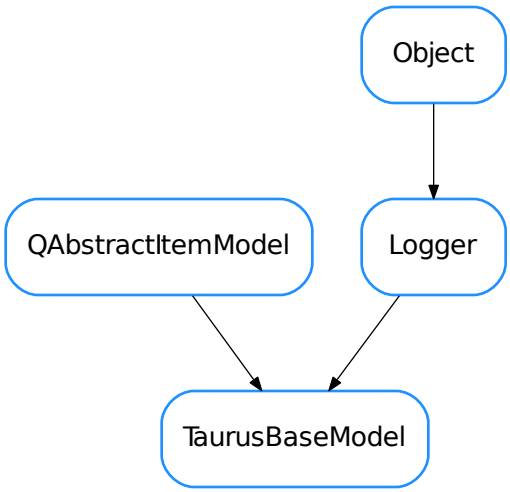
```
view = Qt.QTableView()
db = taurus.Database()
model = taurus.qt.qtcore.TaurusDbPlainDeviceModel()
model.setDataSource(db)
view.setModel(model)
```

And now inside a `PyQt4.QtGui.QComboBox`:

```
view = Qt.QComboBox()
db = taurus.Database()
model = taurus.qt.qtcore.TaurusDbPlainDeviceModel()
model.setDataSource(db)
view.setModel(model)
```

Classes

TaurusBaseModel



class TaurusBaseModel (*parent=None*, *data=None*)

Bases: `PyQt4.QtCore.QAbstractItemModel`, `taurus.core.util.log.Logger`

The base class for all Taurus Qt models.

ColumnNames = ()

ColumnRoles = ((),)

DftFont

columnCount(*parent=<PyQt4.QtCore.QModelIndex object>*)

columnIcon(*column*)

columnSize(*column*)

columnToolTip(*column*)

createNewRootItem()

data(*index*, *role=0*)

dataSource()

flags(*index*)

hasChildren(*parent=<PyQt4.QtCore.QModelIndex object>*)

headerData(*section*, *orientation*, *role=0*)

index(*row*, *column*, *parent=<PyQt4.QtCore.QModelIndex object>*)

parent(*index*)

pyData(*index*, *role=0*)

refresh(*refresh_source=False*)

role(*column*, *depth=0*)

roleIcon(*role*)

roleSize(*role*)

roleToolTip(*role*)

rowCount(*parent=<PyQt4.QtCore.QModelIndex object>*)

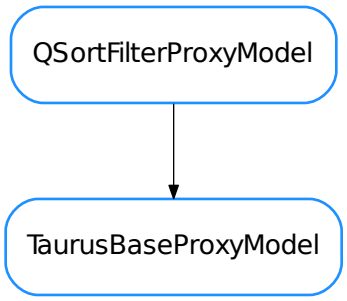
selectables()

setDataSource(*data_src*)

setSelectables(*seq_elem_types*)

setupModelData(*data*)

TaurusBaseProxyModel

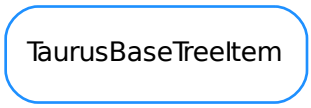


class **TaurusBaseProxyModel** (*parent=None*)

Bases: PyQt4.QtGui.QSortFilterProxyModel

A taurus base Qt filter & sort model

TaurusBaseTreeItem



class **TaurusBaseTreeItem** (*model*, *data*, *parent=None*)

Bases: object

A generic node

DisplayFunc

alias of str

appendChild (*child*)

Adds a new child node

Parameters **child** (TaurusTreeBaseItem) – child to be added

child (*row*)

Returns the child in the given row

Return type TaurusTreeBaseItem

Returns the child node for the given row

childCount()
Returns the number of childs for this node

Return type int

Returns number of childs for this node

data(*index*)
Returns the data of this node for the given index

Return type object

Returns the data for the given index

depth()
Depth of the node in the hierarchy

Return type int

Returns the node depth

display()
Returns the display string for this node

Return type str

Returns the node's display string

hasChildren()

icon(*index*)

itemData()
The internal itemData object

Return type object

Returns object holding the data of this item

mimeData(*index*)

parent()
Returns the parent node or None if no parent exists

Return type TaurusTreeBaseItem

Returns the parent node

qdisplay()
Returns the display QString for this node

Return type QString

Returns the node's display string

role()
Returns the prefered role for the item. This implementation returns taurus.core.taurusbasetypes.TaurusElementType.Unknown

This method should be able to return any kind of python object as long as the model that is used is compatible.

Return type TaurusElementType

Returns the role in form of element type

row()
Returns the row for this node

Return type int

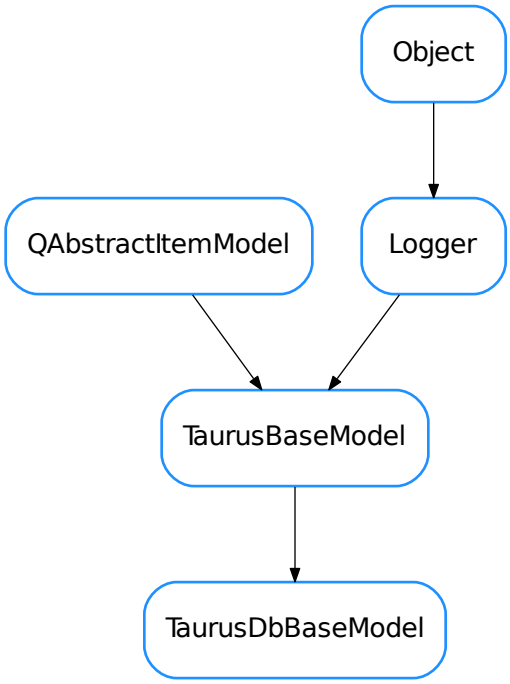
Returns row number for this node

setData(*index*, *data*)
Sets the node data

Parameters **data** (object) – the data to be associated with this node

toolTip(*index*)

TaurusDbBaseModel



class **TaurusDbBaseModel**(*parent=None*, *data=None*)
Bases: taurus.qt.qtcore.model.taurusmodel.TaurusBaseModel

The base class for all Taurus database Qt models. By default, this model represents a plain device perspective of the underlying database.

ColumnNames = ('Device', 'Alias', 'Server', 'Class', 'Alive', 'Host')

ColumnRoles = ((3, 3), 4, 8, 2, 11, 12)

columnIcon(*column*)

columnSize(*column*)

columnToolTip(*column*)

createNewRootItem()

mimeData(*indexes*)

mimeTypes()

pyData(*index*, *role*)

refresh(*refresh_source=False*)

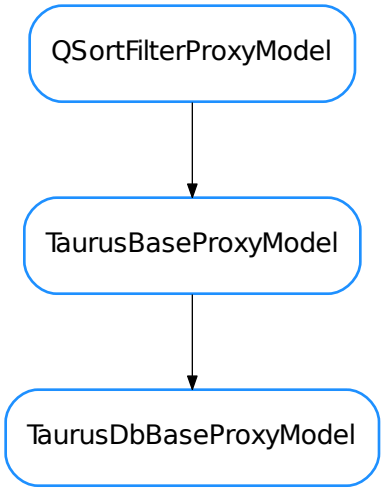
roleIcon(*taurus_role*)

roleSize(*taurus_role*)

roleToolTip(*taurus_role*)

setupModelData(*data*)

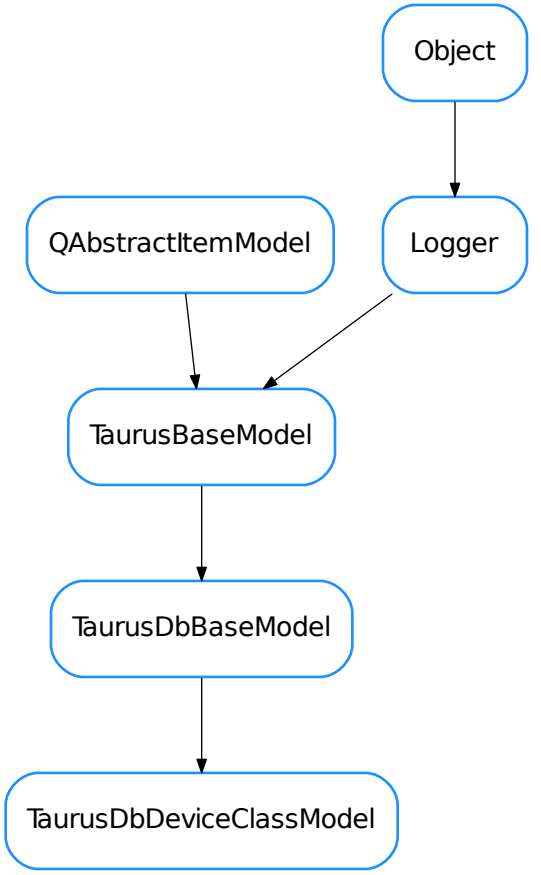
TaurusDbBaseProxyModel



class TaurusDbBaseProxyModel(*parent=None*)

Bases: `taurus.qt.qtcore.model.taurusmodel.TaurusBaseProxyModel`

TaurusDbDeviceClassModel



class **TaurusDbDeviceClassModel** (*parent=None*, *data=None*)

Bases: `taurus.qt.qtcore.model.taurusdatabasemodel.TaurusDbBaseModel`

A Qt model that structures class elements in a tree organized as:

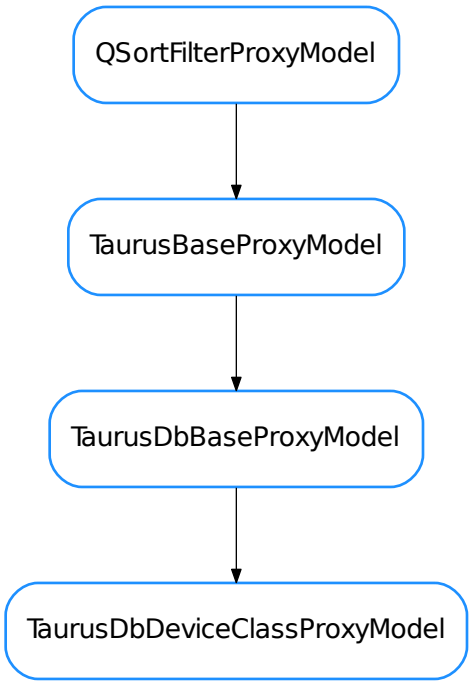
- **<Class>**
  - **<Device>**
    - <Attribute>

ColumnNames = ('Class', 'Alive', 'Host')

ColumnRoles = ((2, 2, 3, 13), 11, 12)

setupModelData (*data*)

TaurusDbDeviceClassProxyModel



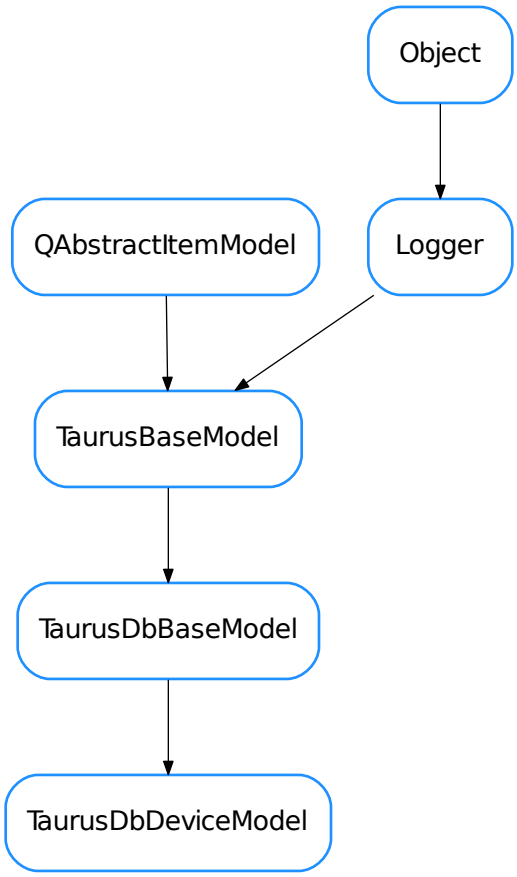
class **TaurusDbDeviceClassProxyModel** (*parent=None*)

Bases: `taurus.qt.qtcore.model.taurusdatabasemodel.TaurusDbBaseProxyModel`

A Qt filter & sort model for the TaurusDbDeviceClassModel

filterAcceptsRow (*sourceRow*, *sourceParent*)

TaurusDbDeviceModel



class **TaurusDbDeviceModel** (*parent=None*, *data=None*)

Bases: `taurus.qt.qtcore.model.taurusdatabasemodel.TaurusDbBaseModel`

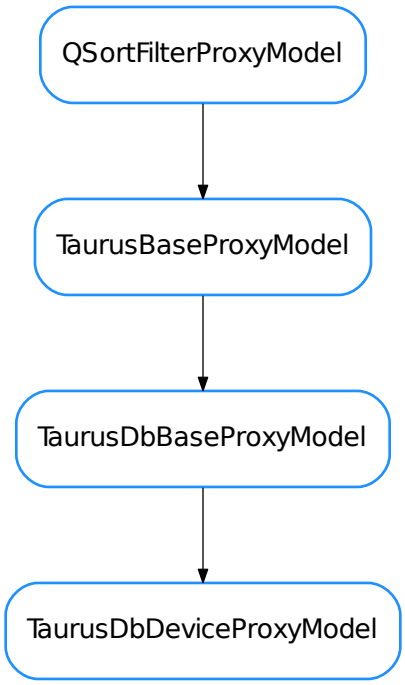
A Qt model that structures device elements is a 3 level tree organized as:

- <domain>
- <family>
- <member>

ColumnRoles = ((3, 5, 6, 7, 13), 4, 8, 2, 11, 12)

setupModelData (*data*)

TaurusDbDeviceProxyModel



class **TaurusDbDeviceProxyModel** (*parent=None*)

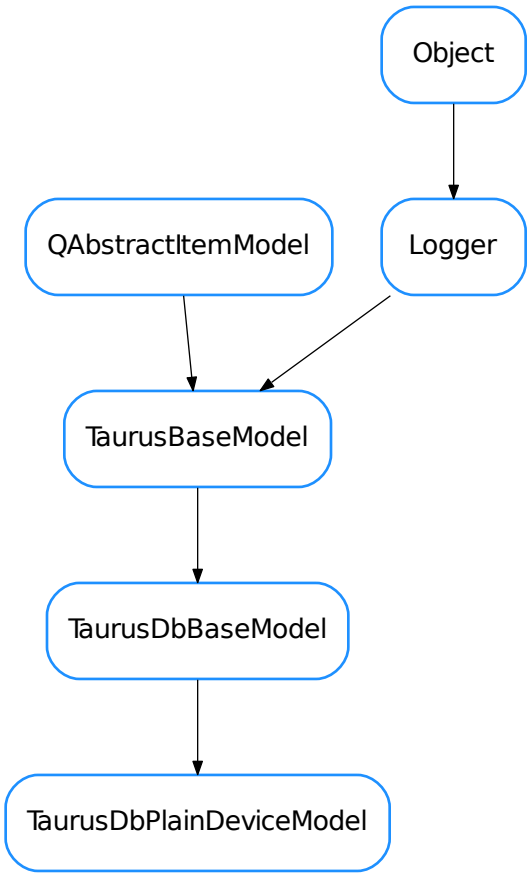
Bases: `taurus.qt.qtcore.model.taurusdatabasemodel.TaurusDbBaseProxyModel`

A Qt filter & sort model for model for the taurus models: - TaurusDbBaseModel - TaurusDbDeviceModel - TaurusDbSimpleDeviceModel - TaurusDbPlainDeviceModel

deviceMatches (*device*, *regexp*)

filterAcceptsRow (*sourceRow*, *sourceParent*)

TaurusDbPlainDeviceModel



class TaurusDbPlainDeviceModel (*parent=None*, *data=None*)

Bases: `taurus.qt.qtcore.model.taurusdatabasemodel.TaurusDbBaseModel`

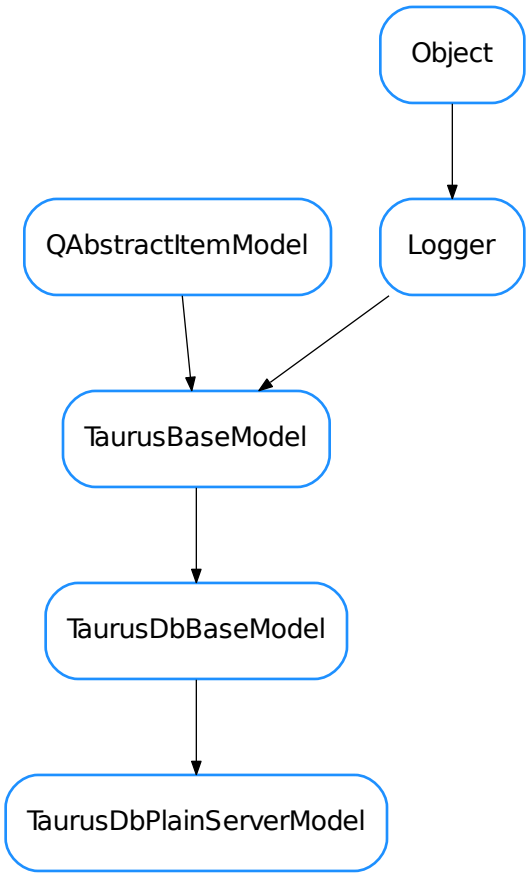
A Qt model that structures device elements in 1 level tree. Device nodes will have attribute child nodes if the device is running.

ColumnNames = ('Device', 'Alias', 'Server', 'Class', 'Alive', 'Host')

ColumnRoles = ((3, 3, 13), 4, 8, 2, 11, 12)

setupModelData (*data*)

TaurusDbPlainServerModel



class **TaurusDbPlainServerModel** (*parent=None*, *data=None*)

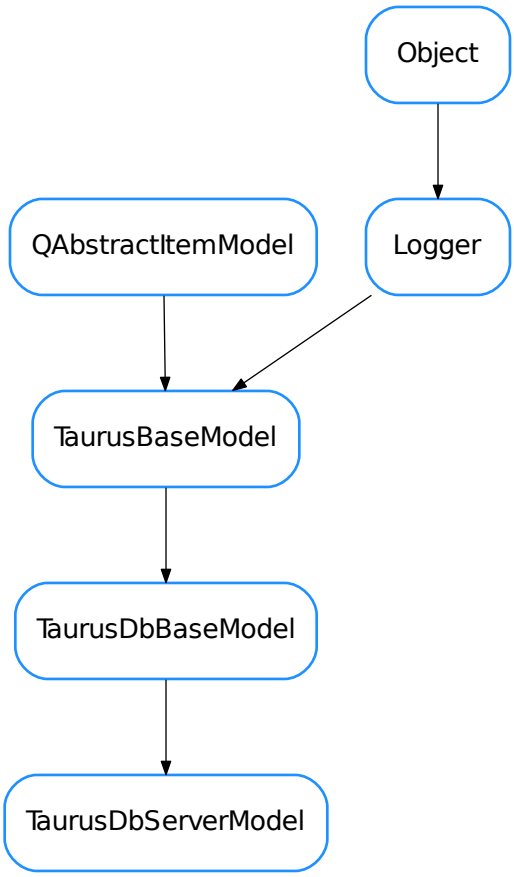
Bases: `taurus.qt.qtcore.model.taurusdatabasemodel.TaurusDbBaseModel`

ColumnNames = ('Server', 'Alive', 'Host')

ColumnRoles = ((8, 10), 11, 12)

setupModelData (*data*)

TaurusDbServerModel



class TaurusDbServerModel (*parent=None*, *data=None*)

Bases: `taurus.qt.qtcore.model.taurusdatabasemodel.TaurusDbBaseModel`

A Qt model that structures server elements in a tree organized as:

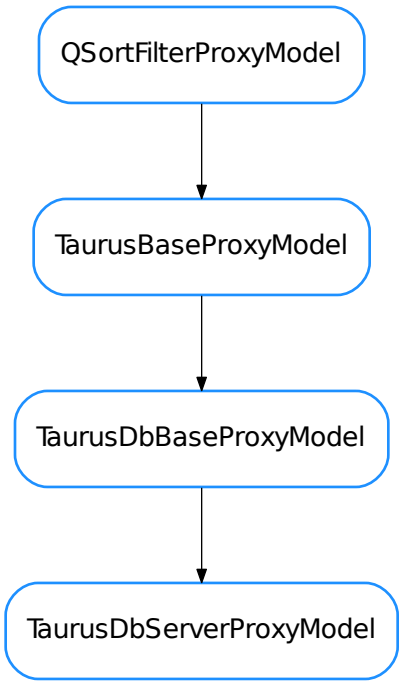
- <Server name>
- <Server instance>
- <Class>
- <Device>
- <Attribute>

ColumnNames = ('Server', 'Alive', 'Host')

ColumnRoles = ((8, 9, 10, 2, 3, 13), 11, 12)

setupModelData (*data*)

TaurusDbServerProxyModel



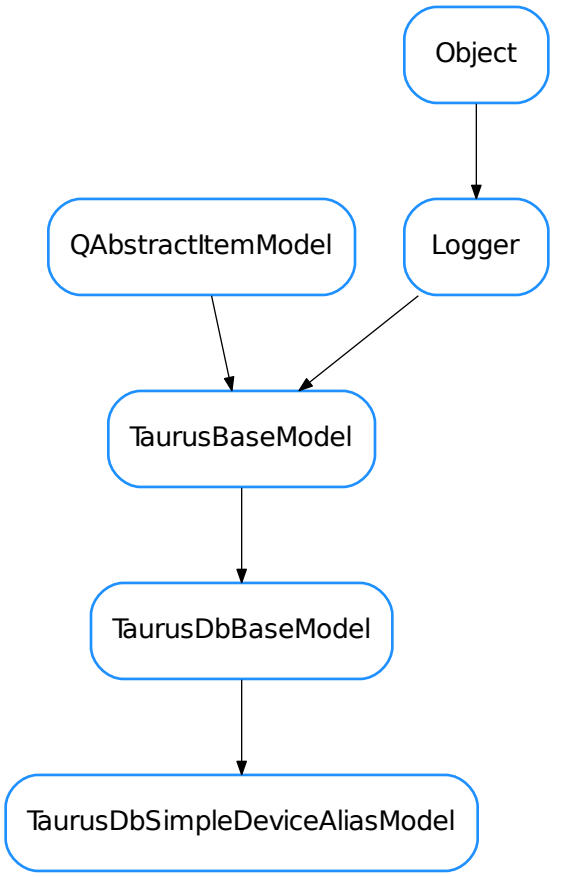
class **TaurusDbServerProxyModel** (*parent=None*)

Bases: `taurus.qt.qtcore.model.taurusdatabasemodel.TaurusDbBaseProxyModel`

A Qt filter & sort model for the TaurusDbServerModel

filterAcceptsRow (*sourceRow*, *sourceParent*)

TaurusDbSimpleDeviceAliasModel



class **TaurusDbSimpleDeviceAliasModel** (*parent=None*, *data=None*)

Bases: `taurus.qt.qtcore.model.taurusdatabasemodel.TaurusDbBaseModel`

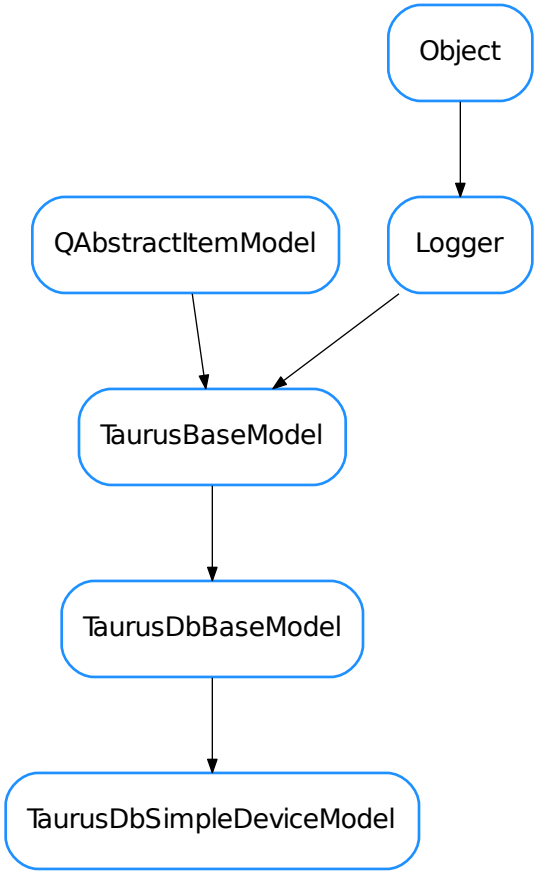
A Qt model that structures device elements in 1 level tree with device alias as node leafs. This model contains only 1 column.

ColumnNames = ('Alias',)

ColumnRoles = ((4, 4),)

setupModelData (*data*)

TaurusDbSimpleDeviceModel



class TaurusDbSimpleDeviceModel (*parent=None*, *data=None*)

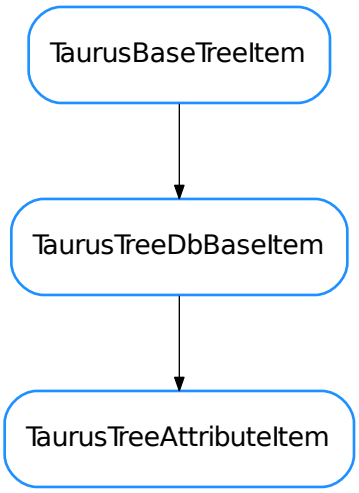
Bases: `taurus.qt.qtcore.model.taurusdatabasemodel.TaurusDbBaseModel`

A Qt model that structures device elements in 1 level tree with device name as node leafs. This model contains only 1 column.

ColumnNames = ('Device',)

ColumnRoles = ((3, 3),)

TaurusTreeAttributeItem



class **TaurusTreeAttributeItem**(*model*, *data*, *parent=None*)

Bases: `taurus.qt.qtcore.model.taurusdatabasemodel.TaurusTreeDbBaseItem`

A node designed to represent an attribute

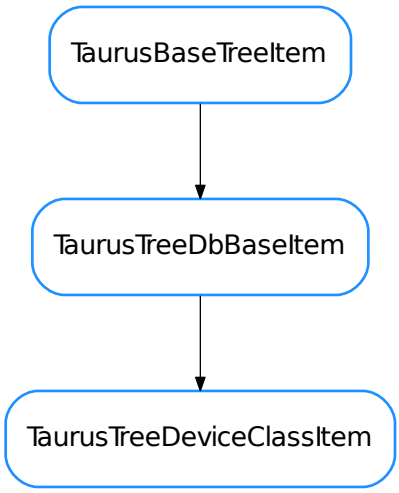
data(*index*)

mimeData(*index*)

role()

toolTip(*index*)

TaurusTreeDeviceClassItem



class **TaurusTreeDeviceClassItem**(*model*, *data*, *parent=None*)

Bases: taurus.qt.qtcore.model.taurusdatabasemodel.TaurusTreeDbBaseItem

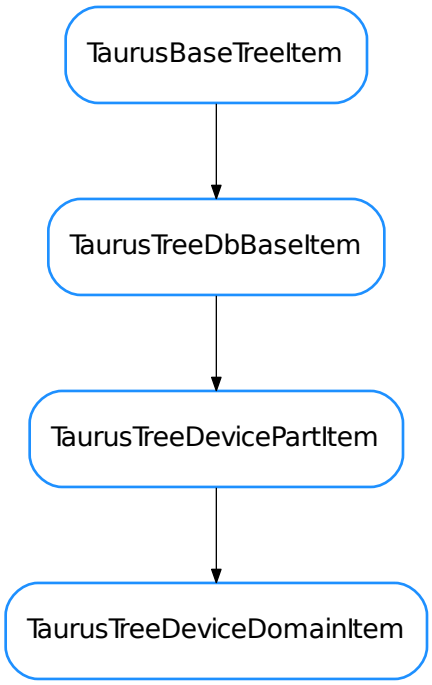
A node designed to represent a device class

data(*index*)

mimeData(*index*)

role()

TaurusTreeDeviceDomainItem



class **TaurusTreeDeviceDomainItem**(*model*, *data*, *parent=None*)

Bases: `taurus.qt.qtcore.model.taurusdatabasemodel.TaurusTreeDevicePartItem`

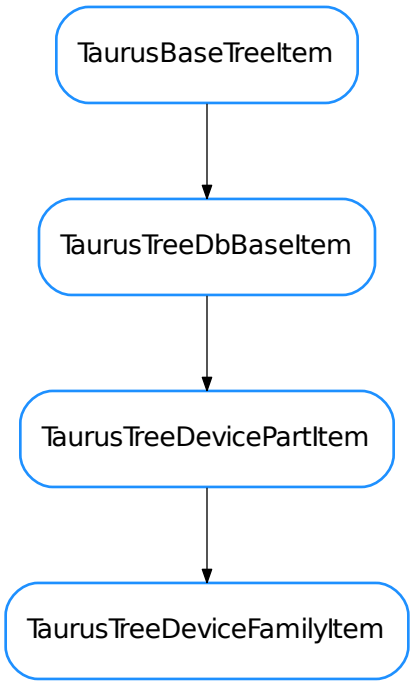
A node designed to represent a the domain part of a device name

DisplayFunc

alias of `str`

role()

TaurusTreeDeviceFamilyItem



class **TaurusTreeDeviceFamilyItem**(*model*, *data*, *parent=None*)

Bases: `taurus.qt.qtcore.model.taurusdatabasemodel.TaurusTreeDevicePartItem`

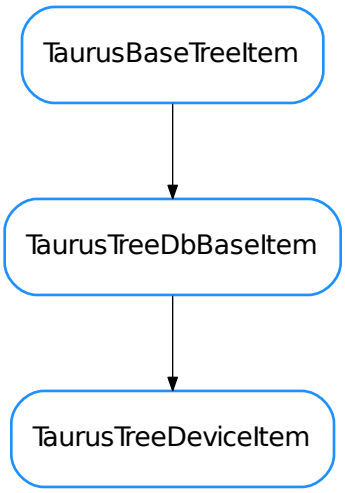
A node designed to represent a the family part of a device name

DisplayFunc

alias of `str`

role()

TaurusTreeDeviceItem



class **TaurusTreeDeviceItem**(*model*, *data*, *parent=None*)

Bases: taurus.qt.qtcore.model.taurusdatabasemodel.TaurusTreeDbBaseItem

A node designed to represent a device

child(*row*)

childCount()

data(*index*)

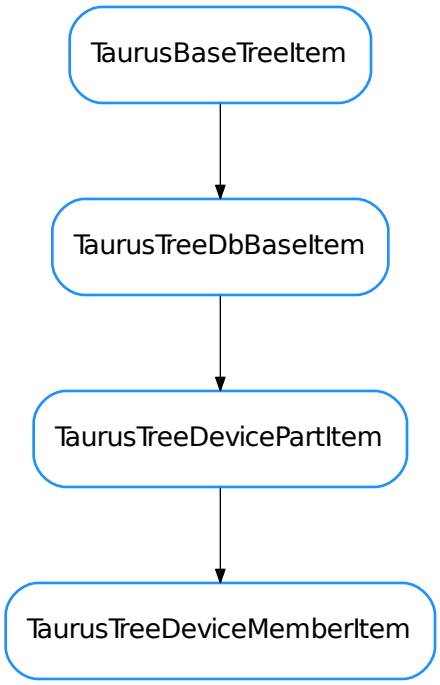
hasChildren()

mimeData(*index*)

role()

updateChilds()

TaurusTreeDeviceMemberItem



class **TaurusTreeDeviceMemberItem**(*model*, *data*, *parent=None*)

Bases: `taurus.qt.qtcore.model.taurusdatabasemodel.TaurusTreeDevicePartItem`

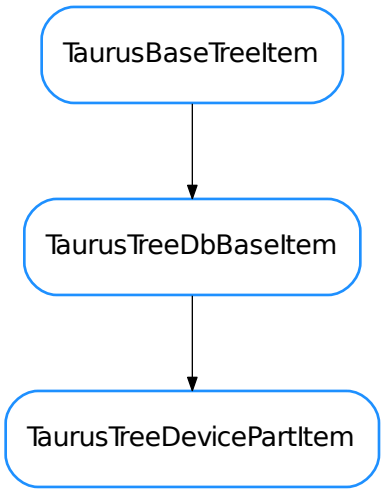
A node designed to represent a the member part of a device name

DisplayFunc

alias of `str`

role()

TaurusTreeDevicePartItem



class **TaurusTreeDevicePartItem**(*model*, *data*, *parent=None*)

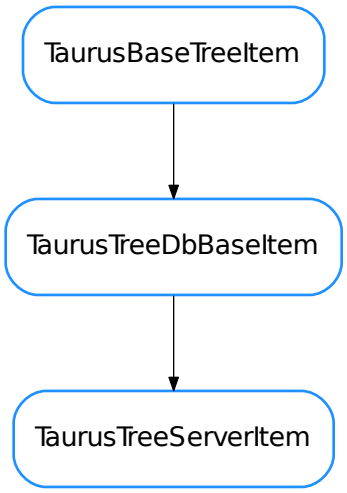
Bases: taurus.qt.qtcore.model.taurusdatabasemodel.TaurusTreeDbBaseItem

A node designed to represent a 'part' (or totality) of a device name

data(*index*)

role()

TaurusTreeServerItem



class **TaurusTreeServerItem**(*model*, *data*, *parent=None*)

Bases: taurus.qt.qtcore.model.taurusdatabasemodel.TaurusTreeDbBaseItem

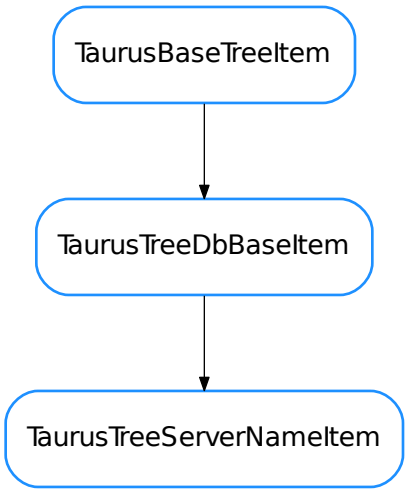
A node designed to represent a server

data(*index*)

mimeData(*index*)

role()

TaurusTreeServerNameItem



class **TaurusTreeServerNameItem**(*model*, *data*, *parent=None*)

Bases: taurus.qt.qtcore.model.taurusdatabasemodel.TaurusTreeDbBaseItem

A node designed to represent the server name part of a server

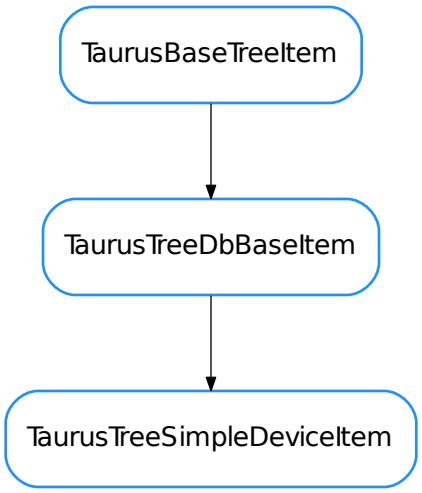
DisplayFunc

alias of str

data(*index*)

role()

TaurusTreeSimpleDeviceItem



class **TaurusTreeSimpleDeviceItem**(*model*, *data*, *parent=None*)

Bases: `taurus.qt.qtcore.model.taurusdatabasemodel.TaurusTreeDbBaseItem`

A node designed to represent a device (without any child nodes)

childCount()

data(*index*)

hasChildren()

mimeData(*index*)

role()

- *TaurusBaseModel*
- *TaurusBaseProxyModel*
- *TaurusBaseTreeItem*
- *TaurusDbBaseModel*
- *TaurusDbBaseProxyModel*
- *TaurusDbDeviceClassModel*
- *TaurusDbDeviceClassProxyModel*
- *TaurusDbDeviceModel*
- *TaurusDbDeviceProxyModel*
- *TaurusDbPlainDeviceModel*
- *TaurusDbPlainServerModel*
- *TaurusDbServerModel*
- *TaurusDbServerProxyModel*
- *TaurusDbSimpleDeviceAliasModel*
- *TaurusDbSimpleDeviceModel*
- *TaurusTreeAttributeItem*
- *TaurusTreeDeviceClassItem*

- *TaurusTreeDeviceDomainItem*
- *TaurusTreeDeviceFamilyItem*
- *TaurusTreeDeviceItem*
- *TaurusTreeDeviceMemberItem*
- *TaurusTreeDevicePartItem*
- *TaurusTreeServerItem*
- *TaurusTreeServerNameItem*
- *TaurusTreeSimpleDeviceItem*

taurus.qt.qtcore.tango

taurus.qt.qtcore.util

This package provides a set of utilities (e.g. logging) to taurus qtcore

Functions

getQtLogger()

initTaurusQtLogger()

taurus.qt.qtdesigner

The taurus.qt.qtdesigner submodule. It contains qt-specific part of taurus

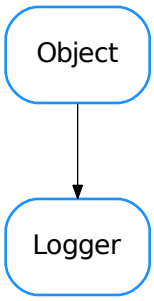
Modules

taurus.qt.qtdesigner.taurusplugin

The taurus.qt.qtdesigner.taurusplugin submodule.

Classes

Logger



class Logger (*name=''*, *parent=None*, *format=None*)
Bases: taurus.core.util.object.Object

The taurus logger class. All taurus pertinent classes should inherit directly or indirectly from this class if they need taurus logging facilities.

Critical = 50
Critical message level (constant)

Debug = 10
Debug message level (constant)

DftLogFormat = <logging.Formatter object>
Default log format (constant)

DftLogLevel = 20
Default log level (constant)

DftLogMessageFormat = '%(threadName)-14s %(levelname)-8s %(asctime)s %(name)s: %(message)s'
Default log message format (constant)

Error = 40
Error message level (constant)

Fatal = 50
Fatal message level (constant)

Info = 20
Info message level (constant)

Trace = 5
Trace message level (constant)

Warning = 30
Warning message level (constant)

addChild (*child*)
Adds a new logging child

Parameters **child** (Logger) – the new child

classmethod addLevelName(*level_no*, *level_name*)
Registers a new log level

Parameters

- **level_no** (int) – the level number
- **level_name** (str) – the corresponding name

addLogHandler(*handler*)
Registers a new handler in this object's logger

Parameters **handler** (Handler) – the new handler to be added

classmethod addRootLogHandler(*h*)
Adds a new handler to the root logger

Parameters **h** (Handler) – the new log handler

changeLogName(*name*)
Change the log name for this object.

Parameters **name** (str) – the new log name

cleanUp()
The cleanUp. Default implementation does nothing Overwrite when necessary

copyLogHandlers(*other*)
Copies the log handlers of other object to this object

Parameters **other** (object) – object which contains 'log_handlers'

critical(*msg*, **args*, ***kw*)
Record a critical message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.critical().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

debug(*msg*, **args*, ***kw*)
Record a debug message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.debug().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

deprecated(*msg=None*, *dep=None*, *alt=None*, *rel=None*, *dbg_msg=None*, *_callerinfo=None*, ***kw*)
Record a deprecated warning message in this object's logger. If message is not passed, a estandard deprecation message is constructued using dep, alt, rel arguments. Also, an extra debug message can be recorded, followed by traceback info.

Parameters

- **msg** (str) – the message to be recorded (if None passed, it will be constructed using dep (and, optionally, alt and rel)
- **dep** (str) – name of deprecated feature (in case msg is None)

- **alt** (str) – name of alternative feature (in case msg is None)
- **rel** (str) – name of release from which the feature was deprecated (in case msg is None)
- **dbg_msg** (str) – msg for debug (or None to log only the warning)
- **_callerinfo** – for internal use only. Do not use this argument.
- **kw** – any additional keyword arguments, are passed to logging.Logger.warning()

classmethod disableLogOutput()
Disables the logging.StreamHandler which dumps log records, by default, to the stderr.

classmethod enableLogOutput()
Enables the logging.StreamHandler which dumps log records, by default, to the stderr.

error(*msg*, **args*, ***kw*)
Record an error message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.error().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

exception(*msg*, **args*)
Log a message with severity 'ERROR' on the root logger, with exception information.. Accepted *args* are the same as logging.Logger.exception().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments

fatal(*msg*, **args*, ***kw*)
Record a fatal message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.fatal().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

flushOutput()
Flushes the log output

getChildren()
Returns the log children for this object

Return type Logger

Returns the list of log children

classmethod getLogFormat()
Retuns the current log message format (the root log format)

Return type str

Returns the log message format

getLogFullName()
Gets the full log name for this object

Return type str

Returns the full log name

classmethod getLogLevel()
Retuns the current log level (the root log level)

Return type int

Returns a number representing the log level

getLogName()
Gets the log name for this object

Return type str

Returns the log name

getLogObj()
Returns the log object for this object

Return type Logger

Returns the log object

classmethod getLogger(*name=None*)

getParent()
Returns the log parent for this object or None if no parent exists

Return type Logger or None

Returns the log parent for this object

classmethod getRootLog()
Retuns the root logger

Return type Logger

Returns the root logger

info(*msg*, **args*, ***kw*)
Record an info message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.info().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

classmethod initRoot()
Class method to initialize the root logger. Do **NOT** call this method directly in your code

log(*level*, *msg*, **args*, ***kw*)
Record a log message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.log().

Parameters

- **level** (int) – the record level
- **msg** (str) – the message to be recorded

- **args** – list of arguments
- **kw** – list of keyword arguments

log_format = <logging.Formatter object>
Default log message format

log_level = 20
Current global log level

classmethod removeRootLogHandler(*h*)
Removes the given handler from the root logger

Parameters **h** (Handler) – the handler to be removed

classmethod resetLogFormat()
Resets the log message format (the root log format)

classmethod resetLogLevel()
Resets the log level (the root log level)

root_init_lock = <thread.lock object>
Internal usage

root_inited = True
Internal usage

classmethod setLogFormat(*format*)
sets the new log message format

Parameters **level** (str) – the new log message format

classmethod setLogLevel(*level*)
sets the new log level (the root log level)

Parameters **level** (int) – the new log level

stack(*target=5*)
Log the usual stack information, followed by a listing of all the local variables in each frame.

Parameters **target** (int) – the log level assigned to the record

Return type str

Returns The stack string representation

stream_handler = <logging.StreamHandler object>
the main stream handler

syncLog()
Synchronises the log output

trace(*msg*, **args*, ***kw*)
Record a trace message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.log().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

traceback(*level=5*, *extended=True*)
Log the usual traceback information, followed by a listing of all the local variables in each frame.

Parameters

- **level** (int) – the log level assigned to the traceback record
- **extended** (bool) – if True, the log record message will have multiple lines

Return type str

Returns The traceback string representation

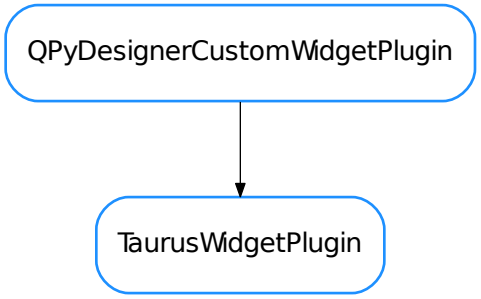
warning(*msg*, **args*, ***kw*)

Record a warning message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.warning().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

TaurusWidgetPlugin



class TaurusWidgetPlugin(*parent=None*)

Bases: PyQt4.QtDesigner.QPyDesignerCustomWidgetPlugin

createWidget(*parent*)

domXml()

getIconName()

getWidgetClass()

getWidgetInfo(*key*, *dft=None*)

group()

Returns the name of the group in Qt Designer's widget box that this widget belongs to. It returns 'Taurus Widgets'. Overwrite if want another group.

icon()

includeFile()

Returns the module containing the custom widget class. It may include a module path.

initialize(*formEditor*)
Overwrite if necessary. Don't forget to call this method in case you want the generic taurus extensions in your widget.

isContainer()

isInitialized()

name()

toolTip()

whatsThis()

- *Logger*
- *TaurusWidgetPlugin*

Functions

Q_TYPEID(*class_name*)
Helper function to generate an IID for Qt. Returns a QString.

taurus.qt.qtgui

This package contains a collection of Qt based widgets designed to interact with taurus models. The widgets are generic in the sence that they do not assume any behavior associated with a specific HW device. They intend to represent only abstract model data.

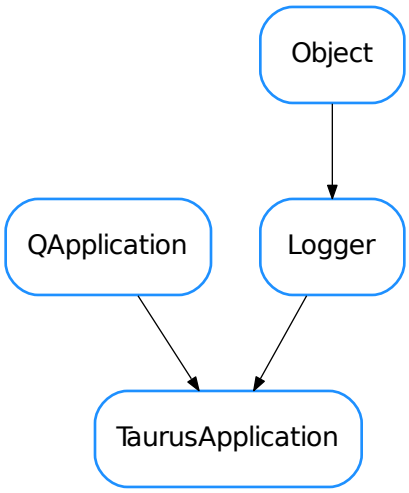
Modules

taurus.qt.qtgui.application

This module provides the base *taurus.qt.qtgui.application.TaurusApplication* class.

Classes

TaurusApplication



class TaurusApplication (**args*, ***kwargs*)

Bases: PyQt4.QtGui.QApplication, taurus.core.util.log.Logger

A QApplication that additionally parses the command line looking for taurus options. This is done using the *taurus.core.util.argparse*. To create a TaurusApplication object you should use the same parameters as in QApplication.

The optional keyword parameters:

- app_name: (str) application name
- app_version: (str) application version
- org_name: (str) organization name
- org_domain: (str) organization domain

...And at last the 'cmd_line_parser' which should be an instance of optparse.OptionParser. Simple example:

```
import sys
import taurus.qt.qtgui.application
import taurus.qt.qtgui.display

app = taurus.qt.qtgui.application.TaurusApplication()

w = taurus.qt.qtgui.display.TaurusLabel()
w.model = 'sys/tg_test/1/double_scalar'
w.show()

sys.exit(app.exec_())
```

A more complex example showing how to add options and a usage help:

```
import sys
import taurus.core.util.argparse
import taurus.qt.qtgui.application
import taurus.qt.qtgui.display

parser = taurus.core.util.argparse.get_taurus_parser()
parser.usage = "%prog [options] <model>"
parser.add_option("--hello")

app = taurus.qt.qtgui.application.TaurusApplication(cmd_line_parser=parser)
args = app.get_command_line_args()
if len(args) < 1:
    sys.stderr.write("Need to supply model attribute")
    sys.exit(1)

w = taurus.qt.qtgui.display.TaurusLabel()
w.model = args[1]
w.show()

sys.exit(app.exec_())
```

For more details on taurus command line parsing check *taurus.core.util.argparse*.

basicConfig(*log_file_name=None*, *maxBytes=10000000.0*, *backupCount=5*, *with_gui_exc_handler=True*)

static exec_(**args*, ***kwargs*)

get_command_line_args()
Returns the list of arguments that resulted from parsing the command line parameters.

Returns the command line arguments

Return type list of strings

get_command_line_options()
Returns the `optparse.Option` that resulted from parsing the command line parameters.

Returns the command line options

Return type `optparse.Option`

get_command_line_parser()
Returns the `optparse.OptionParser` used to parse the command line parameters.

Returns the parser used in the command line

Return type `optparse.OptionParser`

setTaurusStyle(*styleName*)
Sets taurus application style to the given style name

Parameters **styleName** (*str*) – the new style name to be applied

- *TaurusApplication*

taurus.qt.qtgui.base

This package provides the set of base classes from which the Qt taurus widgets should inherit to be considered valid taurus widgets.

Classes

TaurusAttributeControllerHelper

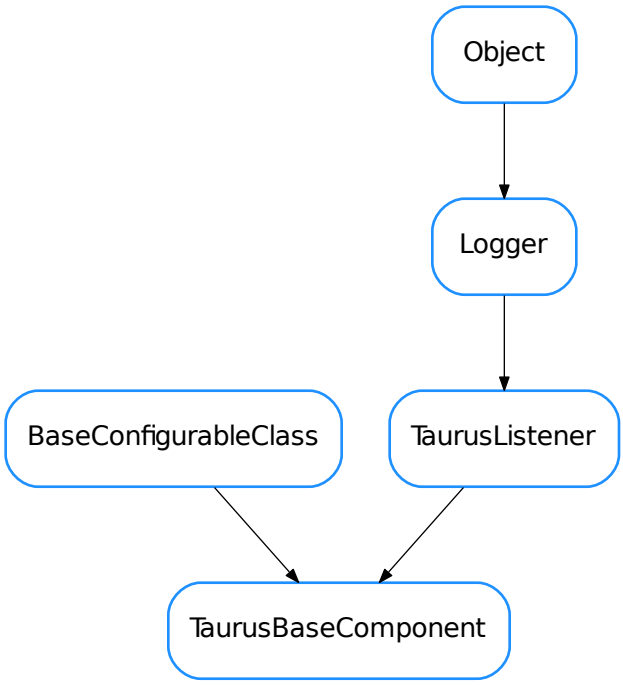
class TaurusAttributeControllerHelper

Bases: object

configObj()

deviceObj()

TaurusBaseComponent



class TaurusBaseComponent(*name*, *parent=None*, *designMode=False*)

Bases: taurus.core.tauruslistener.TaurusListener, taurus.qt.qtcore.configuration.configuration.BaseConfigurableClass

A generic Taurus component.

Note: Any class which inherits from TaurusBaseComponent is expected to also inherit from QObject (or from a QObject derived class).

Note: *getSignaller()* is now unused and deprecated. This is because *taurusEvent* is implemented using baseSignal(), that doesn't require the class to inherit from QObject.

FORMAT(*dtype=None*, *basecomponent=None*, ***kwargs*)

Default formatter callable. Returns a format string based on dtype and the mapping provided by **:attribute:'TaurusBaseComponent.defaultFormatDict'**

Parameters

- **dtype** (object) – data type
- **basecomponent** – widget whose display is to be formatted
- **kwargs** – other keyworld arguments

Return type str

Returns The format string corresponding to the given dtype.

applyPendingOperations(*ops=None*)

Applies operations without caring about danger messages. Use TaurusBaseWidget.safeApplyOperation() if you want to warn the user before applying

Parameters **ops** (sequence <TaurusOperation> or None) – list of operations to apply. If None is given (default) the component fetches the pending operations

contextMenuEvent(*event*)

Handle the popup menu event

Parameters **event** – the popup menu event

defaultFormatDict = {<type 'float'>: '{:.{bc.modelObj.precision}f}', <class 'taurus.external.pint.pint_local.quantity.

deleteLater()

Reimplements the Qt.QObject deleteLater method to ensure that the this object stops listening its model.

displayValue(*v*)

Returns a string representation of the given value

This method will use a format string which is determined dynamically from **:attribute:'FORMAT'**.

By default *TaurusBaseComponent.FORMAT* is set to **:function:'defaultFormatter'**, which makes use of **:attribute:'defaultFormatDict'**.

In order to customize the formatting behaviour, one can change **:attribute:'defaultFormatDict'** or **:attribute:'FORMAT'** directly at class level, or use **:method:'setFormat'** to alter the format string of an specific instance

FORMAT can be set to a python format string [1] or a callable that returns a python format string. If a callable is used, it will be called with the following keyword arguments: - dtype: the data type of the value to be formatted - basecomponent: the affected widget

The following are some examples for customizing the formatting:

•Change FORMAT for all widgets (using a string):

TaurusBaseComponent.FORMAT = "{:.2e}"

•Change FORMAT for all TaurusLabels (using a callable):

def baseFormatter(dtype=None, basecomponent=None, **kwargs): return "{:.1f}"

TaurusLabel.FORMAT = baseFormatter

•Use the defaultFormatter but modify the format string for dtype=str:

TaurusBaseComponent.defaultFormatDict.update({"str": "{!r}"})

[1] https://docs.python.org/2/library/string.html

Parameters **v** (object) – the value to be translated to string

Return type str

Returns a string representing the given value

eventReceived(*evt_src*, *evt_type*, *evt_value*)

The basic implementation of the event handling chain is as follows:

•eventReceived just calls *fireEvent()* which emits a "taurusEvent" PyQt signal that is connected (by *preAttach()*) to the *filterEvent()* method.

•After filtering, *handleEvent()* is invoked with the resulting filtered event

---

Note: in the earlier steps of the chain (i.e., in *eventReceived()*/ *fireEvent()*), the code is executed in a Python thread, while from eventFilter ahead, the code is executed in a Qt thread. When writing widgets, one should normally work on the Qt thread (i.e. reimplementing *handleEvent()*)

Parameters

- **evt_src** (object) – object that triggered the event
- **evt_type** (TaurusEventType) – type of event
- **evt_value** (object) – event value

filterEvent(*evt_src=-1*, *evt_type=-1*, *evt_value=-1*)

The event is processed by each and all filters in strict order unless one of them returns None (in which case the event is discarded)

Parameters

- **evt_src** (object) – object that triggered the event
- **evt_type** (TaurusEventType) – type of event
- **evt_value** (object) – event value

findModelClass()

Do a "best effort" to determine which model type corresponds to the given model name.

Return type class TaurusModel or None

Returns The class object corresponding to the type of Taurus model this widget handles or None if no valid class is found.

fireBufferedEvents()

Fire all events currently buffered (and flush the buffer)

Note: this method is normally called from an event buffer timer thread but it can also be called any time the buffer needs to be flushed

fireEvent(*evt_src=None*, *evt_type=None*, *evt_value=None*)
Emits a "taurusEvent" signal. It is unlikely that you need to reimplement this method in subclasses. Consider reimplementing *eventReceived()* or *handleEvent()* instead depending on whether you need to execute code in the python or Qt threads, respectively

Parameters

- **evt_src** (object or None) – object that triggered the event
- **evt_type** (TaurusEventType or None) – type of event
- **evt_value** (object or None) – event value

getDangerMessage()
Returns the current apply danger message or None if the apply operation is safe

Return type str or None

Returns the apply danger message

getDisplayValue(*cache=True*, *fragmentName=None*)
Returns a string representation of the model value associated with this component.

Parameters

- **cache** (bool) – (ignored, just for bck-compat).
- **fragmentName** (str or None) – the returned value will correspond to the given fragmentName. If None passed, self.modelFragmentName will be used, and if None is set, the defaultFragmentName of the model will be used instead.

Return type str

Returns a string representation of the model value.

getEventBufferPeriod()
Returns the event buffer period

Return type float

Returns period (in s). 0 means event buffering is disabled.

getEventFilters(*preqt=False*)
Returns the list of event filters for this widget

Parameters **preqt** (bool) – If true, return the pre-filters (that are applied in eventReceived, at the python thread), otherwise, return the filters to be applied at the main Qt thread (default)

Return type sequence <callable>

Returns the event filters

getForceDangerousOperations()
Returns if apply dangerous operations is forced

Return type bool

Returns wheter or not apply dangerous operations is forced

getFormatedToolTip(*cache=True*)
Returns a string with contents to be displayed in a tooltip.

Parameters **cache** (bool) – if set to True (default) use the cache value. If set to False will force a connection to the server.

Return type str

Returns a tooltip

getFullModelName()
Returns the full name of the current model object.

Return type str

Returns the model name

getModel()
Returns the model name for this component.

Return type str

Returns the model name.

getModelClass()
Return the class object for the widget. Default behavior is to do a 'best effort' to determine which model type corresponds to the current model name. Overwrite as necessary.

Return type class TaurusModel or None

Returns The class object corresponding to the type of Taurus model this widget handles or None if no valid class is found.

getModelFragmentObj(*fragmentName=None*)
Returns a fragment object of the model. A fragment of a model is a python attribute of the model object.

Fragment names including dots will be used to recursively get fragments of fragments.

For a simple fragmentName (no dots), this is roughly equivalent to getattr(self.getModelObj(), fragmentName)

If the model does not have that fragment, AttributeError is raised (other exceptions may be raised when accessing the fragment as well)

Parameters **fragmentName** (str or None) – the returned value will correspond to the given fragmentName. If None passed, self.modelFragmentName will be used, and if None is set, the defaultFragmentName of the model will be used instead.

Return type obj

Returns the member of the modelObj referred by the fragment.

getModelInConfig()

getModelName()
Returns the current model name.

Return type str

Returns the model name

getModelObj()
Returns the taurus model obj associated with this component or None if no taurus model is associated.

Return type TaurusModel or None

Returns the taurus model object

getModelType()
Returns the taurus model type associated with this component or taurus.core.taurusbasetypes.TaurusElementType.Unknown if no taurus model is associated.

Return type TaurusElementType

Returns the taurus model type

getModelValueObj(*cache=True*)
Returns the tango obj value associated with this component or None if no taurus model is associated.

Parameters **cache** (bool) – if set to True (default) use the cache value. If set to False will force a connection to the server.

Return type TaurusAttrValue

Returns the tango value object.

getNoneValue()
Returns the current string representation when no valid model or model value exists.

Return type str

Returns a string representation for an invalid value

getParentModelName()
Returns the parent model name or an empty string if the component has no parent

Return type str

Returns the parent model name

getParentModelObj()
Returns the parent model object or None if the component has no parent or if the parent's model is None

Return type TaurusModel or None

Returns the parent taurus model object

getParentTaurusComponent()
Returns a parent Taurus component or None if no parent *taurus.qt.qtgui.base.TaurusBaseComponent* is found.

Raise RuntimeError

getPendingOperations()
Returns the sequence of pending operations

Return type sequence <TaurusOperation>

Returns a list of pending operations

getShowQuality()
Returns if showing the quality as a background color

Return type bool

Returns True if showing the quality or False otherwise

getShowText()
Returns if showing the display value

Return type bool

Returns True if showing the display value or False otherwise

getSignaller(**args*, ***kwargs*)
Deprecated since version 4.0.

getTaurusFactory(*scheme=''*)
Returns the taurus factory singleton for the given scheme. This is just a helper method. It is the equivalent of doing:

```
import taurus
factory = taurus.Factory(scheme)
```

Parameters **scheme** (str or None) – the scheme. If scheme is an empty string, or is not passed, the scheme will be obtained from the model name. For backwards compatibility (but deprecated), passing None is equivalent to 'tango'.

Return type TaurusFactory

Returns the TaurusFactory

getTaurusManager()
Returns the taurus manager singleton. This is just a helper method. It is the equivalent of doing:

```
import taurus
manager = taurus.Manager()
```

Return type TaurusManager

Returns the TaurusManager

getTaurusPopupMenu()
Returns an xml string representing the current taurus popup menu

Return type str

Returns an xml string representing the current taurus popup menu

getUseParentModel()
Returns whether this component is using the parent model

Return type bool

Returns True if using parent model or False otherwise

handleEvent(*evt_src*, *evt_type*, *evt_value*)
Event handling. Default implementation does nothing. Reimplement as necessary

Parameters

- **evt_src** (object or None) – object that triggered the event
- **evt_type** (TaurusEventType or None) – type of event
- **evt_value** (object or None) – event value

hasPendingOperations()
Returns if the component has pending operations

Return type bool

Returns True if there are pending operations or False otherwise

insertEventFilter(*filter*, *index=-1*, *preqt=False*)
insert a filter in a given position

Parameters

- **filter** (callable(evt_src, evt_type, evt_value) –) a filter
- **index** (int) – index to place the filter (default = -1 meaning place at the end)

- **preqt** (bool) – If true, set the pre-filters (that are applied in eventReceived, at the python thread), otherwise, set the filters to be applied at the main Qt thread (default)

See also: setEventFilters

isAttached()
Determines if this component is attached to the taurus model.

Return type bool

Returns True if the component is attached or False otherwise.

isAutoProtectOperation()
Tells if this widget's operations are protected against exceptions

Return type bool

Returns True if operations are protected against exceptions or False otherwise

isChangeable()
Tells if this component value can be changed by the user. Default implementation will return True if and only if:

•this component is attached to a valid taurus model and

•the taurus model is writable and

•this component is not read-only

Return type bool

Returns True if this component value can be changed by the user or False otherwise

isDangerous()
Returns if the apply operation for this component is dangerous

Return type bool

Returns wheter or not the apply operation for this component is dangerous

isModifiableByUser()
whether the user can change the contents of the widget

Return type bool

Returns True if the user is allowed to modify the look&feel

isPaused()
Return the current pause state

Return type bool

Returns wheater or not the widget is paused

isReadOnly()
Determines if this component is read-only or not in the sense that the user can interact with it. Default implementation returns True.

Override when necessary.

Return type bool

Returns whether or not this component is read-only

postAttach()
Called inside self.attach() after actual attach is performed. Default implementation does not do anything.

Override when necessary.

postDetach()
Called inside self.detach() after actual deattach is performed. Default implementation does not do anything.

Override when necessary.

preAttach()
Called inside self.attach() before actual attach is performed. Default implementation just emits a signal.

Override when necessary.

preDetach()
Called inside self.detach() before actual deattach is performed. Default implementation just disconnects a signal.

Override when necessary.

resetAutoProtectOperation()
Resets protecting operations

resetDangerMessage()
Clears the danger message. After this method is executed the apply operation for this component will be considered safe.

resetForceDangerousOperations()
Clears forcing apply dangerous operations

resetFormat()
Reset the internal format string. It forces a recalculation in the next call to **:method:'displayValue'**.

resetModel()
Sets the model name to the empty string

resetModelInConfig()

resetModifiableByUser()
Equivalent to setModifiableByUser(self.__class__._modifiableByUser)

resetPendingOperations()
Clears the list of pending operations

resetShowQuality()
Resets the show quality to self.__class__._showQuality

resetShowText()
Resets the showing of the display value to True

resetTaurusPopupMenu()
Resets the taurus popup menu to empty

resetUseParentModel()
Resets the usage of parent model to False

setAutoProtectOperation(*protect*)
Sets/unsets this widget's operations are protected against exceptions

Parameters **protect** (*bool*) – wheater or not to protect widget operations

setDangerMessage(*dangerMessage=''*)
Sets the danger message when applying an operation. If dangerMessage is None, the apply operation is considered safe

Parameters **dangerMessage** (str or None) – the danger message. If None is given (default) the apply operation is considered safe

setEventBufferPeriod(*period*)
Set the period at wich *fireBufferedEvents()* will be called. If period is 0, the event buffering is disabled (i.e., events are fired as soon as they are received)

Parameters **period** (float) – period in seconds for the automatic event firing. period=0 will disable the event buffering.

setEventFilters(*filters=None*, *preqt=False*)
sets the taurus event filters list. The filters are run in order, using each output to feed the next filter. A filter must be a function that accepts 3 arguments `(evt_src, evt_type, evt_value)` If the event is to be ignored, the filter must return None. If the event is not to be ignored, filter must return a `(evt_src, evt_type, evt_value)` tuple which may (or not) differ from the input.

For a library of common filters, see taurus/core/util/eventfilters.py

Parameters

- **filters** (sequence) – a sequence of filters
- **preqt** (bool) – If true, set the pre-filters (that are applied in eventReceived, at the python thread), otherwise, set the filters to be applied at the main Qt thread (default)

Note: If you are setting a filter that applies a transformation on the parameters, you may want to generate a fake event to force the last value to be filtered as well. This can be done as in this example:

```
TaurusBaseComponent.fireEvent( TaurusBaseComponent.getModelObj(),
                               taurus.core.taurusbasetypes.TaurusEventType.
↪Periodic,
                               TaurusBaseComponent.getModelObj().getValueObj())
```

See also: insertEventFilter

setForceDangerousOperations(*yesno*)
Forces/clears the dangerous operations

Parameters **yesno** (bool) – force or not the dangerous operations

setFormat(*format*)
Method to set the *FORMAT* attribute for this instance. It also resets the internal format string, which will be recalculated in the next call to :method"*displayValue*

Parameters **format** (str or callable) – A format string or a callable that returns it

setModel(*model*)
Sets/unsets the model name for this component

Parameters **model** (str) – the new model name

setModelCheck(*model*, *check=True*)
Sets the component taurus model. Setting the check argument to True (default) will check if the current model is equal to the given argument. If so then nothing is done. This should be the standard way to call this method since it will avoid recursion.

Parameters

- **model** (str) – the new model name

- **check** (bool) – whether or not to check against the actual model name

setModelInConfig(*yesno*)
Sets whether the model-related properties should be stored for this widget when creating the config dict with `createConfig()` (and restored when calling `applyConfig()`). By default this is not enabled. The following properties are affected by this: - "model"

Parameters **yesno** (bool) – If True, the model-related properties will be registered as config properties. If False, they will be unregistered.

See also:

`registerConfigProperty()`, `createConfig()`, `applyConfig()`

setModelName(*modelName*, *parent=None*)
This method will detach from the previous taurus model (if any), it will set the new model to the given modelName and it will attach this component to the new taurus model.

Parameters

- **modelName** (str) – the new taurus model name (according to the taurus convention)
- **parent** (*TaurusBaseComponent*) – the parent or None (default) if this component does not have a parent Taurus component

setModifiableByUser(*modifiable*)
sets whether the user is allowed to modify the look&feel

Parameters **modifiable** (bool) –

setNoneValue(*v*)
Sets the new string representation when no model or no model value exists.

Parameters **v** (str) – the string representation for an invalid value

setPaused(*paused=True*)
Toggles the pause mode.

Parameters **paused** (bool) – whether or not to pause (default = True)

setShowQuality

setShowText

setTaurusPopupMenu(*menuData*)
Sets/unsets the taurus popup menu

Parameters **menuData** (str) – an xml representing the popup menu

setUseParentModel

taurusEvent
Base signal taurusEvent

toolTipObjToStr(*toolTipObj*)
Converts a python dict to a tooltip string.

Parameters **toolTipObj** (dict) – a python object

Return type str

Returns a tooltip

updateStyle()
Method called when the component detects an event that triggers a change in the style. Default implementation doesn't do anything. Overwrite when necessary

TaurusBaseController

class TaurusBaseController(*widget*, *updateAsPalette=True*)

Bases: object

Base class for all taurus controllers

attrObj()

configObj()

deviceObj()

eventReceived(*evt_src*, *evt_type*, *evt_value*)

getDisplayValue(*write=False*)

handleEvent(*evt_src*, *evt_type*, *evt_value*)

modelObj()

quality()

state()

update()

usePalette()

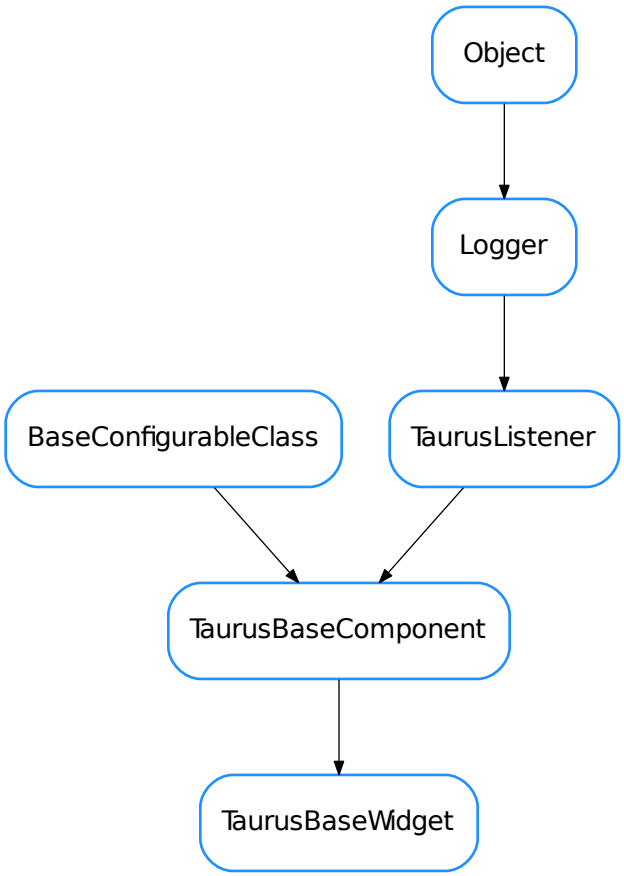
value()

valueObj()

w_value()

widget()

TaurusBaseWidget



class TaurusBaseWidget (*name*, *parent=None*, *designMode=False*)

Bases: `taurus.qt.qtgui.base.taurusbase.TaurusBaseComponent`

The base class for all Qt Taurus widgets.

Note: Any class which inherits from TaurusBaseWidget is expected to also inherit from QWidget (or from a QWidget derived class)

---

changeEvent (*evt*)

overwrites QWidget.changeEvent(self, evt) to handle the ParentChangeEvent in case this widget is using the parent model. Always calls the QWidget.changeEvent in order not to lose events

closeEvent (*event*)

Override of the QWidget.closeEvent()

dragEnterEvent (*event*)

reimplemented to support drag&drop of models. See `QWidget`

dropEvent(*event*)
reimplemented to support drag&drop of models. See QWidget

emitValueChanged(**args*)
Connect the specific XXXXChanged signals from derived classes to this method in order to have a unified signal which can be used by Taurus Widgets

getAutoTooltip()
Returns if the widget is automatically generating a tooltip based on the current widget model.

Return type bool

Returns True if automatically generating tooltip or False otherwise

getDropEventCallback()
returns the method to be called when a dropping event occurs. The default implementation returns *self.setModel*. Reimplement it subclasses to call different methods.

Return type callable

Returns

getModelMimeData()
Returns a MimeData object containing the model data. The default implementation fills the *TAURUS_MODEL_MIME_TYPE*. If the widget's Model class is Attribute or Device, it also fills *TAURUS_ATTR_MIME_TYPE* or *TAURUS_DEV_MIME_TYPE*, respectively

Return type QMimeData

Returns

getParentTaurusComponent()
Returns the first taurus component in the widget hierarchy or None if no taurus component is found

Return type *TaurusBaseComponent* or None

Returns the parent taurus base component

getQtClass(*bases=None*)
Returns the parent Qt class for this widget

Parameters **bases** (sequence <class> or None) – the list of class objects. If None is given (default) it uses the object base classes from __bases__

Return type QWidget class

Returns the QWidget class object

classmethod getQtDesignerPluginInfo()
Returns pertinent information in order to be able to build a valid QtDesigner widget plugin.

The dictionary returned by this method should contain *at least* the following keys and values: - 'module' : a string representing the full python module name (ex.: 'taurus.qt.qtgui.base') - 'icon' : a string representing valid resource icon (ex.: 'designer:combobox.png') - 'container' : a bool telling if this widget is a container widget or not.

This default implementation returns the following dictionary:

```
{ 'group'     : 'Taurus [Unclassified]',
  'icon'      : 'logos:taurus.png',
  'container' : False }
```

Return type dict

Returns a map with pertinent designer information

getSupportedMimeTypes()
returns a list of supported mimeTypes that this widget support (ordered by priority). If none is set explicitely via *setSupportedMimeTypes()*, a best effort will be tried based on the model class

..seealso: *setSupportedMimeTypes()*

This provides only a very basic implementation. Reimplement in derived classes if needed

Return type list <str>

Returns list of MIME type names

handleEvent(*evt_src*, *evt_type*, *evt_value*)
very basic and generalistic handling of events.

Override when necessary.

Parameters

- **evt_src** (object or None) – object that triggered the event
- **evt_type** (TaurusEventType or None) – type of event
- **evt_value** (object or None) – event value

handleException(*e*)

handleMimeData(*mimeData*, *method*)
Selects the most appropriate data from the given mimeData object (in the order returned by *getSupportedMimeTypes()*) and passes it to the given method.

Parameters

- **mimeData** (QMimeData) – the MIME data object from which the model is to be extracted
- **method** (callable <str>) – a method that accepts a string as argument. This method will be called with the data from the mimeData object

Return type str or None

Returns returns the MimeType used if the model was successfully set, or None if the model could not be set

hideEvent(*event*)
Override of the QWidget.hideEvent()

isDragEnabled()
whether the user can drag data from this widget

Return type bool

Returns True if the user can drag data

modelChanged
Base signal modelChanged

mouseMoveEvent(*event*)
reimplemented to provide drag events. See QWidget

mousePressEvent(*event*)
reimplemented to record the start position for drag events. See QWidget

parentModelChanged(*parentmodel_name*)
Invoked when the Taurus parent model changes

Parameters **parentmodel_name** (str) – the new name of the parent model

recheckTaurusParent()
Forces the widget to recheck its Taurus parent. Taurus Widgets will in most situations keep track of changes in their taurus parenting, but in some special cases (which unfortunately tend to occur when using Qt Designer) they may not update it correctly.

If this happens, you can manually call this method.

For more information, check the `issue demo example`

resetDragEnabled()
Equivalent to setDragEnabled(self.__class__._dragEnabled)

safeApplyOperations(*ops=None*)
Applies the given operations (or the pending operations if None passed)

Parameters **ops** (sequence <TaurusOperation> or None) – list of operations to apply. If None is given (default) the component fetches the pending operations

Return type bool

Returns False if the apply was aborted by the user or if the widget is in design mode. True otherwise.

setAutoTooltip(*yesno*)
Determines if the widget should automatically generate a tooltip based on the current widget model.

Parameters **yesno** (bool) – True to automatically generate tooltip or False otherwise

setDisconnectOnHide(*disconnect*)
Sets/unsets disconnection on hide event

Parameters **disconnect** (bool) – whether or not to disconnect on hide event

setDragEnabled(*enabled*)
sets whether the user is allowed to drag data from this widget

Parameters **modifiable** (bool) –

setModelCheck(*model*, *check=True*)
Sets the component taurus model. Setting the check argument to True (default) will check if the current model is equal to the given argument. If so then nothing is done. This should be the standard way to call this method since it will avoid recursion.

Parameters

- **model** (str) – the new model name
- **check** (bool) – whether or not to check against the actual model name

setModelInConfig(*yesno*)
extends *TaurusBaseComponent.setModelInConfig()* to include also the "useParentModel" property

See also:

TaurusBaseComponent.setModelInConfig()

setModifiableByUser(*modifiable*)
Reimplemented to acept/reject drops based on whether the widget is modifiable by the user. See *TaurusBaseComponent.setModifiableByUser()*

setSupportedMimeTypes(*mimetypes*)
sets the mimeTypes that this widget support

Parameters **mimetypes** (list <str>) – list (ordered by priority) of MIME type names

setUseParentModel

showEvent(*event*)
Override of the QWidget.showEvent()

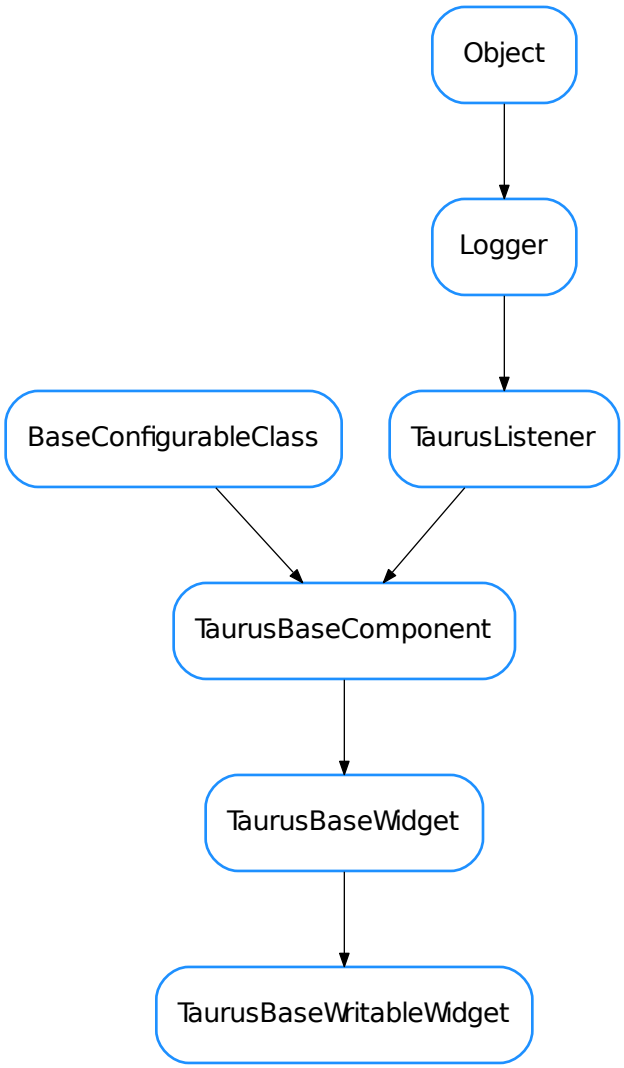
updatePendingOpsStyle()
This method should be reimplemented by derived classes that want to change their appearance depending whether there are pending operations or not

updateStyle()
Updates the widget style. Default implementation just calls QWidget.update()

Override when necessary.

valueChangedSignal
Base signal valueChanged

TaurusBaseWritableWidget



class **TaurusBaseWritableWidget** (*name*, *taurus_parent=None*, *designMode=False*)

Bases: `taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget`

The base class for all taurus input widgets

it emits the applied signal when the value has been applied.

applied

Base signal applied

forceApply()

It (re)applies the value regardless of pending operations. WARNING: USE WITH CARE. In most cases what you need is to make sure that pending operations are properly created, not calling this method

It emits the applied signal if apply is not aborted.

getAutoApply()
whether autoApply mode is enabled or not.

Return type bool

Returns

getForcedApply()
whether forcedApply mode is enabled or not.

Return type bool

Returns

getModelClass()
reimplemented from *TaurusBaseWidget*

getOperationCallbacks()

returns the operation callbacks (i.e., a sequence of methods that will be called after an operation is executed
(this default implementation it returns an empty list).

Return type sequence<callable>

Returns

classmethod getQtDesignerPluginInfo()
reimplemented from *TaurusBaseWidget*

getValue()
This method must be implemented in derived classes to return the value to be written. Note that this may differ from the displayed value (e.g. for a numeric value being edited by a QLineEdit-based widget, the displayed value will be a string while getValue will return a number)

handleEvent(*src*, *evt_type*, *evt_value*)
reimplemented from *TaurusBaseWidget*

isReadOnly()
reimplemented from *TaurusBaseWidget*

notifyValueChanged(**args*)
Subclasses should connect some particular signal to this method for indicating that something has changed. e.g., a QLineEdit should connect its "textChanged" signal...

postAttach()
reimplemented from *TaurusBaseWidget*

resetAutoApply()
resets the autoApply mode (i.e.: sets it to False)

resetForcedApply()
resets the forcedApply mode (i.e.: sets it to False)

resetPendingOperations()
reimplemented from *TaurusBaseWidget*

setAutoApply(*auto*)
Sets autoApply mode. In autoApply mode, the widget writes the value automatically whenever it is changed by the user (e.g., when *notifyValueChanged()* is called). If False, a value changed just flags a "pending operation" which needs to be applied manually by the user before the value gets written.

Parameters **auto** (bool) – True for setting autoApply mode. False for disabling

setForcedApply(*forced*)
Sets the forcedApply mode. In forcedApply mode, values are written even if there are not pending operations (e.g. even if the displayed value is the same as the currently applied one).

Parameters **forced** (bool) – True for setting forcedApply mode. False for disabling

setValue(*v*)
This method must be implemented in derived classes to provide a (widget-specific) way of updating the displayed value based on a given attribute value

Parameters **v** – The attribute value

updatePendingOperations()
reimplemented from *TaurusBaseWidget*

updateStyle()
reimplemented from *TaurusBaseWidget*

valueChanged(**args*, ***kwargs*)
Deprecated since version 4.0: Use notifyValueChanged instead

writeValue(*forceApply=False*)
Writes the value to the attribute, either by applying pending operations or (if the ForcedApply flag is True), it writes directly when no operations are pending

It emits the applied signal if apply is not aborted.

Parameters **forceApply** (bool) – If True, it behaves as in forceApply mode (even if the forceApply mode is disabled by `setForceApply()`)

TaurusConfigurationControllerHelper

class TaurusConfigurationControllerHelper
Bases: object

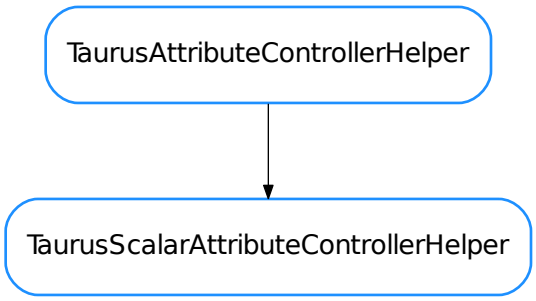
attrObj()

configParam

deviceObj()

getDisplayValue(*write=False*)

TaurusScalarAttributeControllerHelper



class TaurusScalarAttributeControllerHelper

Bases: taurus.qt.qtgui.base.tauruscontroller.TaurusAttributeControllerHelper

displayValue(*value*)

getDisplayValue(*write=False*)

- *TaurusAttributeControllerHelper*
- *TaurusBaseComponent*
- *TaurusBaseController*
- *TaurusBaseWidget*
- *TaurusBaseWritableWidget*
- *TaurusConfigurationControllerHelper*
- *TaurusScalarAttributeControllerHelper*

Functions

defaultFormatter(*dtype=None*, *basecomponent=None*, ***kwargs*)

Default formatter callable. Returns a format string based on dtype and the mapping provided by **:attribute:'TaurusBaseComponent.defaultFormatDict'**

Parameters

- **dtype** (object) – data type
- **basecomponent** – widget whose display is to be formatted
- **kwargs** – other keyworld arguments

Return type str

Returns The format string corresponding to the given dtype.

updateLabelBackground(*ctrl*, *widget*)

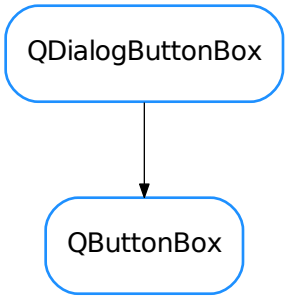
Helper method to setup background of taurus labels and lcds

taurus.qt.qtgui.button

This package contains a collection taurus Qt button widgets

Classes

QButtonBox



class **QButtonBox**(*parent=None*, *designMode=False*, *buttons=None*, *orientation=1*)

Bases: PyQt4.QtGui.QDialogButtonBox

abortClicked

applyClicked

cancelClicked

closeClicked

discardClicked

helpClicked

ignoreClicked

noClicked

okClicked

onClicked(*button*)

openClicked

resetClicked

restoreDefaultsClicked

retryClicked

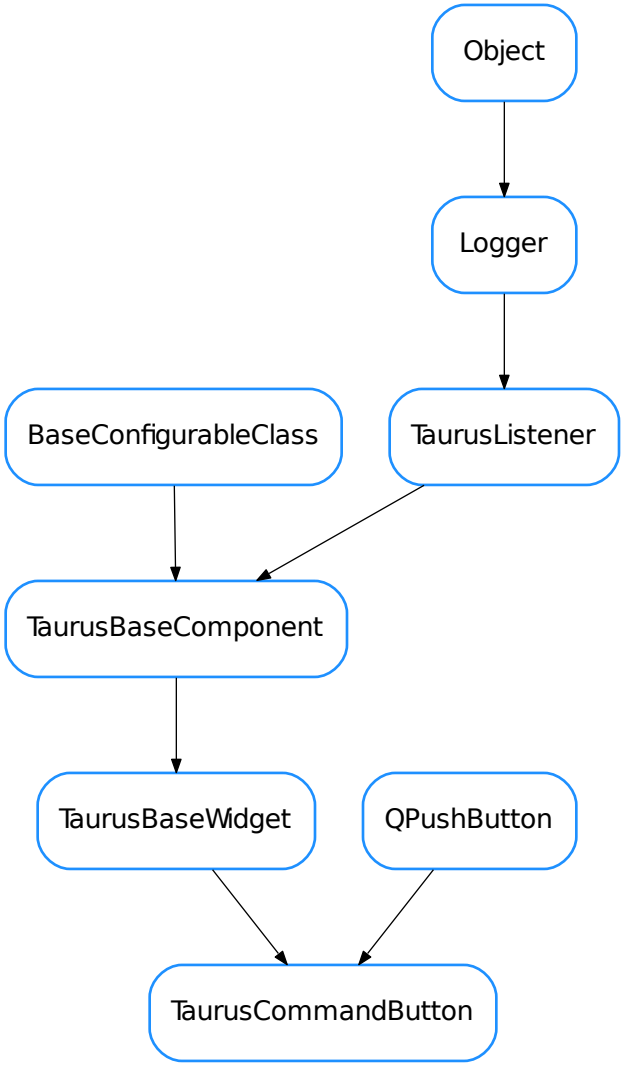
saveAllClicked

saveClicked

yesClicked

yesToAllClicked

TaurusCommandButton



class TaurusCommandButton(*parent=None*, *designMode=False*, *command=None*, *parameters=None*, *icon=None*, *text=None*, *timeout=None*)

Bases: `PyQt4.QtGui.QPushButton`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget`

This class provides a button that executes a tango command on its device.

Code examples:

```
# a button that executes the "status" command for the 'a/b/c' device server
button =  TaurusCommandButton(command = 'Status', icon='logos:taurus.png')
button.setModel('a/b/c')

#  a button that executes the "exec" command for the 'a/b/c' device server with␣
↪one parameter
button =  TaurusCommandButton(command = 'Status', parameters=['2+2'],icon=
↪'logos:taurus.png')
button.setModel('a/b/c')
```

See also:

`TaurusCommandsForm` provides a good example of use of TaurusCommandButton (including managing the return value)

Command

CustomText

DangerMessage

Model

Parameters

Timeout

UseParentModel

commandExecuted

getCommand()
returns the command name to be executed when the button is clicked

Return type `str` or `None`

Returns the command name

getCustomText()
Returns the custom text of the buttom, or None if no custom text is used

getDisplayValue()
see `TaurusBaseComponent.displayValue()`

getParameters()
returns the parameters to be used on command execution

Parameters **parameters** (`sequence`) –

classmethod getQtDesignerPluginInfo()

getTimeout()
Returns the number of seconds to wait for the result of the command (or -1 if timeout is disabled)

onClicked(**args*, ***kwargs*)

resetCommand()
equivalent to self.setCommand(None)

resetCustomText()
Equivalent to setCustomText(None)

resetParameters()
Equivalent to setParameters([])

resetTimeout()
Equivalent to setTimeout(None)

setCommand(*commandName*)
sets the command to be executed when the button is clicked

Parameters **commandName** (str or None) – the command name

setCustomText(*customText=None*)
Sets a custom text for the button (by default it is the command name)

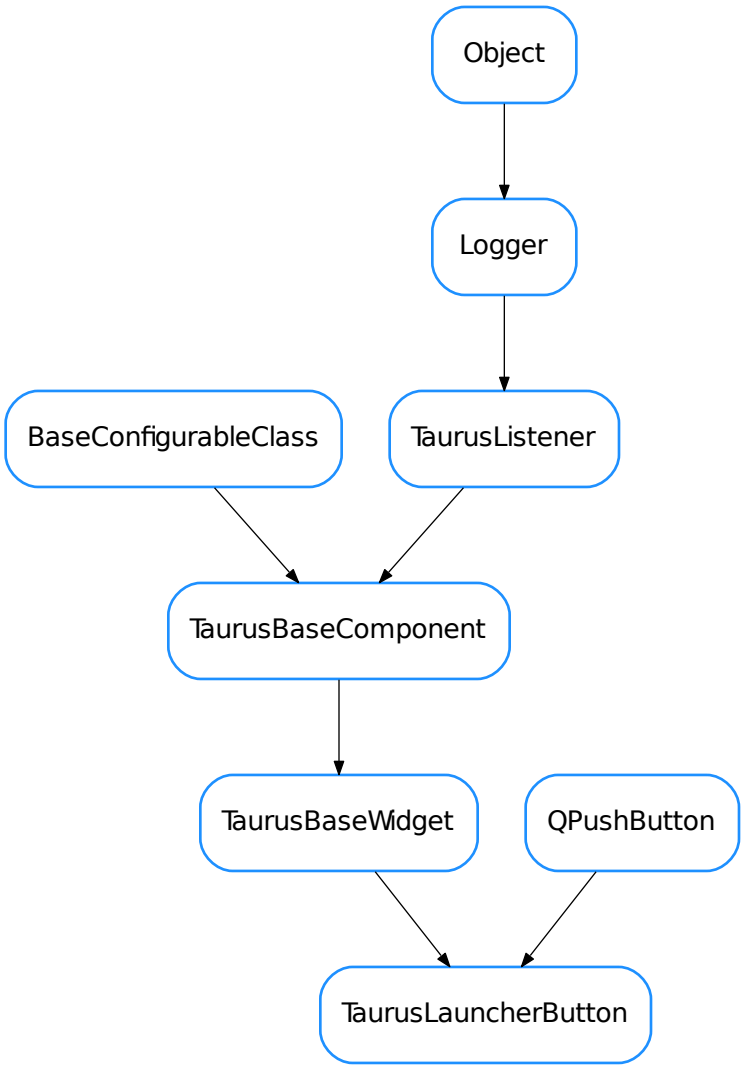
Parameters **customText** (str or None) – the custom text. If None passed, it will use the command name

setParameters(*parameters*)
Sets the parameters to be used on command execution.

Parameters **parameters** (sequence) – a sequence of parameters. If the elements of the sequence are not of the right type required for the parameter, an automatic conversion will be attempted on execution time. As a special case, if parameters is a string not starting and ending in quote characters, it will be splitted on whitespace to obtain a sequence of parameters. If it is a string starting and ending with quotes, the quotes will be removed and the quoted text will not be splitted.

setTimeout

TaurusLauncherButton



class TaurusLauncherButton(*parent=None*, *designMode=False*, *widget=None*, *icon=None*, *text=None*)

Bases: `PyQt4.QtGui.QPushButton`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget`

This class provides a button that launches a modeless dialog containing a specified Taurus widget which gets the same model as the button. The button does not use the model directly. Instead it passes it to the associated widget.

Code examples:

```
# a button that launches a TaurusAttrForm when clicked
button =  TaurusLauncherButton(widget = TaurusAttrForm())
button.setModel('a/b/c') #a device name, which will be set at the TaurusAttrForm␣
↪when clicking

# a button that launches a taurusLabel (whose model is an attribute: 'a/b/c/
↪attrname')
button =  TaurusLauncherButton(widget = TaurusLabel())
button.setModel('a/b/c/attrname') # attr name, which will be set at the␣
↪TaurusLabel when clicking

#same as the previous one, but using the parent model and putting a custom text␣
↪and icon:
button =  TaurusLauncherButton(widget = TaurusLabel(), text='click me', icon=
↪'logos:taurus.png')
button.setUseParentModel(True)  #let's assume that the button's parent has a␣
↪model of type "/a/b/c"
button.setModel('/attrname')
```

Model

UseParentModel

createWidget()

displayValue(*v*)
see TaurusBaseComponent.displayValue()

getDisplayValue()
see TaurusBaseComponent.getDisplayValue()

getModelClass()
see TaurusBaseComponent.getModelClass(). Note that in the case of *TaurusLauncherButton*, the class is completely dependent on the widget's class

classmethod getQtDesignerPluginInfo()

getWidgetClassName()

onClicked()
Slot called when the button is clicked. Note that the dialog will only be created once. Subsequent clicks on the button will only raise the existing dialog

resetWidgetClassName(*className*, *args=None*, *kwargs=None*)

setText(*text*)
Sets the text of the button. see Qt.QPushButton.setText()

setWidget(*widget*)
sets the widget that will be shown when clicking the button

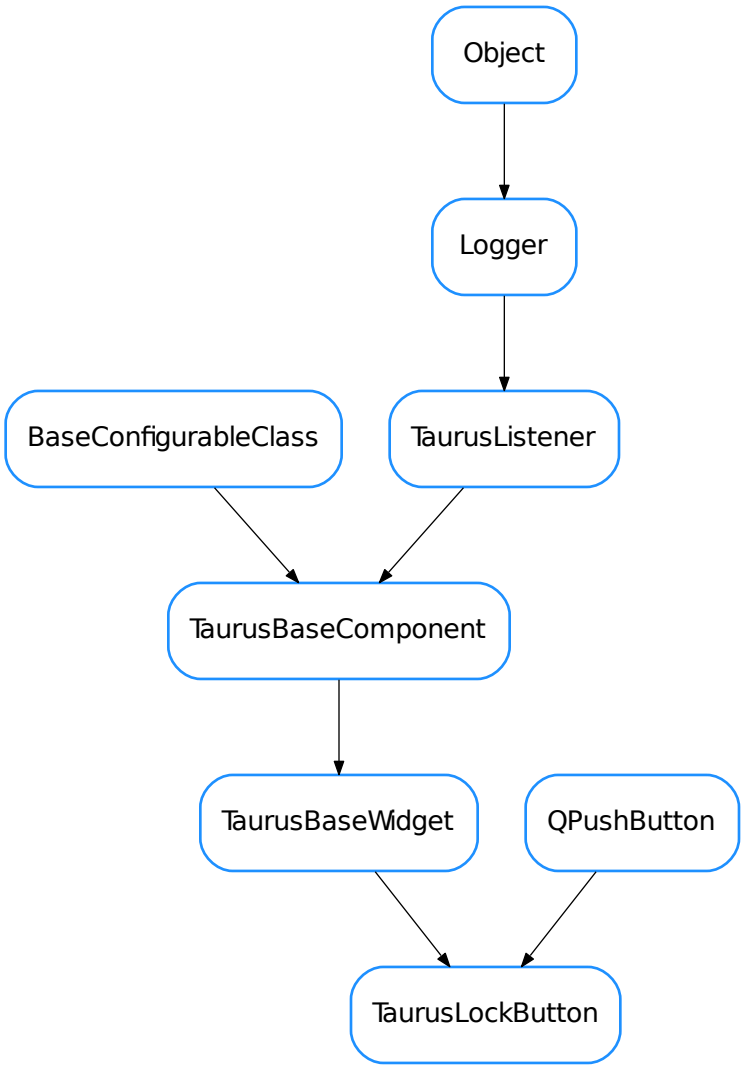
Parameters **widget** (QWidget) –

setWidgetClassName(*className*, *args=None*, *kwargs=None*)

widget()

widgetClassName

TaurusLockButton



class TaurusLockButton(*parent=None*, *designMode=False*)

Bases: PyQt4.QtGui.QPushButton, taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget

getModelClass()

classmethod getQtDesignerPluginInfo()

get_lock_info(*cache=False*)

model

on_toggle(*down*)

setModel(*model*)

update_button(*lock_info=None*)

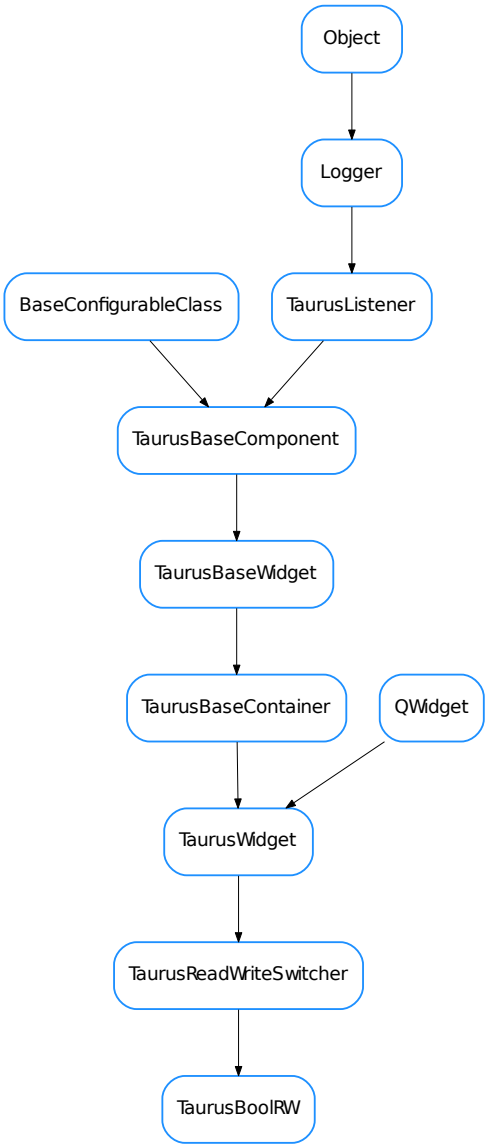
- *QButtonBox*
- *TaurusCommandButton*
- *TaurusLauncherButton*
- *TaurusLockButton*

taurus.qt.qtgui.compact

This package contains a collection of taurus Qt widgets that combine other widgets to provide several functionalities in a reduced space

Classes

TaurusBoolRW



class TaurusBoolRW (*parent=None*, *designMode=False*, *readWClass=None*, *writeWClass=None*, *enterEditTriggers=None*, *exitEditTriggers=None*)

Bases: `taurus.qt.qtgui.compact.abstractswitcher.TaurusReadWriteSwitcher`

A Switcher combining a TaurusLed and a TaurusValueCheckBox

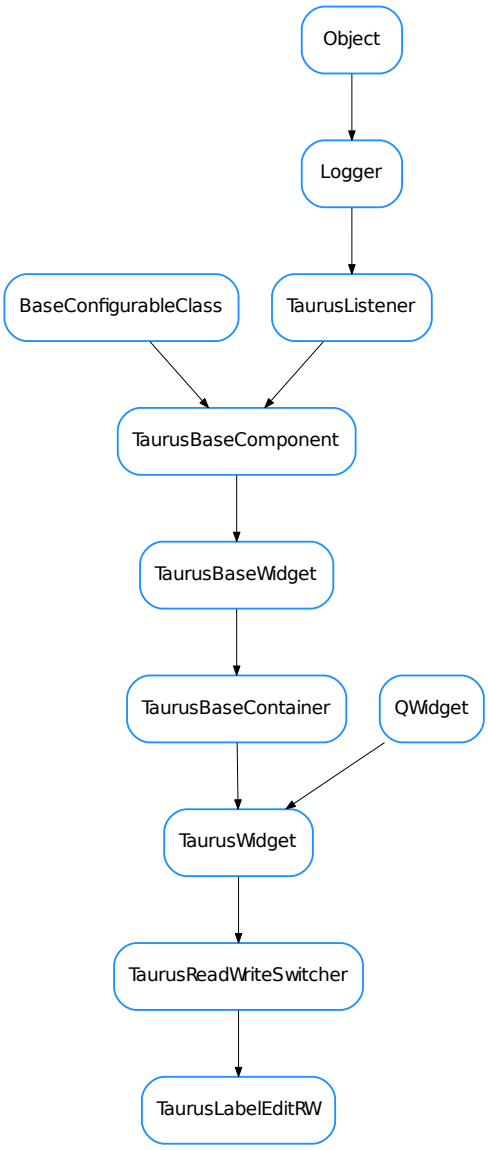
readWClass

alias of `TaurusLed`

setWriteWidget (*widget*)

writeWClass
alias of `TaurusValueCheckBox`

TaurusLabelEditRW



class **TaurusLabelEditRW**(*parent=None*, *designMode=False*, *readWClass=None*, *writeWClass=None*, *enterEditTriggers=None*, *exitEditTriggers=None*)

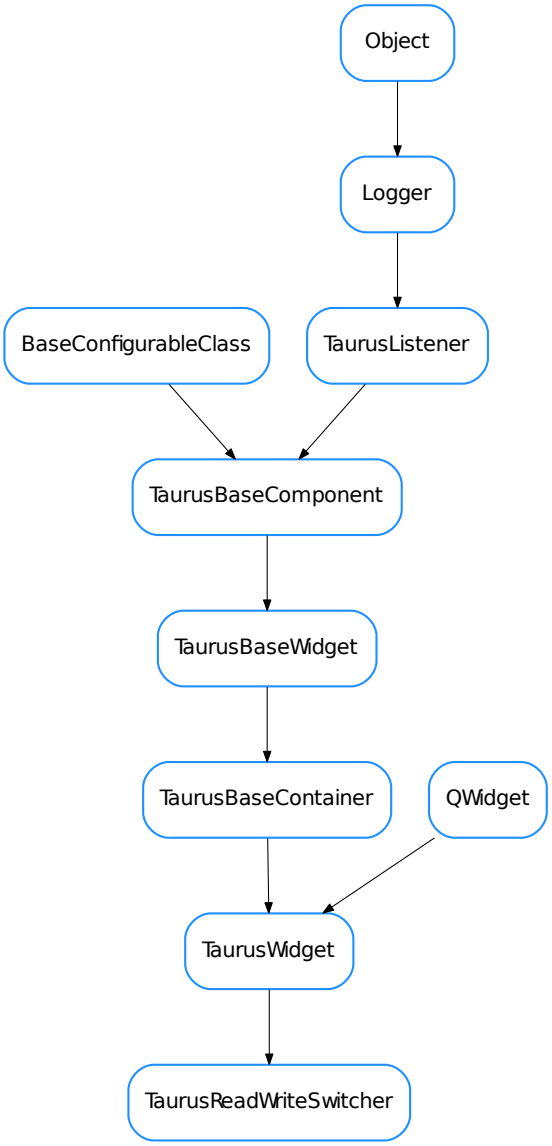
Bases: `taurus.qt.qtgui.compact.abstractswitcher.TaurusReadWriteSwitcher`

A Switcher combining a TaurusLabel and a TaurusValueLineEdit

readWClass
alias of `TaurusLabel`

writeWClass
alias of `TaurusValueLineEdit`

TaurusReadWriteSwitcher



class TaurusReadWriteSwitcher(*parent=None*, *designMode=False*, *readWClass=None*, *writeWClass=None*, *enterEditTriggers=None*, *exitEditTriggers=None*)
Bases: `taurus.qt.qtgui.container.tauruswidget.TaurusWidget`

This is a base class for creating widgets that can switch beetween read and write mode by combining a Taurus widget for reading and a Taurus Widget for writing.

For example, if you want to combine a TaurusLabel with a TaurusValueLineEdit, you can implement it as follows:

```
class MyRWSwitcher(TaurusReadWriteSwitcher):
    readWClass = TaurusLabel
    writeWClass = TaurusValueLineEdit
```

Alternatively, you can instantiate the TaurusReadWriteSwitcher class directly and pass the read and write classes to the constructor:

```
w = TaurusReadWriteSwitcher(readWClass=TaurusLabel,
                            writeWClass=TaurusValueLineEdit)
```

Or you can even set the read and write widgets (instead of classes) after instantiation:

```
w = TaurusReadWriteSwitcher()
a = TaurusLabel()
b = TaurusValueLineEdit()
w.setReadWidget(a)
w.setWriteWidget(b)
```

TaurusReadWriteSwitcher will normally show the read widget by default, but it will allow to switch to "edit mode" (where the write widget is shown instead). Enetering and exiting the edit mode is controlled by "triggers". Triggers can be key presses, QEvents or signals.

The default implementation sets pressing F2 or doubleclicking the read widget as the triggers for entering edit mode, and pressing Escape, losing the focus or applying the value on the write widget as the triggers for leaving the edit mode. This can be customized by changing *enterEditTriggers* and *exitEditTriggers* class members or by passing *enterEditTriggers* and *exitEditTriggers* keyword parameters to the constructor of TaurusReadWriteSwitcher:

- *enterEditTriggers* is a tuple containing one or more of the following:
  - key shortcut (either a Qt.Qt.Key or a QKeySequence)
  - event type on the read widget (a Qt.QEvent.Type)
  - signal from the read widget (a str representing a Signal signature)
- *exitEditTriggers* is a tuple containing one or more of the following:
  - key shortcut (either a Qt.Qt.Key or a QKeySequence)
  - event type on the write widget (a Qt.QEvent.Type)
  - signal from the write widget (a str representing a Signal signature)

#@todo: check integration with designer

enterEdit(**args*, ***kwargs*)
Slot for entering Edit mode

> **Note:** args and kwargs are ignored

enterEditTriggers = (16777265, 4)

eventFilter(*obj*, *event*)
reimplemented to intercept events from the read and write widgets

exitEdit(**args*, ***kwargs*)
Slot for entering Edit mode

> **Note:** args and kwargs are ignored

exitEditTriggers = (16777216, 9, 'applied')

classmethod getQtDesignerPluginInfo()

model

readWClass = None

setModel(*model*)
This implementation propagates the model to the read and write widgets. You may reimplement it to do things like passing different models to each.

setReadWidget(*widget*)
set the read Widget to be used. You can reimplement this method to tweak the read widget.

Parameters **widget** (QWidget) – This should be Taurus widget

setWriteWidget(*widget*)
set the write Widget to be used You can reimplement this method to tweak the write widget.

Parameters **widget** (QWidget) – This should be Taurus widget (typically a TaurusBaseWritableWidget)

writeWClass = None

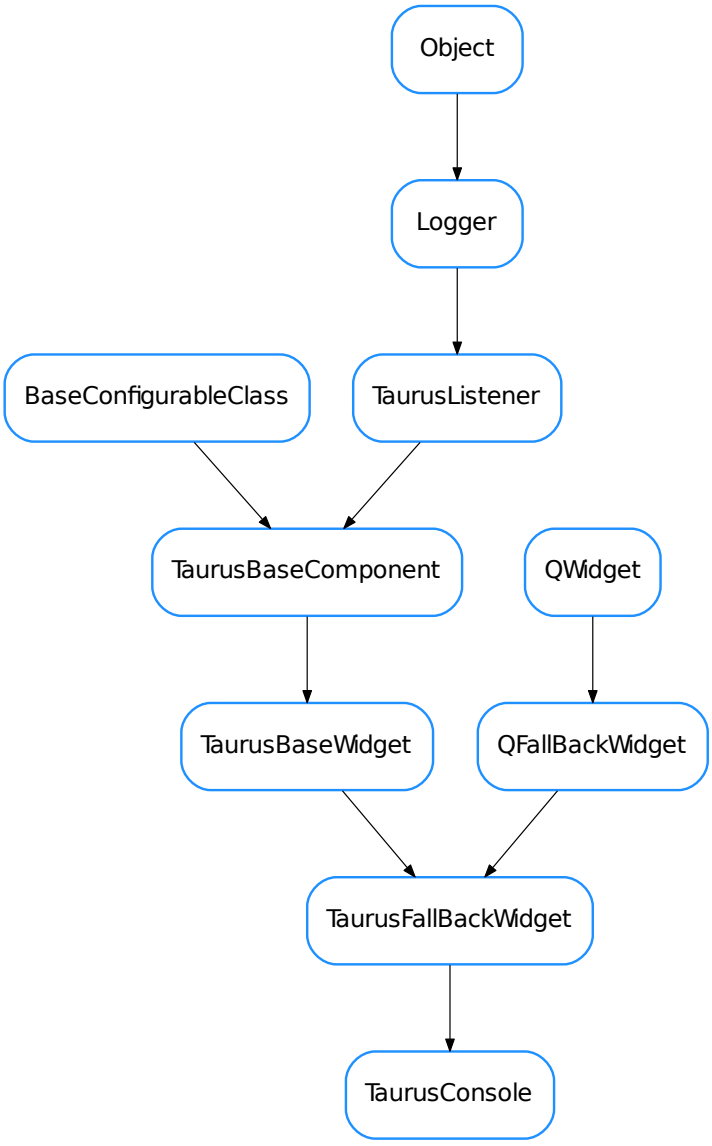
- *TaurusBoolRW*
- *TaurusLabelEditRW*
- *TaurusReadWriteSwitcher*

taurus.qt.qtgui.console

DEPRECATED since 4.0,4. Use *silx.gui.console*

Classes

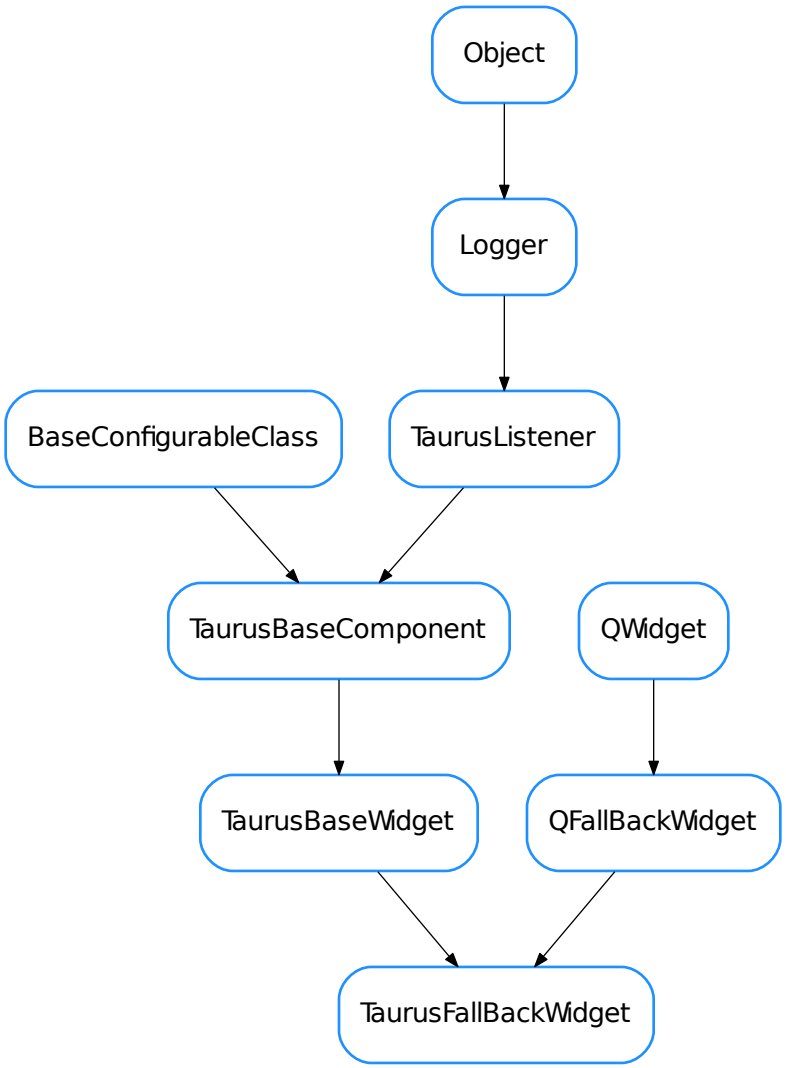
TaurusConsole



class **TaurusConsole**(*replaces=None*, *parent=None*, **args*, ***kwargs*)

Bases: `taurus.qt.qtgui.display.qfallback.TaurusFallBackWidget`

TaurusFallBackWidget



class **TaurusFallBackWidget**(*replaces=None*, *parent=None*, **args*, ***kwargs*)

Bases: `taurus.qt.qtgui.display.qfallback.QFallBackWidget`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget`

- *TaurusConsole*
- *TaurusFallBackWidget*

Functions

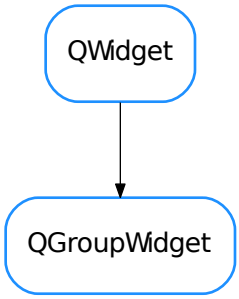
deprecated(**args*, ***kw*)

taurus.qt.qtgui.container

This package provides generic taurus container widgets

Classes

QGroupWidget



class **QGroupWidget** (*parent=None*, *designMode=False*)

Bases: PyQt4.QtGui.QWidget

An expandable/collapsible composite widget

DefaultContentStyle = {'stop_color': 'rgb(255, 255, 255)', 'border_color': 'rgb(0, 85, 227)', 'border_radius': '5px',

DefaultContentVisible = True

DefaultTitleBarHeight = 16

DefaultTitleBarStyle = {'font_color': 'white', 'stop_color': 'rgb(0, 65, 200)', 'border_radius': '5px', 'start_color':

DefaultTitleBarVisible = True

collapseButton()

Returns the collapse button widget

Return type QToolButton

Returns the collapse button widget

content()

Returns the contents widget

Return type QFrame

Returns the content widget

contentStyle

contentVisible

This property contains the widget's content's visibility

Access functions:

- *taurus.qt.qtgui.container.QGroupWidget.isContentVisible()*
- *taurus.qt.qtgui.container.QGroupWidget.setContentVisible()*
- *taurus.qt.qtgui.container.QGroupWidget.resetContentVisible()*

getContentStyle()
Returns this widget's content style

Return type dict

Returns this widget's content style

getContentStyleStr()
Returns this widget's content style

Return type dict

Returns this widget's content style

classmethod getQtDesignerPluginInfo()

getTitle()
Returns this widget's title

Return type str

Returns this widget's title

getTitleHeight()
Returns this widget's title height

Return type bool

Returns this widget's title height

getTitleIcon()
Returns this widget's title icon

Return type QIcon

Returns this widget's title icon

getTitleStyle()
Returns this widget's title style

Return type dict

Returns this widget's title style

getTitleStyleStr()
Returns this widget's title style

Return type dict

Returns this widget's title style

isContentVisible()
Returns this widget's contents visibility

Return type bool

Returns this widget's contents visibility

isTitleVisible()
Returns this widget's title visibility

Return type bool

Returns this widget's title visibility

resetContentStyle()
Resets this widget's content style

resetContentStyleStr()
Resets this widget's content style

resetContentVisible()
Resets this widget's contents visibility

resetTitleHeight()
Resets this widget's title height

resetTitleStyle()
Resets this widget's title style

resetTitleStyleStr()
Resets this widget's title style

resetTitleVisible()
Resets this widget's title visibility

setContentStyle(*style_map*)
Sets this widget's content style Used key/values for style_map: - 'start_color' : brush (Ex.: '#E0E0E0', 'rgb(0,0,0)', 'white') - 'stop_color' : brush (Ex.: '#E0E0E0', 'rgb(0,0,0)', 'white')

Parameters **style_map** (dict) – the new widget content style

setContentStyleStr(*style_map*)
Sets this widget's content style Used key/values for style_map: - 'start_color' : brush (Ex.: '#E0E0E0', 'rgb(0,0,0)', 'white') - 'stop_color' : brush (Ex.: '#E0E0E0', 'rgb(0,0,0)', 'white')

Parameters **style_map** (dict) – the new widget content style

setContentVisible(*show*)
Sets this widget's contents visibility

Parameters **show** (bool) – the new widget contents visibility

setTitle(*title*)
Sets this widget's title

Parameters **title** (str) – the new widget title

setTitleHeight(*h*)
Sets this widget's title height

Parameters **icon** (bool) – the new widget title height

setTitleIcon(*icon*)
Sets this widget's title icon

Parameters **icon** (QIcon) – the new widget title icon

setTitleStyle(*style_map*)
Sets this widget's title style Used key/values for style_map: - 'start_color' : brush (Ex.: '#E0E0E0', 'rgb(0,0,0)', 'white') - 'stop_color' : brush (Ex.: '#E0E0E0', 'rgb(0,0,0)', 'white') - 'font_color' : brush (Ex.: '#E0E0E0', 'rgb(0,0,0)', 'white') - 'border_radius': radius (Ex.: '5px', '5px,2px')

Parameters **style_map** (dict) – the new widget title style

setTitleStyleStr(*style_map*)
Sets this widget's title style Used key/values for style_map: - 'start_color' : brush (Ex.: '#E0E0E0',

'rgb(0,0,0)', 'white') - 'stop_color' : brush (Ex.: '#E0E0E0', 'rgb(0,0,0)', 'white') - 'font_color' : brush (Ex.: '#E0E0E0', 'rgb(0,0,0)', 'white') - 'border_radius': radius (Ex.: '5px', '5px,2px')

Parameters **style_map** (dict) – the new widget title style

setTitleVisible(*show*)
Sets this widget's title visibility

Parameters **icon** (bool) – the new widget title visibility

switchContentVisible()
Switches this widget's contents visibility

title
This property contains the widget's title

Access functions:

- *taurus.qt.qtgui.container.QGroupWidget.getTitle()*
- *taurus.qt.qtgui.container.QGroupWidget.setTitle()*

titleBar()
Returns the title bar widget

Return type QFrame

Returns the title bar widget

titleButton()
Returns the title button widget

Return type QToolButton

Returns the title button widget

titleHeight
This property contains the widget's title height

Access functions:

- *taurus.qt.qtgui.container.QGroupWidget.getTitleHeight()*
- *taurus.qt.qtgui.container.QGroupWidget.setTitleHeight()*
- *taurus.qt.qtgui.container.QGroupWidget.resetTitleHeight()*

titleIcon
This property contains the widget's title icon

Access functions:

- *taurus.qt.qtgui.container.QGroupWidget.getTitleIcon()*
- *taurus.qt.qtgui.container.QGroupWidget.setTitleIcon()*

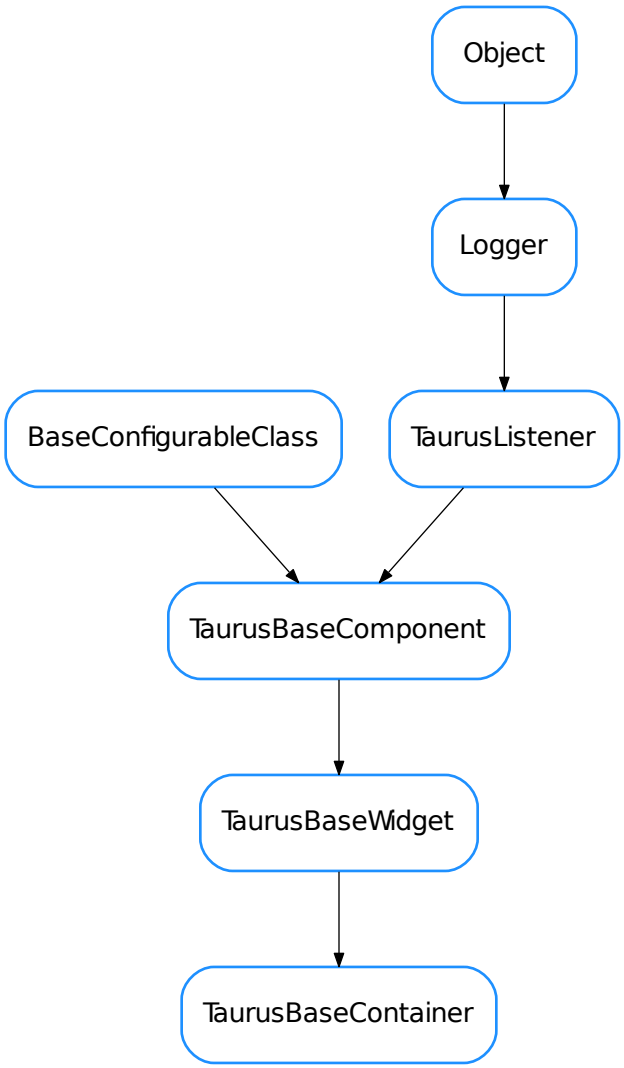
titleStyle

titleVisible
This property contains the widget's title visibility

Access functions:

- *taurus.qt.qtgui.container.QGroupWidget.isTitleVisible()*
- *taurus.qt.qtgui.container.QGroupWidget.setTitleVisible()*

TaurusBaseContainer



class TaurusBaseContainer (*name*, *parent=None*, *designMode=False*)

Bases: `taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget`

Base class for the Taurus container widgets. This type of taurus container classes are specially useful if you define a parent taurus model to them and set all contained taurus widgets to use parent model. Example:

```
from taurus.qt.qtgui.container import *
from taurus.qt.qtgui.display import *

widget = TaurusWidget()
layout = Qt.QVBoxLayout()
widget.setLayout(layout)
```

```
widget.model = 'sys/database/2'
stateWidget = TaurusLabel()
layout.addWidget(stateWidget)
stateWidget.useParentModel = True
stateWidget.model = '/state'
```

defineStyle()

getPendingOperations()

handleEvent(*evt_src*, *evt_type*, *evt_value*)

hasPendingOperations()

isReadOnly()

resetPendingOperations()

sizeHint()

taurusChildren(*objs=None*)

returns a list of all taurus children of this taurus container (recurses down skipping non-taurus widgets)

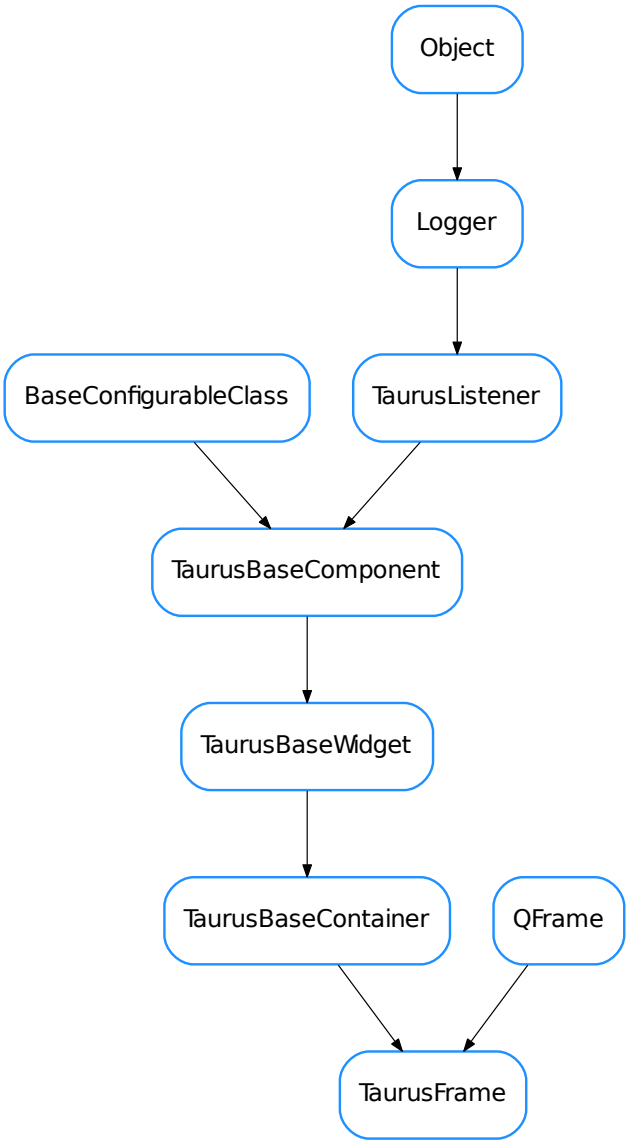
Parameters **objs** (list <objects>) – if given, the search starts at the objects passed in this list

Return type list <TaurusBaseWidget>

Returns

updateStyle()

TaurusFrame



class TaurusFrame (*parent=None*, *designMode=False*)

Bases: `PyQt4.QtGui.QFrame`, `taurus.qt.qtgui.container.taurusbasecontainer.TaurusBaseContainer`

This is a Qt.QFrame that additionally accepts a model property. This type of taurus container classes are specially useful if you define a parent taurus model to them and set all contained taurus widgets to use parent model. Example:

```
from taurus.qt.qtgui.container import *
from taurus.qt.qtgui.display import *

widget = TaurusFrame()
layout = Qt.QVBoxLayout()
widget.setLayout(layout)
widget.model = 'sys/database/2'
stateWidget = TaurusLabel()
layout.addWidget(stateWidget)
stateWidget.useParentModel = True
stateWidget.model = '/state'
```

applyPendingChanges

classmethod getQtDesignerPluginInfo()

model

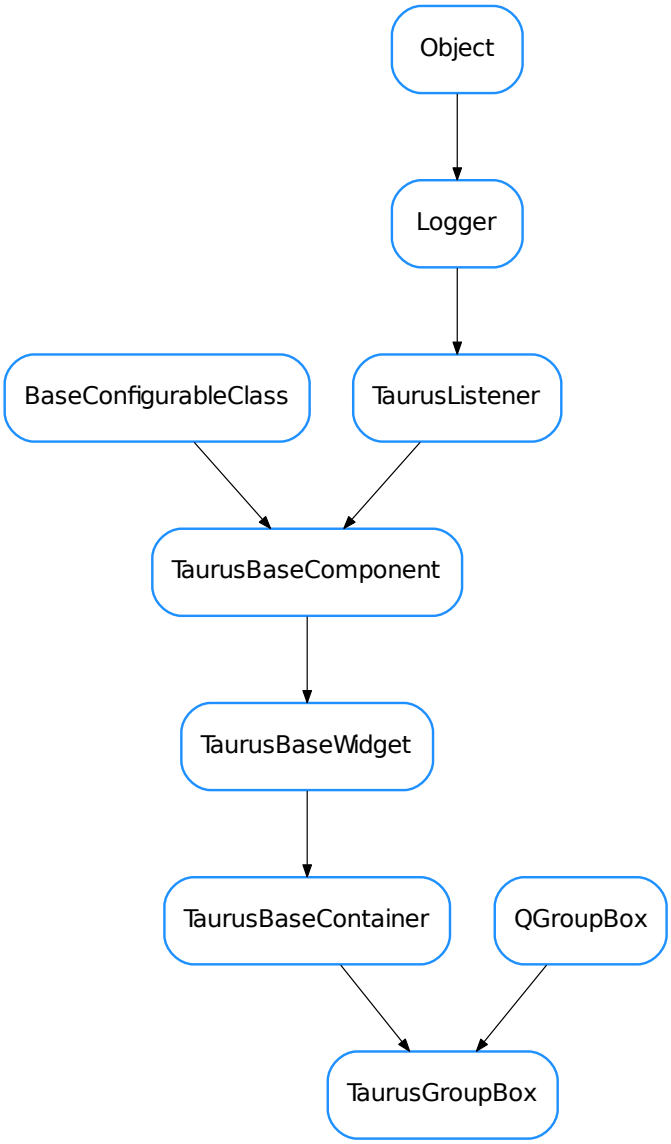
modelChanged

resetPendingChanges

showQuality

useParentModel

TaurusGroupBox



class TaurusGroupBox(*parent=None*, *designMode=False*)

Bases: `PyQt4.QtGui.QGroupBox`, `taurus.qt.qtgui.container.taurusbasecontainer.TaurusBaseContainer`

This is a Qt.QGroupBox that additionally accepts a model property. This type of taurus container classes are specially useful if you define a parent taurus model to them and set all contained taurus widgets to use parent model. Example:

```
from taurus.qt.qtgui.container import *
from taurus.qt.qtgui.display import *

widget = TaurusGroupBox("Example")
layout = Qt.QVBoxLayout()
widget.setLayout(layout)
widget.model = 'sys/database/2'
stateWidget = TaurusLabel()
layout.addWidget(stateWidget)
stateWidget.useParentModel = True
stateWidget.model = '/state'
```

applyPendingChanges

getDisplayValue()

getPrefixText()

classmethod **getQtDesignerPluginInfo**()

getSuffixText()

model

modelChanged

pendingOperationsChanged

prefixText

resetPendingChanges

setPrefixText

setSuffixText

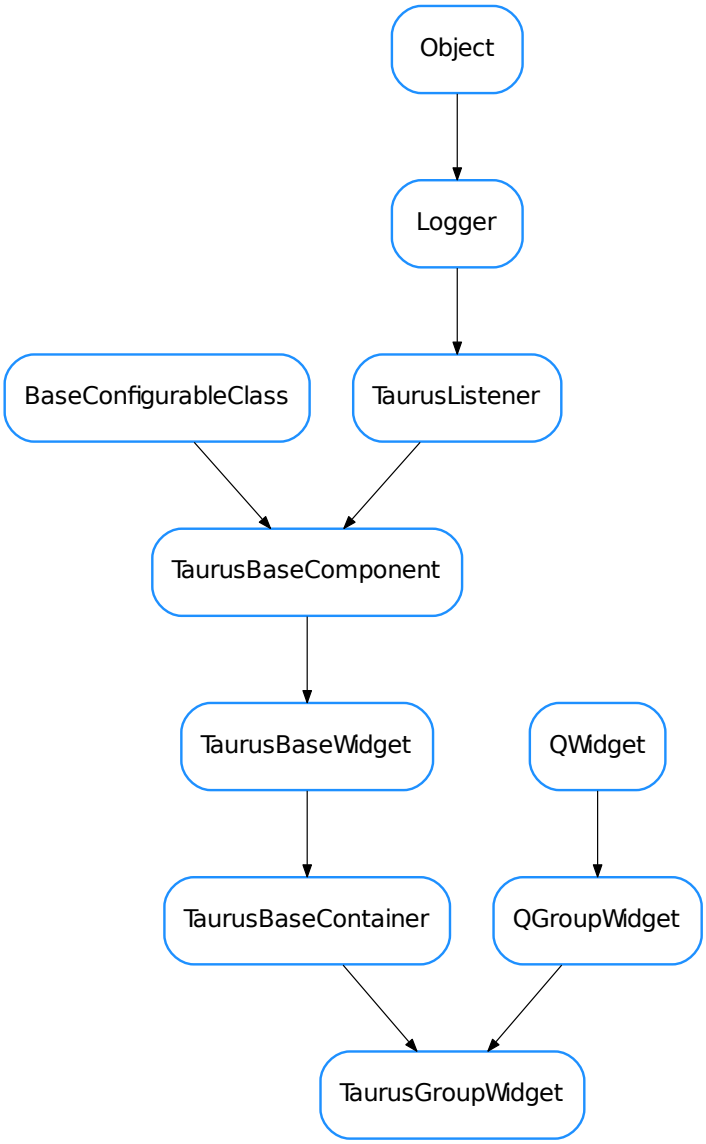
showQuality

showText

suffixText

useParentModel

TaurusGroupWidget



class TaurusGroupWidget (*parent=None*, *designMode=False*)

Bases: `taurus.qt.qtgui.container.qcontainer.QGroupWidget`, `taurus.qt.qtgui.container.taurusbasecontainer.TaurusBaseContainer`

This is a *taurus.qt.qtgui.container.QGroupWidget* that additionally accepts a model property. This type of taurus container classes are specially useful if you define a parent taurus model to them and set all contained taurus widgets to use parent model. Example:

```
from taurus.qt.qtgui.container import *
from taurus.qt.qtgui.display import *

widget = QGroupWidget(title="Example")
layout = Qt.QVBoxLayout()
widget.setLayout(layout)
widget.model = 'sys/database/2'
stateWidget = TaurusLabel()
layout.addWidget(stateWidget)
stateWidget.useParentModel = True
stateWidget.model = '/state'
```

applyPendingChanges

classmethod getQtDesignerPluginInfo()

model

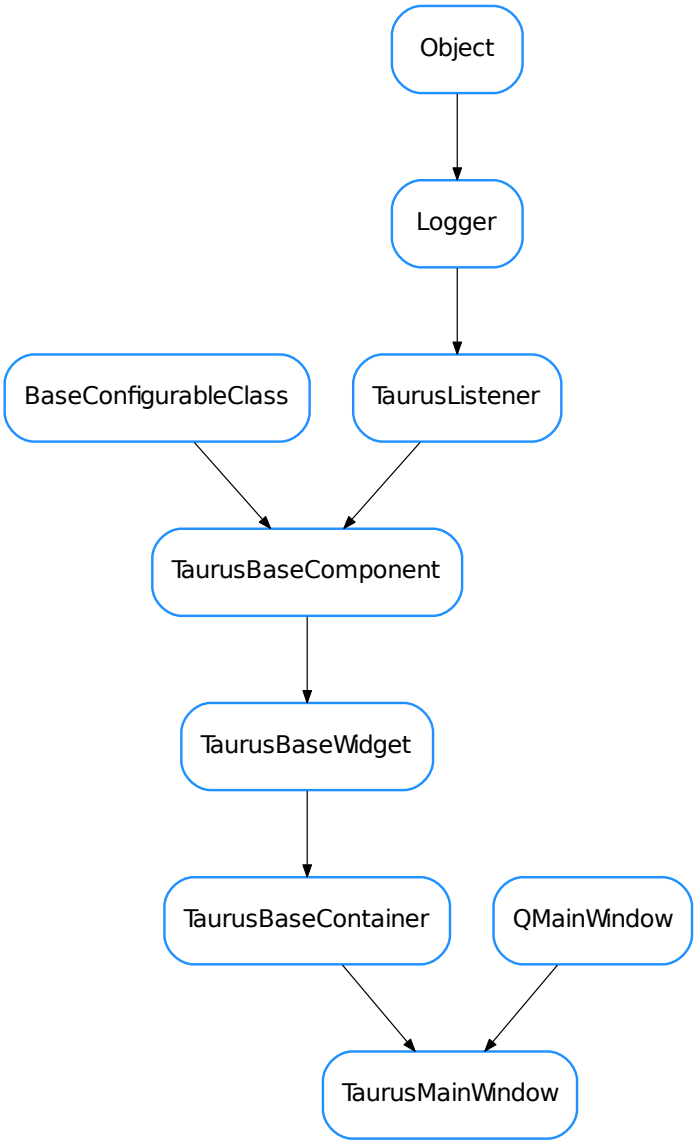
modelChanged

resetPendingChanges

showQuality

useParentModel

TaurusMainWindow



class TaurusMainWindow (*parent=None*, *designMode=False*, *splash=None*)

Bases: `PyQt4.QtGui.QMainWindow`, `taurus.qt.qtgui.container.taurusbasecontainer.TaurusBaseContainer`

A Taurus-aware QMainWindow with several customizations:

- It takes care of (re)storing its geometry and state (see *`loadSettings()`*)
- Supports perspectives (programmatic access and, optionally, accessible by user), and allows defining a set of "factory settings"

- It provides a customizable splashScreen (optional)
- Supports spawning remote consoles and remote debugging
- Supports full-screen mode toggling
- Supports adding launchers to external applications
- It provides a statusBar with an optional heart-beat LED
- **The following Menus are optionally provided and populated with basic actions:**
  - File (accessible by derived classes as *self.fileMenu*)
  - View (accessible by derived classes as *self.viewMenu*)
  - Taurus (accessible by derived classes as *self.taurusMenu*)
  - Tools (accessible by derived classes as *self.toolsMenu*)
  - Help (accessible by derived classes as *self.helpMenu*)

addExternalAppLauncher(*extapp*, *toToolBar=True*, *toMenu=True*)
Adds launchers for an external application to the Tools Menu and/or to the Tools ToolBar.

Parameters

- **extapp** (ExternalAppAction or list <str>) – the external application to be launched passed as a ExternalAppAction (recommended because it allows to specify custom text and icon) or, alternatively, as a list of strings (sys.argv- like) that will be passed to subprocess.Popen().
- **toToolBar** (bool) – If True (default) a button will be added in the Tools toolBar
- **toMenu** (bool) – If True (default) an entry will be added in the Tools Menu, under the "External Applications" submenu

See also:

ExternalAppAction

addLoggerWidget(*hidden=True*)
adds a QLoggingWidget as a dockwidget of the main window (and hides it by default)

applyPendingChanges

basicTaurusToolbar()
returns a QToolBar with few basic buttons (most important, the logo)

Return type QToolBar

Returns

checkSingleInstance(*key=None*)
Tries to connect via a QLocalSocket to an existing application with the given key. If another instance already exists (i.e. the connection succeeds), it means that this application is not the only one

closeEvent(*event*)
This event handler receives widget close events

createFileMenu()
adds a "File" Menu

createHelpMenu()
adds a "Help" Menu

createPerspectivesToolBar()
adds a Perspectives ToolBar

createTaurusMenu()
adds a "Taurus" Menu

createToolsMenu()
adds a "Tools" Menu

createViewMenu()
adds a "View" Menu

deleteExternalAppLauncher(*action*)
Remove launchers for an external application to the Tools Menu and/or to the Tools ToolBar.

Parameters **extapp** (ExternalAppAction) – the external application to be removed passed as a ExternalAppAction

exportSettingsFile(*fname=None*)
copies the current settings file into the given file name.

Parameters **fname** (str) – name of output file. If None given, a file dialog will be shown.

getFactorySettingsFileName()
returns the file name of the "factory settings" (the ini file with default settings). The default implementation returns "<path>/<appname>.ini", where <path> is the path of the module where the main window class is defined and <appname> is the application name (as obtained from QApplication).

Return type str

Returns the absolute file name.

getHeartbeat()
returns the heart beat interval

getHelpManualURI()

getPerspectivesList(*settings=None*)
Returns the list of saved perspectives

Parameters **settings** (QSettings or None) – a QSettings object. If None given, the default one returned by *getQSettings()* will be used

Return type QStringList

Returns the list of the names of the currently saved perspectives

getQSettings()
Returns the main window settings object. If it was not previously set, it will create a new QSettings object following the Taurus convention i.e., it using Ini format and userScope)

Return type QSettings

Returns the main window QSettings object

classmethod getQtDesignerPluginInfo()

getTangoHost()

heartbeat

helpManualURI

importSettingsFile(*fname=None*)
loads settings (including importing all perspectives) from a given ini file. It warns before overwriting an existing perspective.

Parameters **fname** (str) – name of ini file. If None given, a file dialog will be shown.

loadPerspective(*name=None*, *settings=None*)
Loads the settings saved for the given perspective. It emits a 'perspectiveChanged' signal with name as its parameter

Parameters

- **name** (str) – name of the perspective
- **settings** (QSettings or None) – a QSettings object. If None given, the default one returned by *getQSettings()* will be used

loadSettings(*settings=None*, *group=None*, *ignoreGeometry=False*, *factorySettingsFileName=None*)
restores the application settings previously saved with *saveSettings()*.

Note: This method should be called explicitly from derived classes after all initialization is done

Parameters

- **settings** (QSettings or None) – a QSettings object. If None given, the default one returned by *getQSettings()* will be used
- **group** (str) – a prefix that will be added to the keys to be loaded (no prefix by default)
- **ignoreGeometry** (bool) – if True, the geometry of the MainWindow won't be restored
- **factorySettingsFileName** (str) – file name of a ini file containing the default settings to be used as a fallback in case the settings file is not found (e.g., the first time the application is launched after installation)

model

modelChanged

newQSettings()
Returns a settings taurus-specific QSettings object. The returned QSettings object will comply with the Taurus defaults for storing application settings (i.e., it uses Ini format and userScope)

Return type QSettings

Returns a taurus-specific QSettings object

onIncommingSocketConnection()
Slot to be called when another application/instance with the same key checks if this application exists.

Note: This is a dummy implementation which just logs the connection and discards the associated socket
You may want to reimplement this if you want to act on such connections

onShowManual(*anchor=None*)
Shows the User Manual in a dockwidget

perspectiveChanged

removePerspective(*name=None*, *settings=None*)
removes the given perspective from the settings

Parameters

- **name** (str) – name of the perspective
- **settings** (QSettings or None) – a QSettings object. If None given, the default one returned by *getQSettings()* will be used

resetHeartbeat()
resets the heartbeat interval

resetHelpManualURI()

resetPendingChanges

resetQSettings()
equivalent to setQSettings(None)

resetTangoHost()

savePerspective(*name=None*)
Stores current state of the application as a perspective with the given name

Parameters **name** (str) – name of the perspective

saveSettings(*group=None*)
saves the application settings (so that they can be restored with *loadSettings()*)

Note: this method is automatically called by default when closing the window, so in general there is no need to call it from derived classes

---

Parameters **group** (str) – a prefix that will be added to the keys to be saved (no prefix by default)

setHeartbeat(*interval*)
sets the interval of the heartbeat LED for the window. The heartbeat is displayed by a Led in the status bar unless it is disabled by setting the interval to 0

Parameters **interval** (int) – heart beat interval in millisecs. Set to 0 to disable

setHelpManualURI(*uri*)

setQSettings(*settings*)
sets the main window settings object

Parameters **settings** (QSettings or None) –

See also:

getQSettings()

setTangoHost(*host*)

showEvent(*event*)
This event handler receives widget show events

showHelpAbout()

showQuality

splashScreen()
returns a the splashScreen

Return type QSplashScreen

Returns

tangoHost

updatePerspectivesMenu()
re-checks the perspectives available to update self.perspectivesMenu

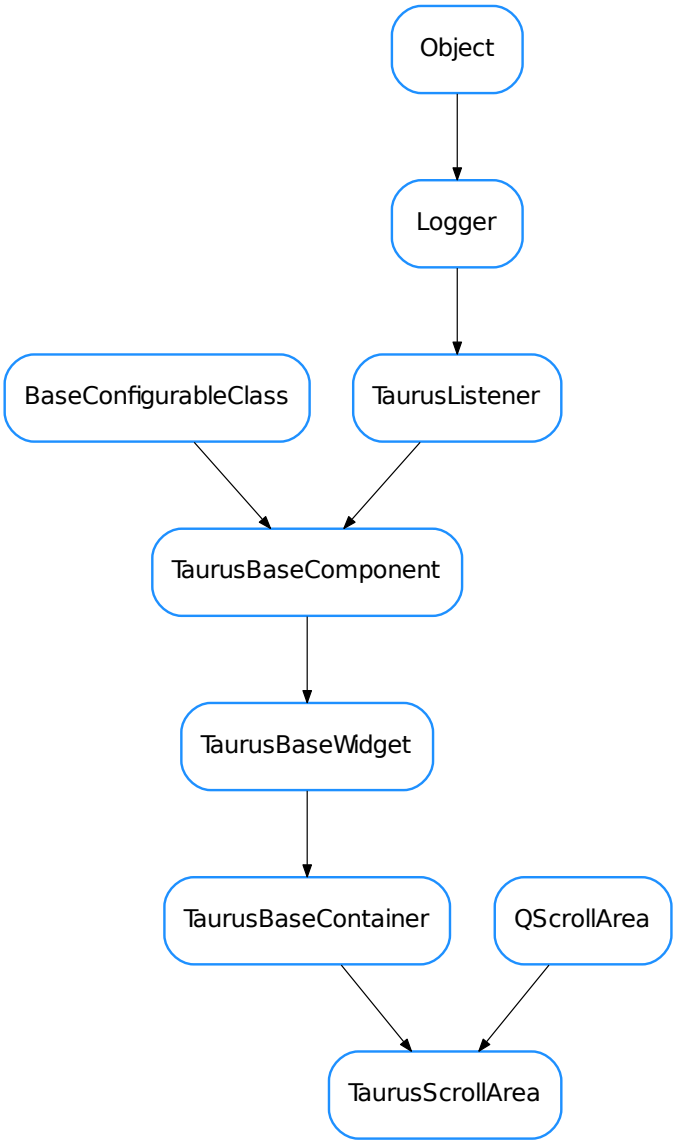
Note: This method may need be called by derived classes at the end of their initialization.

Return type QMenu

Returns the updated perspectives menu (or None if self._supportUserPerspectives is False)

useParentModel

TaurusScrollArea



class **TaurusScrollArea** (*parent=None*, *designMode=False*)

Bases: `PyQt4.QtGui.QScrollArea`, `taurus.qt.qtgui.container.taurusbasecontainer.TaurusBaseContainer`

This is a Qt.QScrollArea that additionally accepts a model property. This type of taurus container classes are specially useful if you define a parent taurus model to them and set all contained taurus widgets to use parent model. Example:

```
from taurus.qt.qtgui.container import *
from taurus.qt.qtgui.display import *

widget = TaurusScrollArea()
layout = Qt.QVBoxLayout()
widget.setLayout(layout)
widget.model = 'sys/database/2'
stateWidget = TaurusLabel()
layout.addWidget(stateWidget)
stateWidget.useParentModel = True
stateWidget.model = '/state'
```

applyPendingChanges

getPendingOperations()

classmethod **getQtDesignerPluginInfo**()

hasPendingOperations()

model

modelChanged

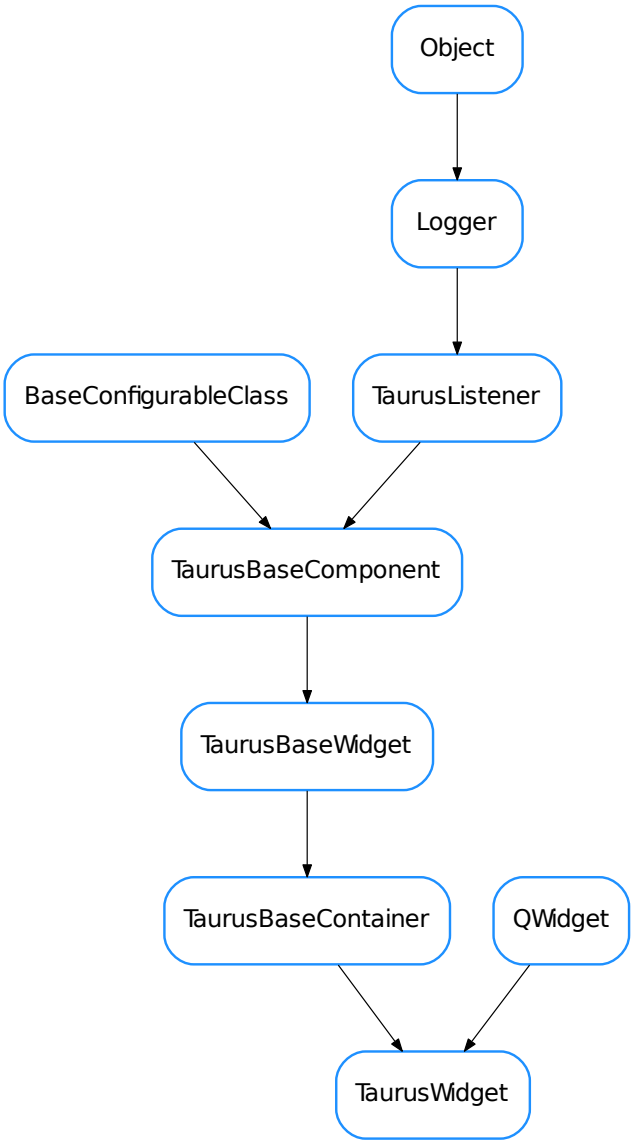
resetPendingChanges

resetPendingOperations()

showQuality

useParentModel

TaurusWidget



class TaurusWidget (*parent=None*, *designMode=False*)

Bases: `PyQt4.QtGui.QWidget`, `taurus.qt.qtgui.container.taurusbasecontainer.TaurusBaseContainer`

This is a Qt.QWidget that additionally accepts a model property. This type of taurus container classes are specially useful if you define a parent taurus model to them and set all contained taurus widgets to use parent model. Example:

```
from taurus.qt.qtgui.container import *
from taurus.qt.qtgui.display import *

widget = TaurusWidget()
layout = Qt.QVBoxLayout()
widget.setLayout(layout)
widget.model = 'sys/database/2'
stateWidget = TaurusLabel()
layout.addWidget(stateWidget)
stateWidget.useParentModel = True
stateWidget.model = '/state'
```

applyPendingChanges

classmethod getQtDesignerPluginInfo()

model

modifiableByUser

resetPendingChanges

showQuality

useParentModel

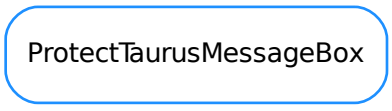
- *QGroupWidget*
- *TaurusBaseContainer*
- *TaurusFrame*
- *TaurusGroupBox*
- *TaurusGroupWidget*
- *TaurusMainWindow*
- *TaurusScrollArea*
- *TaurusWidget*

taurus.qt.qtgui.dialog

This package contains a collection of taurus Qt widgets representing various panels like forms or panels to be inserted in dialogs

Classes

ProtectTaurusMessageBox



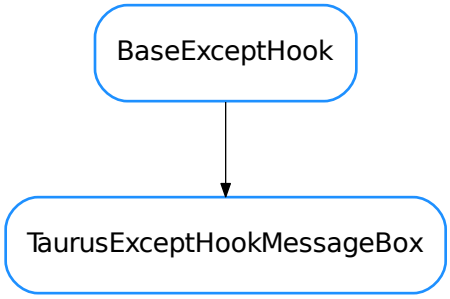
class ProtectTaurusMessageBox (*title=None*, *msg=None*)

Bases: object

The idea of this class is to be used as a decorator on any method you which to protect against exceptions. The handle of the exception is to display a *TaurusMessageBox* with the exception information. The optional parameter title gives the window bar a customized title. The optional parameter msg allows you to give a customized message in the dialog. Example:

```
@ProtectTaurusMessgeBox(title="Error trying to turn the beam on")
def turnBeamOn(device_name):
    d = taurus.Device(device_name)
    d.TurnOn()
```

TaurusExceptHookMessageBox



class TaurusExceptHookMessageBox (*hook_to=None*, *title=None*, *msg=None*)

Bases: taurus.core.util.excepthook.BaseExceptHook

A callable class that acts as an excepthook that displays an unhandled exception in a *TaurusMessageBox*.

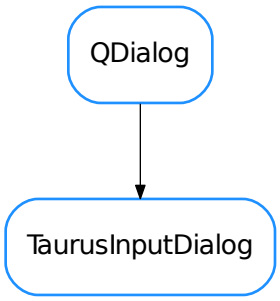
Parameters

- **hook_to** (*callable*) – callable excepthook that will be called at the end of this hook handling [default: None]
- **title** – message box title [default: None meaning use exception value]
- **msg** – message box text [default: None meaning use exception]

MSG_BOX = None

report (**exc_info*)

TaurusInputDialog



class **TaurusInputDialog**(*input_data=None*, *parent=None*, *input_panel_klass=None*, *designMode=False*)

Bases: `PyQt4.QtGui.QDialog`

The TaurusInputDialog class provides a simple convenience dialog to get a single value from the user.

panel()

Returns the *taurus.qt.qtgui.panel.TaurusInputPanel*.

Returns the internal panel

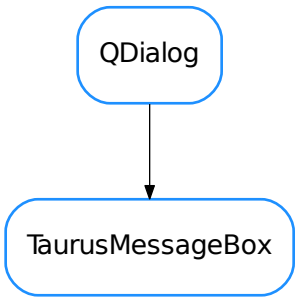
Return type *taurus.qt.qtgui.panel.TaurusInputPanel*

value()

Returns the value selected by the user.

Returns the value selected by the user

TaurusMessageBox



class **TaurusMessageBox**(*err_type=None*, *err_value=None*, *err_traceback=None*, *parent=None*, *designMode=False*)

Bases: PyQt4.QtGui.QDialog

A panel intended to display a taurus error. Example:

```
dev = taurus.Device("sys/tg_test/1")
try:
    print dev.read_attribute("throw_exception")
except PyTango.DevFailed, df:
    msgbox = TaurusMessageBox()
    msgbox.show()
```

You can show the error outside the exception handling code. If you do this, you should keep a record of the exception information as given by sys.exc_info():

```
dev = taurus.Device("sys/tg_test/1")
exc_info = None
try:
    print dev.read_attribute("throw_exception")
except PyTango.DevFailed, df:
    exc_info = sys.exc_info()

if exc_info:
    msgbox = TaurusMessageBox(*exc_info)
    msgbox.show()
```

addButton(*button*, *role=3*)

Adds the given button with the given to the button box

Parameters

- **button** (*PyQt4.QtGui.QPushButton*) – the button to be added
- **role** (*PyQt4.Qt.QDialogButtonBox.ButtonRole*) – button role

getText()

Returns the current text of this panel

Returns the text for this panel

Return type str

panel()

Returns the *taurus.qt.qtgui.panel.TaurusMessagePanel*.

Returns the internal panel

Return type *taurus.qt.qtgui.panel.TaurusMessagePanel*

setDetailedText(*text*)

Sets the detailed text of the dialog

Parameters **text** (*str*) – the new text

setError(*err_type=None*, *err_value=None*, *err_traceback=None*)

Sets the exception object. Example usage:

```
dev = taurus.Device("sys/tg_test/1")
exc_info = None
msgbox = TaurusMessageBox()
try:
    print dev.read_attribute("throw_exception")
```

```
except PyTango.DevFailed, df:
    exc_info = sys.exc_info()

if exc_info:
    msgbox.setError(*exc_info)
    msgbox.show()
```

Parameters

- **err_type** – the exception type of the exception being handled (a class object)
- **err_value** (*object*) – exception object
- **err_traceback** (*TracebackType*) – a traceback object which encapsulates the call stack at the point where the exception originally occurred

setIconPixmap(*pixmap*)
Sets the icon to the dialog

Parameters **pixmap** (*PyQt4.Qt.QPixmap*) – the icon pixmap

setText(*text*)
Sets the text of the dialog

Parameters **text** (*str*) – the new text

- *ProtectTaurusMessageBox*
- *TaurusExceptHookMessageBox*
- *TaurusInputDialog*
- *TaurusMessageBox*

Functions

get_input(*input_data*, *parent=None*, *input_panel_klass=None*)
Static convenience function to get an input from the user using a dialog. The dialog will be modal.

The input_data is a dictionary which contains information on how to build the input dialog. It **must** contains the following keys:

•*prompt* <str>: message to be displayed

The following are optional keys (and their corresponding default values):

•*title* <str> (doesn't have default value)

•*key* <str> (doesn't have default value): a label to be presented left to the input box represeting the label

•*unit* <str> (doesn't have default value): a label to be presented right to the input box representing the units

•*data_type* <str or sequence> ('String'): type of data to be requested. Standard accepted data types are 'String', 'Integer', 'Float', 'Boolean', 'Text'. A list of elements will be interpreted as a selection. Default TaurusInputPanel class will interpret any custom data types as 'String' and will display input widget accordingly. Custom data types can be handled differently by supplying a different input_panel_klass.

•*minimum* <int/float> (-sys.maxint): minimum value (makes sence when data_type is 'Integer' or 'Float')

•*maximum* <int/float> (sys.maxint): maximum value (makes sence when data_type is 'Integer' or 'Float')

•*step* <int/float> (1): step size value (makes sence when data_type is 'Integer' or 'Float')

•*decimals* <int> (1): number of decimal places to show (makes sence when data_type is 'Float')

•*default_value* <obj> (doesn't have default value): default value

•*allow_multiple* <bool> (False): allow more than one value to be selected (makes sence when data_type is a sequence of possibilities)

Parameters

- **input_data** (dict) – a dictionary with information on how to build the input dialog
- **parent** (*PyQt4.QtGui.QWidget*) – parent widget
- **input_panel_klass** (*TaurusInputPanel*) – python class to be used as input panel [default: *TaurusInputPanel*]

Returns a tuple containing value selected and boolean which is true if user accepted the dialog (pressed Ok) or false otherwise

Return type tuple< obj, bool >

Examples:

```
d1 = dict(prompt="What's your name?", data_type="String")
d2 = dict(prompt="What's your age?", data_type="Integer",
          default_value=4, maximum=100, key="Age", unit="years")
d3 = dict(prompt="What's your favourite number?", data_type="Float",
          default_value=0.1, maximum=88.8, key="Number")
d4 = dict(prompt="What's your favourite car brand?",
          data_type=["Mazda", "Skoda", "Citroen", "Mercedes", "Audi", "Ferrari"],
          default_value="Mercedes")
d5 = dict(prompt="Select some car brands", allow_multiple=True,
          data_type=["Mazda", "Skoda", "Citroen", "Mercedes", "Audi", "Ferrari"],
          default_value=["Mercedes", "Citroen"])
d6 = dict(prompt="What's your favourite color?", key="Color",
          data_type=["blue", "red", "green"], default_value="red")
d7 = dict(prompt="Do you like bears?",
          data_type='Boolean', key="Yes/No", default_value=True)
d8 = dict(prompt="Please write your memo",
          data_type='Text', key="Memo", default_value="By default a memo is a␣
↪long thing")
for d in [d1, d2, d3, d4, d5, d6, d7, d8]:
    get_input(input_data=d, title=d['prompt'])
```

protectTaurusMessageBox(*fn*)

The idea of this function is to be used as a decorator on any method you which to protect against exceptions. The handle of the exception is to display a *TaurusMessageBox* with the exception information. Example:

```
@protectTaurusMessgeBox
def turnBeamOn(device_name):
    d = taurus.Device(device_name)
    d.TurnOn()
```

taurus.qt.qtgui.display

This package contains a collection of taurus widgets designed to display taurus information, typically in a read-only fashion (no user interaction is possible). Examples of widgets that suite this rule are labels, leds and LCDs

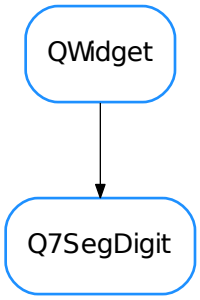
Modules

taurus.qt.qtgui.display.demo

This package contains a collection of taurus display widgets demos

Classes

Q7SegDigit



class **Q7SegDigit** (*parent=None*, ***kwargs*)

Bases: PyQt4.QtGui.QWidget

A widget representing a single seven segment digit. The style can be configured through the widget properties. For example, a typical LCD would have the following style:

- bgColor 170, 170, 127
- ledOnPenColor 30,30,30
- ledOnBgColor 0,0,0
- ledOffPenColor 160, 160, 120
- ledOffbgColor 150, 150, 112

DftAspectRatio = 1

DftBgBrush

DftHeight = 300

DftLedOffBgColor

DftLedOffPenColor

DftLedOnBgColor = 7

DftLedOnPenColor

DftLedPenWidth = 5

DftUseFrame = True

DftValue = ''

DftWidth = 300

LedGeometriesWithFrame300x300 = (<PyQt4.QtGui.QPolygonF object>, <PyQt4.QtGui.QPolygonF object>, <PyQ

LedGeometriesWithFrame300x400 = (<PyQt4.QtGui.QPolygonF object>, <PyQt4.QtGui.QPolygonF object>, <PyQ

LedGeometriesWithoutFrame300x300 = (<PyQt4.QtGui.QPolygonF object>, <PyQt4.QtGui.QPolygonF object>, <

LedGeometriesWithoutFrame300x400 = (<PyQt4.QtGui.QPolygonF object>, <PyQt4.QtGui.QPolygonF object>, <

Leds = ((1, 1, 1, 0, 1, 1, 1), (0, 0, 1, 0, 0, 1, 0), (1, 0, 1, 1, 1, 0, 1), (1, 0, 1, 1, 0, 1, 1), (0, 1, 1, 1, 0, 1, 0), (1, 1, 0, 1, 0, 1, 1), (1, 1,

bgBrush
This property holds the background brush

Access functions:

- *taurus.qt.qtgui.display.Q7SegDigit.getBgBrush()*
- *taurus.qt.qtgui.display.Q7SegDigit.setBgBrush()*
- *taurus.qt.qtgui.display.Q7SegDigit.resetBgBrush()*

getAspectRatio()

getBgBrush()

getLedOffBgColor()

getLedOffPenColor()

getLedOnBgColor()

getLedOnPenColor()

getLedPenWidth()

getUseFrame()

getValue()

ledOffBgColor
This property holds the led background color when the led is light OFF

Access functions:

- *taurus.qt.qtgui.display.Q7SegDigit.getLedOffBgColor()*
- *taurus.qt.qtgui.display.Q7SegDigit.setLedOffBgColor()*
- *taurus.qt.qtgui.display.Q7SegDigit.resetLedOffBgColor()*

ledOffPenColor
This property holds the led pen color when the led is light OFF

Access functions:

- *taurus.qt.qtgui.display.Q7SegDigit.getLedOffPenColor()*
- *taurus.qt.qtgui.display.Q7SegDigit.setLedOffPenColor()*
- *taurus.qt.qtgui.display.Q7SegDigit.resetLedOffPenColor()*

ledOnBgColor
This property holds the led background color when the led is light ON

Access functions:

- *taurus.qt.qtgui.display.Q7SegDigit.getLedOnBgColor()*
- *taurus.qt.qtgui.display.Q7SegDigit.setLedOnBgColor()*
- *taurus.qt.qtgui.display.Q7SegDigit.resetLedOnBgColor()*

ledOnPenColor
This property holds the led pen color when the led is light ON

Access functions:

- *taurus.qt.qtgui.display.Q7SegDigit.getLedOnPenColor()*
- *taurus.qt.qtgui.display.Q7SegDigit.setLedOnPenColor()*
- *taurus.qt.qtgui.display.Q7SegDigit.resetLedOnPenColor()*

ledPenWidth
This property holds the pen width

Access functions:

- *taurus.qt.qtgui.display.Q7SegDigit.getLedPenWidth()*
- *taurus.qt.qtgui.display.Q7SegDigit.setLedPenWidth()*
- *taurus.qt.qtgui.display.Q7SegDigit.resetLedPenWidth()*

minimumSizeHint()

paintEvent(*evt*)

resetAspectRatio()

resetBgBrush()

resetLedOffBgColor()

resetLedOffPenColor()

resetLedOnBgColor()

resetLedOnPenColor()

resetLedPenWidth()

resetUseFrame()

resetValue()

setAspectRatio(*apectRatio*)

setBgBrush(*bgBrush*)

setLedOffBgColor(*bgColor*)

setLedOffPenColor(*penColor*)

setLedOnBgColor(*bgColor*)

setLedOnPenColor(*penColor*)

setLedPenWidth(*w*)

setUseFrame(*useFrame*)

setValue(*n*)

sizeHint()

useFrame
This property holds wheater of not to draw a frame

Access functions:

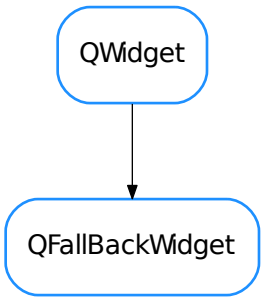
- *taurus.qt.qtgui.display.Q7SegDigit.getUseFrame()*
- *taurus.qt.qtgui.display.Q7SegDigit.setUseFrame()*
- *taurus.qt.qtgui.display.Q7SegDigit.resetUseFrame()*

value
This property holds the widget value

Access functions:

- *taurus.qt.qtgui.display.Q7SegDigit.getValue()*
- *taurus.qt.qtgui.display.Q7SegDigit.setValue()*
- *taurus.qt.qtgui.display.Q7SegDigit.resetValue()*

QFallBackWidget

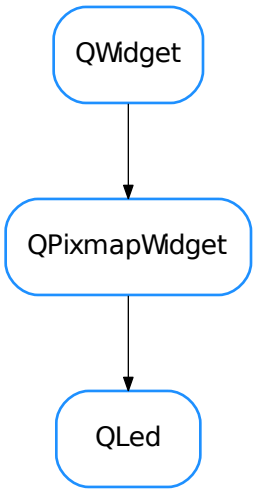


class QFallBackWidget(*replaces=None*, *parent=None*, **args*, ***kwargs*)
Bases: PyQt4.QtGui.QWidget

A FallBack widget to be used when a real widget cannot be loaded for any reason (example: a dependency library is not installed)

onShowDetails()

QLed



class QLed (*parent=None*, *designMode=False*)

Bases: taurus.qt.qtgui.display.qpixmapwidget.QPixmapWidget

A Led

DefaultBlinkingInterval = 0

DefaultLedColor = 'green'

DefaultLedInverted = False

DefaultLedPattern = 'leds_images256:led_{color}_{status}.png'

DefaultLedStatus = True

blinkingInterval

This property holds the blinking interval in millisecs. 0 means no blinking

Access functions:

- *QLed.getBlinkingInterval()*
- *QLed.setBlinkingInterval()*
- *QLed.resetBlinkingInterval()*

getBlinkingInterval()

returns the blinking interval

Return type int

Returns blinking interval or 0 if blinking is not enabled.

getLedColor()

Returns the led color :return: led color :rtype: str

getLedInverted()
Returns if the led is inverted. :return: inverted mode :rtype: bool

getLedPatternName()
Returns the current led pattern name :return: led pattern name :rtype: str

getLedStatus()
Returns the led status :return: led status :rtype: bool

classmethod getQtDesignerPluginInfo()

isLedColorValid(*name*)
Determines if the given color name is valid. :param color: the color :type color: str :return: True is the given color name is valid or False otherwise :rtype: bool

ledColor
This property holds the led color

Access functions:

- *QLed.getLedColor()*
- *QLed.setLedColor()*
- *QLed.resetLedColor()*

ledInverted
This property holds the led inverted: False means do not invert the

Access functions:

- *QLed.getLedInverted()*
- *QLed.setLedInverted()*
- *QLed.resetLedInverted()*

ledPattern
This property holds the led pattern name

Access functions:

- *QLed.getLedPatternName()*
- *QLed.setLedPatternName()*
- *QLed.resetLedPatternName()*

ledStatus
This property holds the led status: False means OFF, True means ON

Access functions:

- *QLed.getLedStatus()*
- *QLed.setLedStatus()*
- *QLed.resetLedStatus()*

minimumSizeHint()
Overwrite the default minimum size hint (0,0) to be (16,16) :return: the minimum size hint 16,16 :rtype: PyQt4.Qt.QSize

resetBlinkingInterval()
resets the blinking interval

resetLedColor()
Resets the led color

resetLedInverted()
Resets the led inverted mode

resetLedPatternName()
Resets the led pattern to **:leds/images256/led_{color}_{status}.png**.

resetLedStatus()
Resets the led status

setBlinkingInterval(*interval*)
sets the blinking interval (the time between status switching). Set to a nonpositive number for disabling blinking

Parameters **interval** (int) – the blinking interval in millisecs. Set to 0 for disabling blinking

setLedColor(*color*)
Sets the led color :param status: the new color :type status: str

setLedInverted(*inverted*)
Sets the led inverted mode :param status: the new inverted mode :type status: bool

setLedPatternName(*name*)
Sets the led pattern name. Should be a string containing a path to valid images. The string can contain the keywords:

1. {status} - transformed to 'on' of 'off' according to the status
2. {color} - transformed to the current led color

Example: **:leds/images256/led_{color}_{status}.png** will be transformed to **:leds/images256/led_red_on.png** when the led status is True and the led color is red.

Parameters **name** (*str*) – new pattern

setLedStatus(*status*)
Sets the led status :param status: the new status :type status: bool

sizeHint()

toLedName(*status=None*, *color=None*, *inverted=None*)
Gives the led name for the given status and color. If status or color are not given, the current led status or color are used.

Parameters

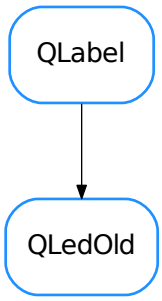
- **status** (*bool*) – the status
- **color** (*str*) – the color

Returns string containing the led name

Return type str

toggleLedStatus()
toggles the current status of the led

QLedOld



class **QLedOld**(*parent=None*, *ledsize=24*, *ledcolor=1*)

Bases: PyQt4.QtGui.QLabel

changeColor(*ledColor*)

changeSize(*ledSize*)

ledDirPattern = ':leds/images%(size)d'

off()

on()

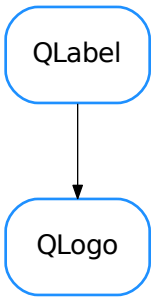
retranslateUi(*Led*)

toCompleteLedName(*size*, *status*, *color*)

toLedName(*status*, *color*)

tr(*string*)

QLogo



class `QLogo` (*parent=None*, *designMode=False*)

Bases: `PyQt4.QtGui.QLabel`

classmethod `getQtDesignerPluginInfo` ()

Returns pertinent information in order to be able to build a valid QtDesigner widget plugin.

The dictionary returned by this method should contain *at least* the following keys and values:

- 'module' : a string representing the full python module name (ex.: 'taurus.qt.qtgui.base')
- 'icon' : a string representing valid resource icon (ex.: 'designer:combobox.png')
- 'container' : a bool telling if this widget is a container widget or not.

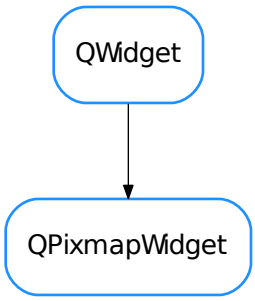
This default implementation returns the following dictionary:

{ 'module' ['taurus.qt.qtgui.base',] 'group' : 'Taurus Widgets', 'icon' : 'logos:taurus.svg', 'container' : False }

Return type `dict`

Returns a map with pertinent designer information

QPixmapWidget



class QPixmapWidget (*parent=None*, *designMode=False*)

Bases: PyQt4.QtGui.QWidget

This widget displays an image (pixmap). By default the pixmap is scaled to the widget size and the aspect ratio is kept. The default alignment of the pixmap inside the widget space is horizontal left, vertical center.

DefaultAlignment = 129

DefaultAspectRatioMode = 1

DefaultTransformationMode = 1

alignment

This property holds the widget's pixmap alignment

Access functions:

- *QPixmapWidget.getAlignment()*
- *QPixmapWidget.setAlignment()*
- *QPixmapWidget.resetAlignment()*

aspectRatioMode

This property holds the widget's pixmap aspect ratio mode

Access functions:

- *QPixmapWidget.getAspectRatioMode()*
- *QPixmapWidget.setAspectRatioMode()*
- *QPixmapWidget.resetAspectRatioMode()*

getAlignment()

Returns the alignment to apply when drawing the pixmap. :return: the current alignment :rtype: PyQt4.Qt.Alignment

getAspectRatioMode()

Returns the aspect ratio to apply when drawing the pixmap. :return: the current aspect ratio :rtype: PyQt4.Qt.AspectRatioMode

getPixmap()
Returns the pixmap.Returns None if no pixmap is set. :return: the current pixmap :rtype: PyQt4.Qt.QPixmap

classmethod getQtDesignerPluginInfo()

getTransformationMode()
Returns the transformation mode to apply when drawing the pixmap. :return: the current transformation mode :rtype: PyQt4.Qt.TransformationMode

paintEvent(*paintEvent*)
Overwrite the paintEvent from QWidget to draw the pixmap

pixmap
This property holds the widget's pixmap

Access functions:

- *QPixmapWidget.getPixmap()*
- *QPixmapWidget.setPixmap()*
- QPixmapWidget.resetLedStatus()

recalculatePixmap()

resetAlignment()
Resets the transformation mode to Qt.Qt.AlignLeft | Qt.Qt.AlignVCenter

resetAspectRatioMode()
Resets the aspect ratio mode to KeepAspectRatio

resetPixmap()
Resets the pixmap for this widget.

resetTransformationMode()
Resets the transformation mode to SmoothTransformation

resizeEvent(*event*)

setAlignment(*alignment*)
Sets the alignment to apply when drawing the pixmap. :param pixmap: the new alignment :type pixmap: PyQt4.Qt.Alignment

setAspectRatioMode(*aspect*)
Sets the aspect ratio mode to apply when drawing the pixmap. :param pixmap: the new aspect ratio mode :type pixmap: PyQt4.Qt.AspectRatioMode

setPixmap(*pixmap*)
Sets the pixmap for this widget. Setting it to None disables pixmap :param pixmap: the new pixmap :type pixmap: PyQt4.Qt.QPixmap

setTransformationMode(*transformation*)
Sets the transformation mode to apply when drawing the pixmap. :param pixmap: the new transformation mode :type pixmap: PyQt4.Qt.TransformationMode

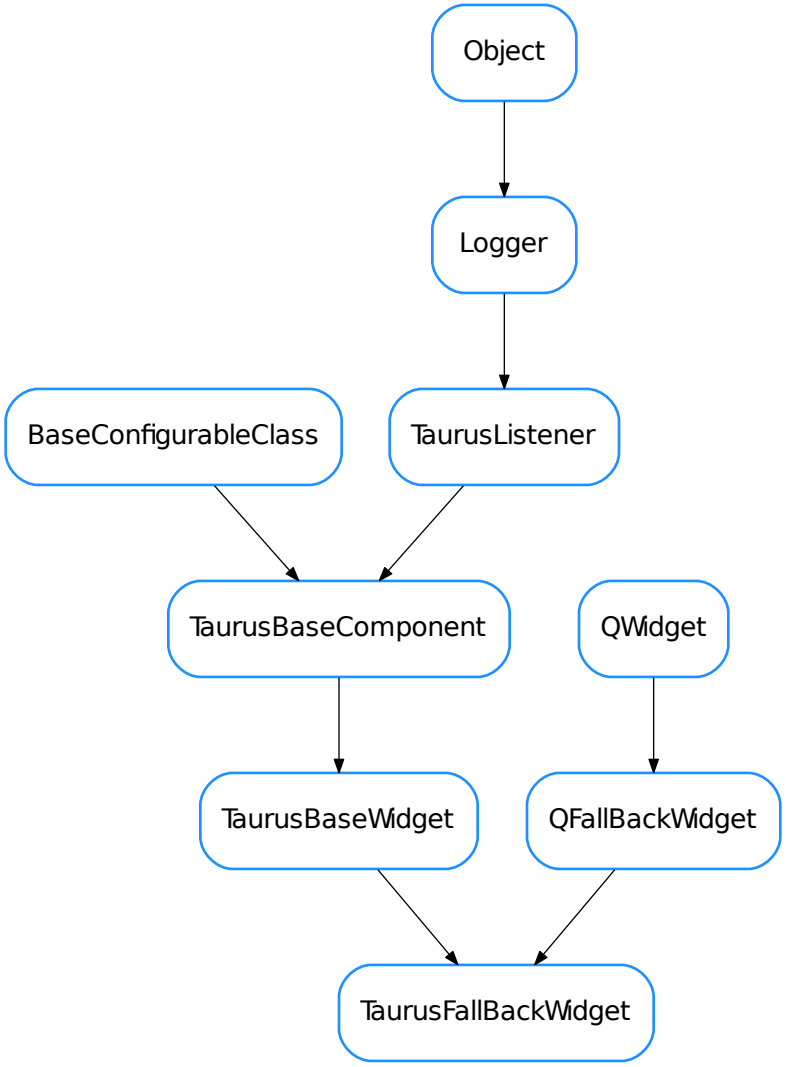
transformationMode
This property holds the widget's pixmap transformation mode

Access functions:

- *QPixmapWidget.getTransformationMode()*
- *QPixmapWidget.setTransformationMode()*

- *QPixmapWidget.resetTransformationMode()*

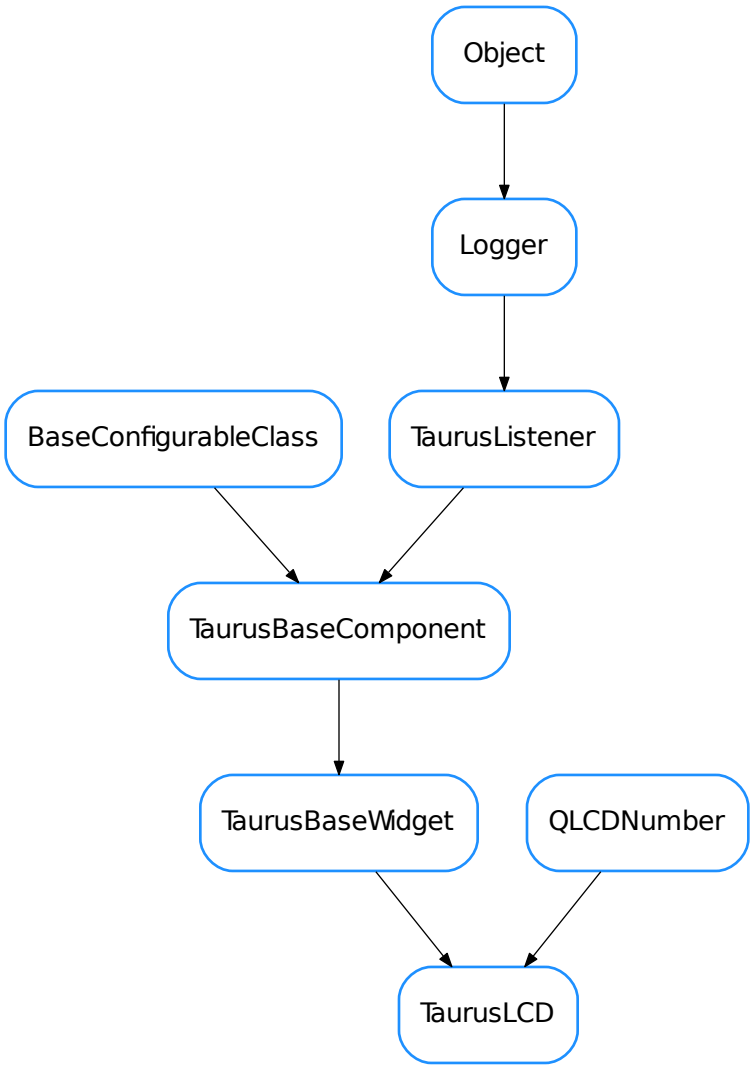
TaurusFallBackWidget



class **TaurusFallBackWidget** (*replaces=None*, *parent=None*, **args*, ***kwargs*)

Bases: `taurus.qt.qtgui.display.qfallback.QFallBackWidget`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget`

TaurusLCD



class TaurusLCD (*parent=None*, *designMode=False*)

Bases: `PyQt4.QtGui.QLCDNumber`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget`

A Taurus-enabled `Qt.QLCDNumber` widget. Its text can represent either the rvalue or wvalue *magnitude* (or nothing), and the background can colour-code the attribute quality or the device state (or nothing)

DefaultBgRole = 'quality'

DefaultFgRole = 'value'

DefaultModelIndex = None

DefaultShowText = True

bgRole
This property holds the background role. Valid values are ‘’/’None’, ‘quality’, ‘state’

Access functions:

- *TaurusLCD.getBgRole()*
- *TaurusLCD.setBgRole()*
- *TaurusLCD.resetBgRole()*

controller()

fgRole
This property holds the foreground role. Valid values are:

1. ‘’/’None’ - no value is displayed
2. ‘value’ - the value is displayed
3. ‘w_value’ - the write value is displayed

Access functions:

- *TaurusLCD.getFgRole()*
- *TaurusLCD.setFgRole()*
- *TaurusLCD.resetFgRole()*

getBgRole()

getFgRole()

getModelIndex()

getModelIndexValue()

classmethod getQtDesignerPluginInfo()

handleEvent(*evt_src*, *evt_type*, *evt_value*)

isReadOnly()

model
This property holds the unique URI string representing the model name with which this widget will get its data from. The convention used for the string can be found *here*.

In case the property *useParentModel* is set to True, the model text must start with a ‘/’ followed by the attribute name.

Access functions:

- TaurusBaseWidget.getModel()
- *TaurusLCD.setModel()*
- TaurusBaseWidget.resetModel()

See also:

Model concept

modelIndex
This property holds the index inside the model value that should be displayed

Access functions:

- *TaurusLCD.getModelIndex()*
- *TaurusLCD.setModelIndex()*
- *TaurusLCD.resetModelIndex()*

See also:

Model concept

resetBgRole()

resetFgRole()

resetModelIndex()

setBgRole(*bgRole*)

setFgRole(*fgRole*)

setModel(*m*)

setModelIndex(*modelIndex*)

useParentModel

This property holds whether or not this widget should search in the widget hierarchy for a model prefix in a parent widget.

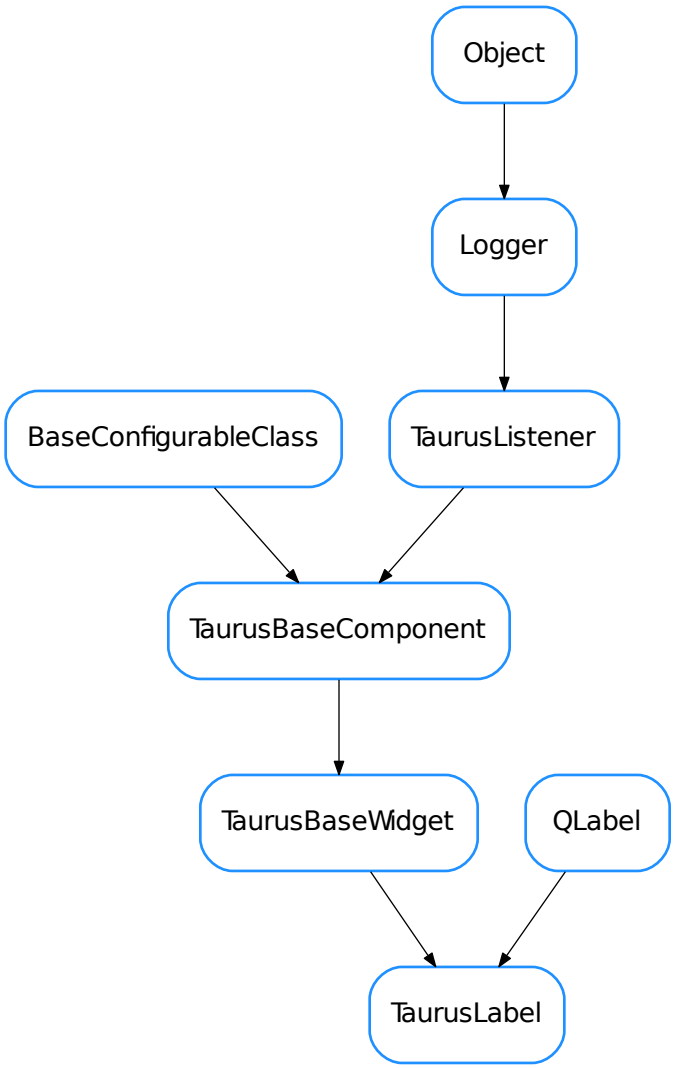
Access functions:

- TaurusBaseWidget.getUseParentModel()
- TaurusBaseWidget.setUseParentModel()
- TaurusBaseWidget.resetUseParentModel()

See also:

Model concept

TaurusLabel



class TaurusLabel (*parent=None*, *designMode=False*)

Bases: PyQt4.QtGui.QLabel, taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget

DefaultAlignment = 130

DefaultAutoTrim = True

DefaultBgRole = 'quality'

DefaultFgRole = 'rvalue'

DefaultModelIndex = None

DefaultPrefix = ''

DefaultShowText = True

DefaultSuffix = ‘’

autoTrim
This property holds the

Access functions:

- *TaurusLabel.getAutoTrim()*
- *TaurusLabel.setAutoTrim()*
- *TaurusLabel.resetAutoTrim()*

bgRole
This property holds the background role. Valid values are ‘’/’None’, ‘quality’, ‘state’

Access functions:

- *TaurusLabel.getBgRole()*
- *TaurusLabel.setBgRole()*
- *TaurusLabel.resetBgRole()*

controller()

controllerUpdate()

dragEnabled
This property holds the

Access functions:

- TaurusLabel.isDragEnabled()
- TaurusLabel.setDragEnabled()
- TaurusLabel.resetDragEnabled()

fgRole
This property holds the foreground role (the text). Valid values are:

1. ‘’/’None’ - no value is displayed
2. ‘value’ - the value is displayed
3. ‘w_value’ - the write value is displayed
4. ‘quality’ - the quality is displayed
5. ‘state’ - the device state is displayed

Access functions:

- *TaurusLabel.getFgRole()*
- *TaurusLabel.setFgRole()*
- *TaurusLabel.resetFgRole()*

getAutoTrim()

getBgRole()

getFgRole()

getModelIndex()

getModelIndexValue()

getModelMimeData()

getPrefixText()

classmethod getQtDesignerPluginInfo()

getSuffixText()

handleEvent(*evt_src*, *evt_type*, *evt_value*)

hasDynamicTextInteractionFlags()

isReadOnly()

model
This property holds the unique URI string representing the model name with which this widget will get its data from. The convention used for the string can be found *here*.

In case the property *useParentModel* is set to True, the model text must start with a '/' followed by the attribute name.

Access functions:

- TaurusBaseWidget.getModel()
- *TaurusLabel.setModel()*
- TaurusBaseWidget.resetModel()

See also:

Model concept

modelIndex
This property holds the index inside the model value that should be displayed

Access functions:

- *TaurusLabel.getModelIndex()*
- *TaurusLabel.setModelIndex()*
- *TaurusLabel.resetModelIndex()*

See also:

Model concept

prefixText
This property holds a prefix text

Access functions:

- *TaurusLabel.getPrefixText()*
- *TaurusLabel.setPrefixText()*
- *TaurusLabel.resetPrefixText()*

resetAutoTrim()

resetBgRole()

resetFgRole()

resetModelIndex()

resetPrefixText()

resetSuffixText()

resetTextInteractionFlags()

resizeEvent(*event*)

setAutoTrim(*trim*)

setBgRole(*bgRole*)

setDynamicTextInteractionFlags(*flags*)

setFgRole(*fgRole*)

setModel(*m*)

setModelIndex(*modelIndex*)

setPrefixText(*prefix*)

setSuffixText(*suffix*)

setTextInteractionFlags(*flags*)

showValueDialog(**args*)

suffixText
This property holds a suffix text

Access functions:

- *TaurusLabel.getSuffixText()*
- *TaurusLabel.setSuffixText()*
- *TaurusLabel.resetSuffixText()*

textInteractionFlags

useParentModel
This property holds whether or not this widget should search in the widget hierarchy for a model prefix in a parent widget.

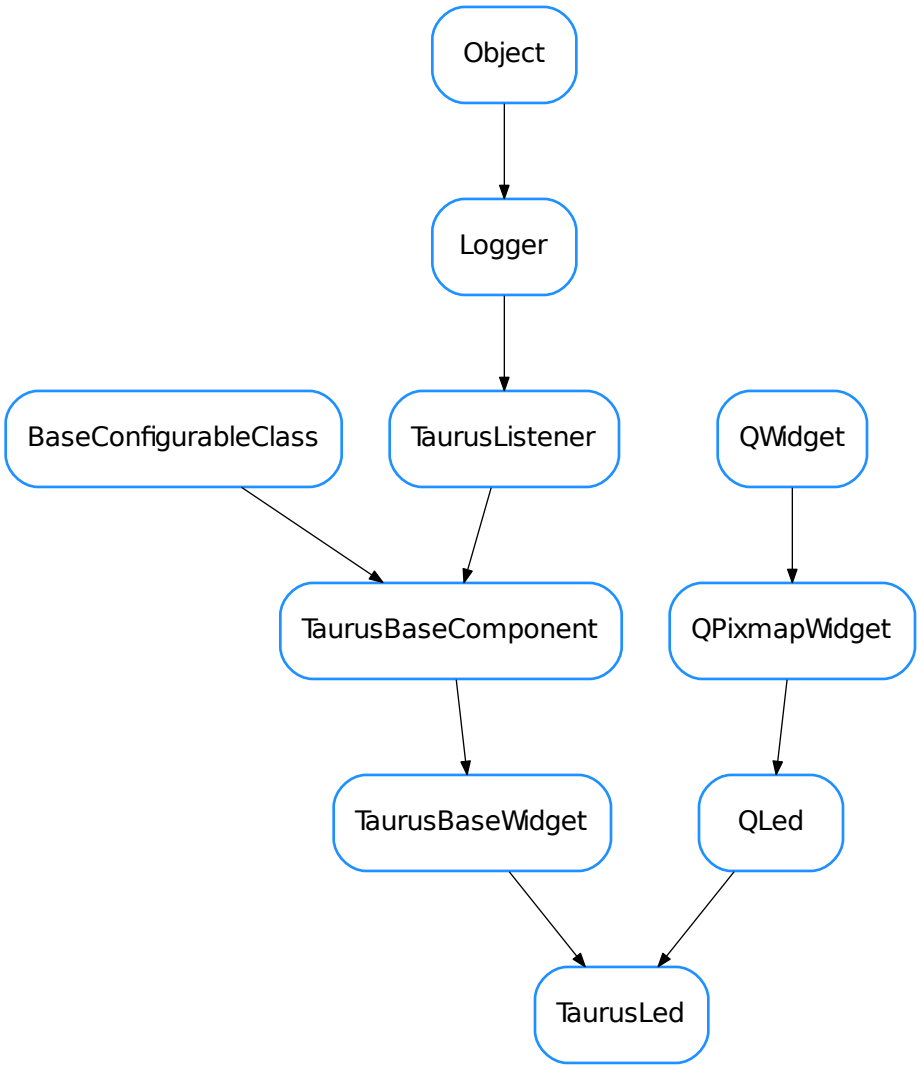
Access functions:

- TaurusBaseWidget.getUseParentModel()
- TaurusBaseWidget.setUseParentModel()
- TaurusBaseWidget.resetUseParentModel()

See also:

Model concept

TaurusLed



class TaurusLed(*parent=None*, *designMode=False*)

Bases: `taurus.qt.qtgui.display.qled.QLed`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget`

A widget designed to represent with a LED image the state of a device, the value of a boolean attribute or the quality of an attribute.

DefaultFgRole = 'rvalue'

DefaultModelIndex = None

DefaultOffColor = 'black'

DefaultOnColor = 'green'

controller()

fgRole
This property holds the foreground role. Valid values are:

1. 'value' - the value is used
2. 'w_value' - the write value is used
3. 'quality' - the quality is used

Access functions:

- *TaurusLed.getFgRole()*
- *TaurusLed.setFgRole()*
- *TaurusLed.resetFgRole()*

getFgRole()

getModelIndex()

getModelIndexValue()

getOffColor()
Returns the preferred led off color :return: led off color :rtype: str

getOnColor()
Returns the preferred led on color :return: led on color :rtype: str

classmethod getQtDesignerPluginInfo()

handleEvent(*evt_src*, *evt_type*, *evt_value*)

isReadOnly()

model
This property holds the unique URI string representing the model name with which this widget will get its data from. The convention used for the string can be found *here*.

In case the property *useParentModel* is set to True, the model text must start with a '/' followed by the attribute name.

Access functions:

- TaurusBaseWidget.getModel()
- *TaurusLabel.setModel()*
- TaurusBaseWidget.resetModel()

See also:

Model concept

modelIndex
This property holds the index inside the model value that should be displayed

Access functions:

- *TaurusLed.getModelIndex()*
- *TaurusLed.setModelIndex()*
- *TaurusLed.resetModelIndex()*

See also:

Model concept

offColor
This property holds the preferred led color This value is used for the cases where the model value does not contain enough information to distinguish between different Off colors. For example, a bool attribute, when it is False it is displayed with the off led but when it is true it may be displayed On in any color. The prefered color would be used in this case.

Access functions:

- *TaurusLed.getOffColor()*
- *TaurusLed.setOffColor()*
- *TaurusLed.resetOffColor()*

onColor
This property holds the preferred led color This value is used for the cases where the model value does not contain enough information to distinguish between different On colors. For example, a bool attribute, when it is False it is displayed with the off led but when it is true it may be displayed On in any color. The prefered color would be used in this case.

Access functions:

- *TaurusLed.getOnColor()*
- *TaurusLed.setOnColor()*
- *TaurusLed.resetOnColor()*

resetFgRole()

resetModelIndex()

resetOffColor()
Resets the preferred led color

resetOnColor()
Resets the preferred led on color

setFgRole(*fgRole*)

setModel(*m*)

setModelIndex(*modelIndex*)

setOffColor(*color*)
Sets the preferred led off color :param status: the new off color :type status: str

setOnColor(*color*)
Sets the preferred led on color :param status: the new on color :type status: str

useParentModel
This property holds whether or not this widget should search in the widget hierarchy for a model prefix in a parent widget.

Access functions:

- TaurusBaseWidget.getUseParentModel()
- TaurusBaseWidget.setUseParentModel()
- TaurusBaseWidget.resetUseParentModel()

See also:

Model concept

- *Q7SegDigit*
- *QFallBackWidget*
- *QLed*
- *QLedOld*
- *QLogo*
- *QPixmapWidget*
- *TaurusFallBackWidget*
- *TaurusLCD*
- *TaurusLabel*
- *TaurusLed*

Functions

create_fallback(*widget_klass_name*)

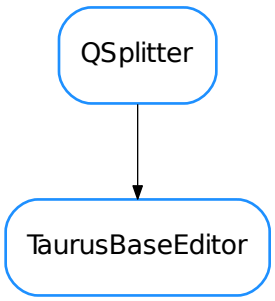
create_taurus_fallback(*widget_klass_name*)

taurus.qt.qtgui.editor

This package contains a collection of taurus text editor widgets

Classes

TaurusBaseEditor



class TaurusBaseEditor(*parent=None*)

Bases: PyQt4.QtGui.QSplitter

clone_editorstack(*editorstack*)

closeEvent(*event*)

close_file_in_all_editorstacks

createMenuActions()
Returns a list of menu actions and a list of IO actions. Reimplement in derived classes. This Base (dummy) implementation creates empty menu actions and a list of 5 dummy actions for the IO actions

create_new_window()

editorStack()

get_focus_widget()

go_to_file(*fname*, *lineno*, *text*)

is_file_opened(*filename=None*)
Dummy implementation that always returns None. Reimplement in derived classes to return the index of already-open files in the editor_stack, or None if the file is not already open.

load(*filename*, *goto=None*)

refresh_save_all_action()

register_editorstack(*editorstack*)

register_editorwindow(*window*)

register_widget_shortcuts(*widget*)
Fake!

reload(*idx=None*, *filename=None*, *goto=None*)

set_current_filename(*filename*)

setup_window(*toolbar_list*, *menu_list*)

unregister_editorstack(*editorstack*)

unregister_editorwindow(*window*)

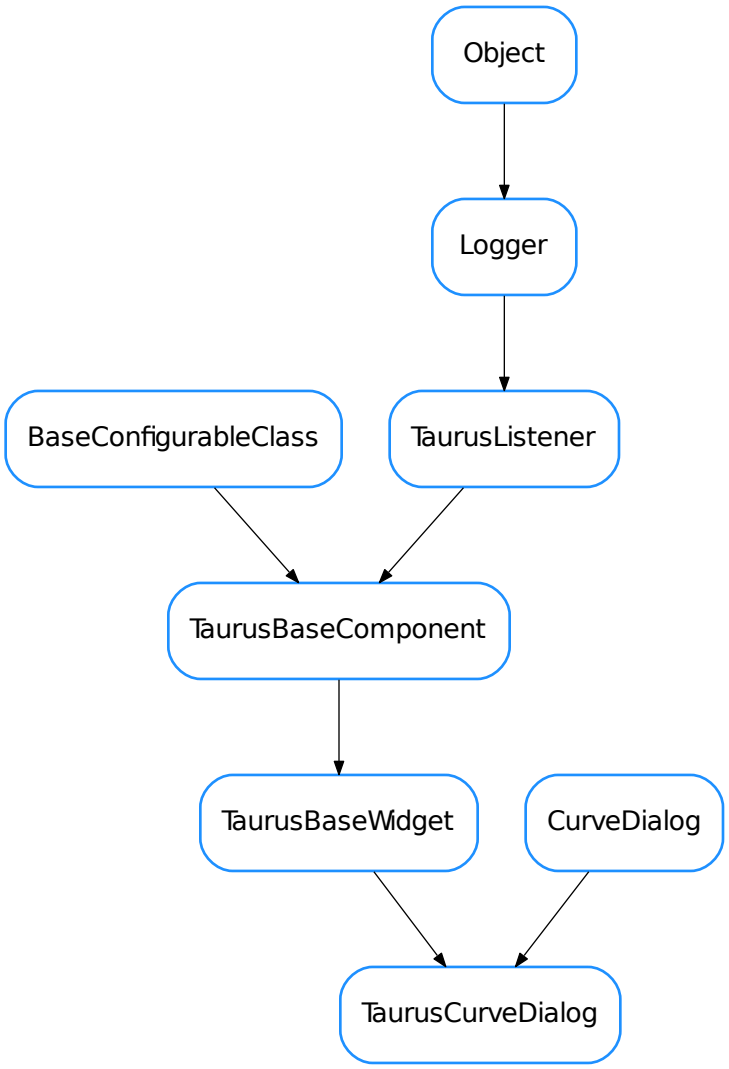
- *TaurusBaseEditor*

taurus.qt.qtgui.extra_guiqwt

This module provides the glue between taurus and guiqwt. It essentially provides taurus extensions to qwtgui

Classes

TaurusCurveDialog



class **TaurusCurveDialog**(*parent=None*, *designMode=False*, *toolbar=True*, ***kwargs*)

Bases: `guiqwt.plot.CurveDialog`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget`

A taurus dialog for showing 1D data. It behaves as a regular `guiqwt.plot.CurveDialog` but it also offers the expected Taurus interface (e.g. setting models, save/apply configs, drag&drops,...)

See also:

`TaurusCurveWidget`

addModels(*modelNames*)

Creates TaurusCurveItems (one for each model in modelNames) and attaches them to the plot.

Note: you can also add curves using `add_items()`. *`addModels()`* is only a more Taurus-oriented interface. `add_items()` gives you more control.

Parameters **`modelNames`** (`sequence` <`str`> or `str`) – the names of the models to be plotted. For convenience, string is also accepted (instead of a sequence of strings), in which case the string will be internally converted to a sequence by splitting it on whitespace and commas. Each model can optionally be composed of two parts, separated by "|" indicating X and Y components for the curve. If only one part is given, it is used for Y and X is automatically generated as an index.

See also:

`add_item()`

`getModel``()`
reimplemented from `TaurusBaseWidget`

`getModelClass``()`
reimplemented from `TaurusBaseWidget`

classmethod **`getQtDesignerPluginInfo`**`()`
reimplemented from `TaurusBaseWidget`

`keyPressEvent`(*event*)

`model`

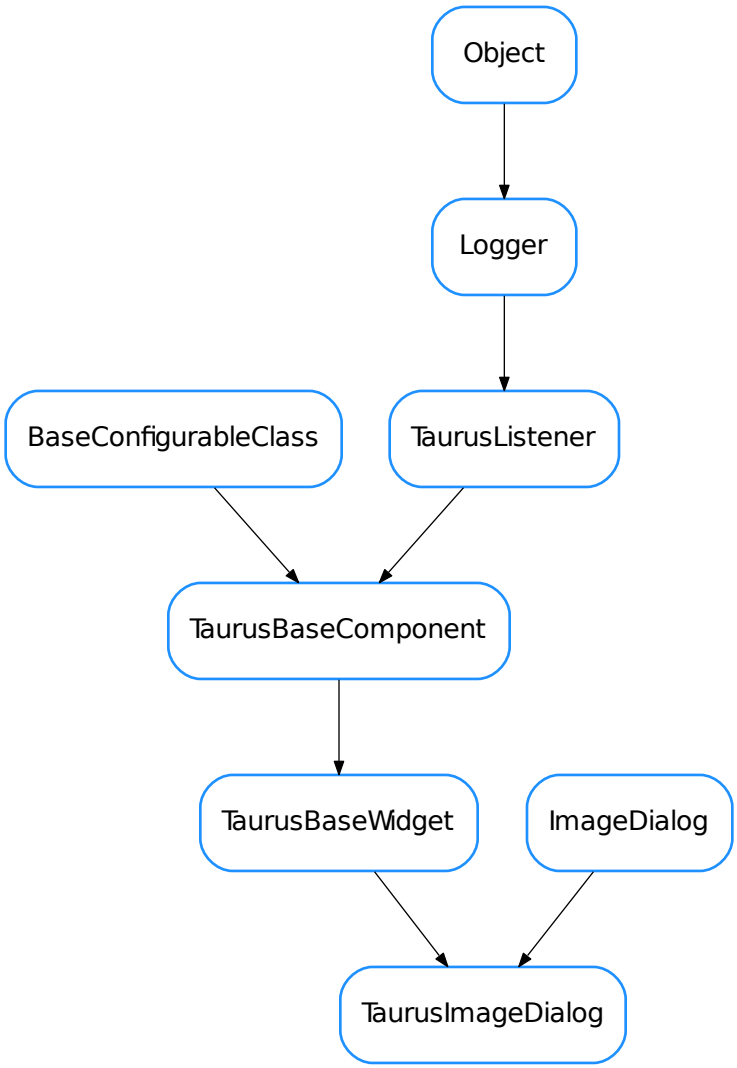
`modelChanged`

`modifiableByUser`

`setModel`

`setModifiableByUser`(*modifiable*)
reimplemented from `TaurusBaseWidget`

TaurusImageDialog



class **TaurusImageDialog**(*parent=None*, *designMode=False*, *toolbar=True*, ***kwargs*)

Bases: `guiqwt.plot.ImageDialog`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget`

A taurus dialog for showing 2D data. It behaves as a regular `guiqwt.plot.ImageDialog` but it also offers the expected Taurus interface (e.g. setting models, save/apply configs, drag&drops,...)

See also:

`TaurusImageWidget`

getModel()

reimplemented from `TaurusBaseWidget`

getModelClass()
reimplemented from TaurusBaseWidget

classmethod getQtDesignerPluginInfo()
reimplemented from TaurusBaseWidget

getRGBmode()

keyPressEvent(*event*)

model

modifiableByUser

resetRGBmode()

rgbmode

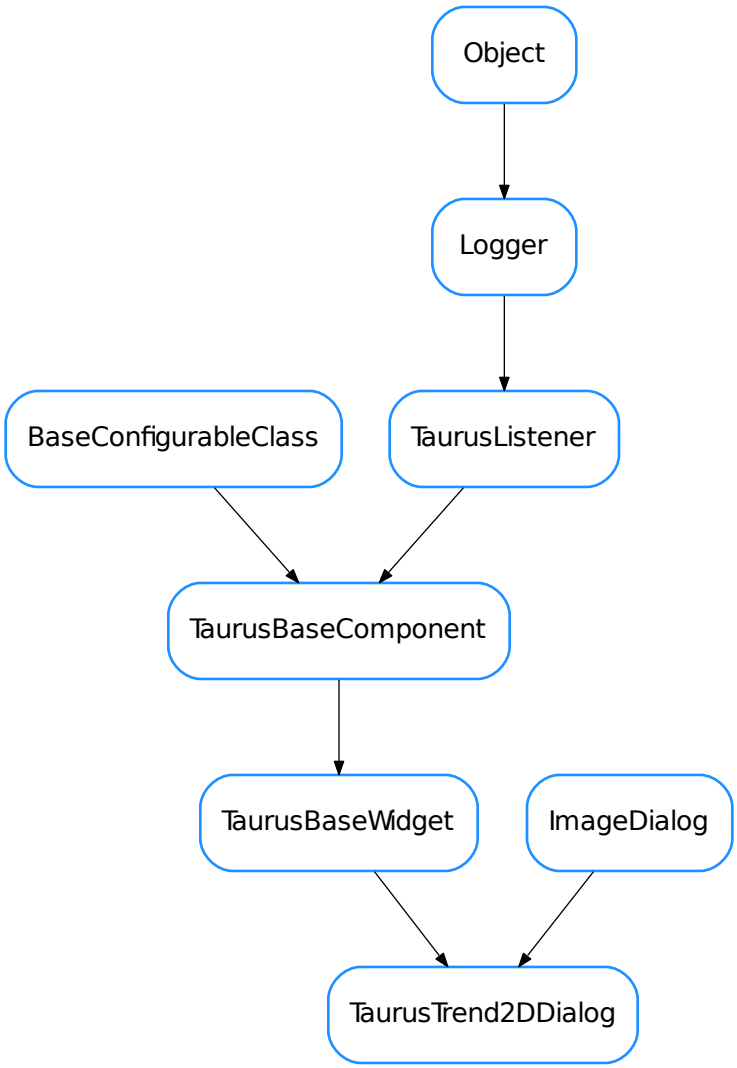
setModel

setModifiableByUser(*modifiable*)
reimplemented from TaurusBaseWidget

setRGBmode(*enable*)

useParentModel

TaurusTrend2DDialog



class **TaurusTrend2DDialog**(*parent=None*, *designMode=False*, *toolbar=True*, *stackMode='deltatime'*, *buffersize=512*, *options=None*, *autoscale='xyz'*, ***kwargs*)

Bases: `guiqwt.plot.ImageDialog`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget`

This is a widget for displaying trends from 1D Taurus attributes (i.e., representing the variation over time of a 1D array). Sometimes this kind of plots are also known as "spectrograms".

The widget shows a 3D plot (Z represented with colors) where the values in the 1D array are plotted in the Y-Z plane and are stacked along the X axis.

getMaxDataBufferSize()

returns the maximum number of events that can be plotted in the trend

Return type int

Returns

getModel()
reimplemented from TaurusBaseWidget

getModelClass()
reimplemented from TaurusBaseWidget

classmethod **getQtDesignerPluginInfo**()
reimplemented from TaurusBaseWidget

getStackMode()

getUseArchiving()
whether TaurusTrend is looking for data in the archiver when needed

Return type bool

Returns

See also:

setUseArchiving()

keyPressEvent(*event*)

maxDataBufferSize

model

modifiableByUser

resetMaxDataBufferSize()
Same as setMaxDataBufferSize(512) (i.e. 512 events)

resetStackMode()

resetUseArchiving()
Same as setUseArchiving(False)

setMaxDataBufferSize(*maxSize*)
sets the maximum number of events that will be stacked

Parameters **maxSize** (int) – the maximum limit

See also:

TaurusTrendSet

setModel(*model*)
reimplemented from TaurusBaseWidget

setModifiableByUser(*modifiable*)
reimplemented from TaurusBaseWidget

setStackMode(*mode*)
set the type of stack to be used. This determines how X values are interpreted:

- as timestamps ('datetime')
- as time deltas ('timedelta')
- as event numbers ('event')

Parameters **mode** (one of 'datetime', 'timedelta' or 'event') –

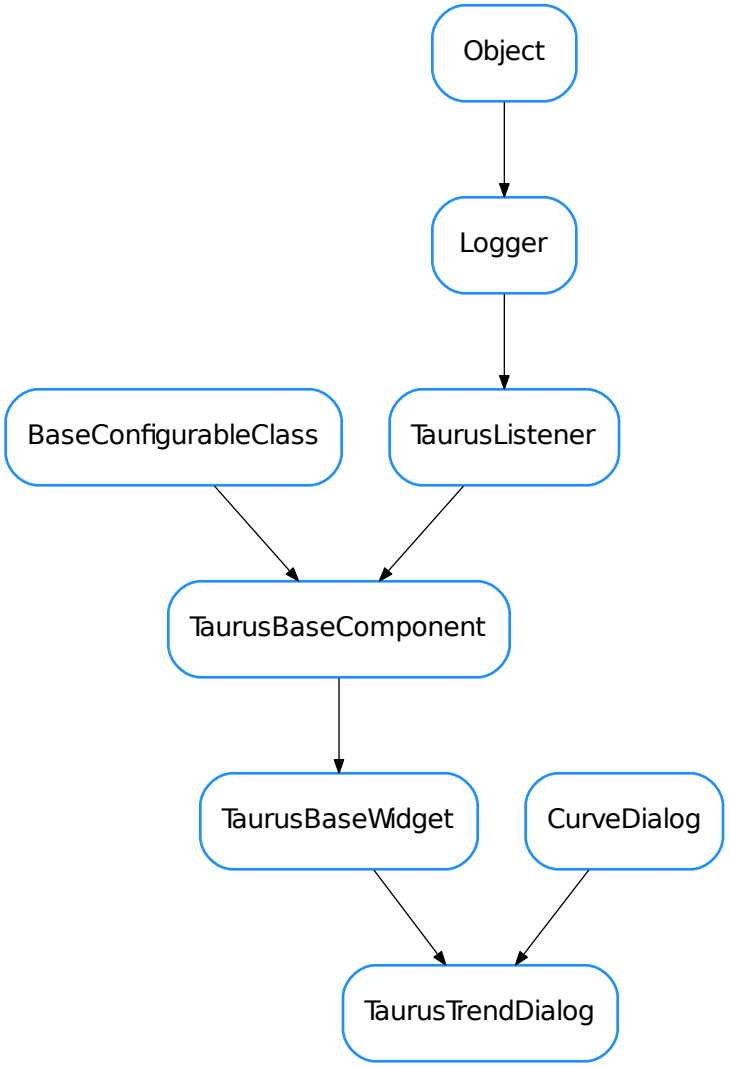
setUseArchiving(*enable*)
enables/disables looking up in the archiver for data stored before the Trend was started

Parameters **enable** (bool) – if True, archiving values will be used if available

stackMode

useArchiving

TaurusTrendDialog



class **TaurusTrendDialog**(*parent=None*, *designMode=False*, *taurusparam=None*, *toolbar=True*, ***kwargs*)
Bases: guiqwt.plot.CurveDialog, taurus.qt.qtgui.base.taurusbase.

TaurusBaseWidget

A taurus widget for showing trends of scalar data. It is an specialization of guiqwt.plot.CurveWidget, for displaying trends and offering the expected Taurus interface (e.g. setting models, save/apply configs, drag&drops,...)

See also:

TaurusTrendDialog

addModels(*modelNames*)
Creates TaurusCurveItems (one for each model in modelNames) and attaches them to the plot.

Note: you can also add curves using add_items(). *addModels()* is only a more Taurus-oriented interface. add_items() gives you more control.

Parameters **modelNames** (sequence <str> or str) – the names of the models to be plotted. For convenience, a string is also accepted (instead of a sequence of strings), in which case the string will be internally converted to a sequence by splitting it on whitespace and commas.

See also:

add_item()

getDropEventCallback()
reimplemented from TaurusBaseWidget

getMaxDataBufferSize()
returns the maximum number of events that can be plotted in the trend

Return type int

Returns

getModel()
reimplemented from TaurusBaseWidget

getModelClass()
reimplemented from TaurusBaseWidget

classmethod getQtDesignerPluginInfo()
reimplemented from TaurusBaseWidget

getStackMode()

getTaurusTrendItems()

getUseArchiving()
whether TaurusTrend is looking for data in the archiver when needed

Return type bool

Returns

See also:

setUseArchiving()

keyPressEvent(*event*)

maxDataBufferSize

model

modelChanged

modifiableByUser

resetMaxDataBufferSize()
Same as setMaxDataBufferSize(16384)

resetStackMode()

resetUseArchiving()
Same as setUseArchiving(False)

setMaxDataBufferSize(*maxSize*)
sets the maximum number of events that will be stacked

Parameters **maxSize** (int) – the maximum limit

See also:

TaurusTrendSet

setModel

setModifiableByUser(*modifiable*)
reimplemented from TaurusBaseWidget

setStackMode(*mode*)
set the type of stack to be used. This determines how X values are interpreted:

•as timestamps ('datetime')

•as time deltas ('timedelta')

•as event numbers ('event')

Parameters **mode** (one of 'datetime', 'timedelta' or 'event') –

setUseArchiving(*enable*)
enables/disables looking up in the archiver for data stored before the Trend was started

Parameters **enable** (bool) – if True, archiving values will be used if available

stackMode

useArchiving

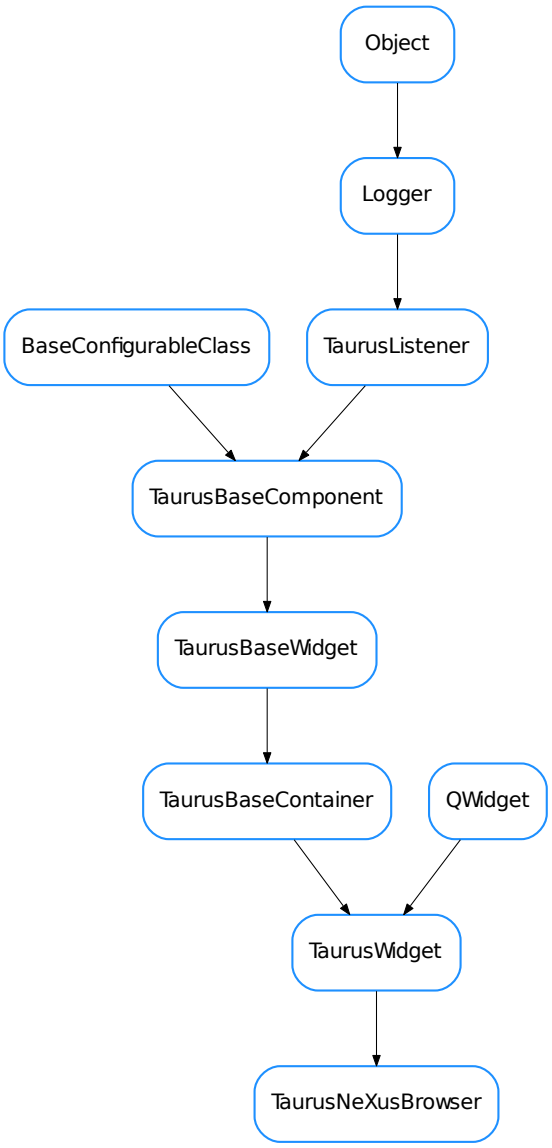
- *TaurusCurveDialog*
- *TaurusImageDialog*
- *TaurusTrend2DDialog*
- *TaurusTrendDialog*

taurus.qt.qtgui.extra_nexus

__init__.py:

Classes

TaurusNeXusBrowser



class TaurusNeXusBrowser (**args*, ***kwargs*)

Bases: `taurus.qt.qtgui.container.tauruswidget.TaurusWidget`

A Browser for nexus files with optional preview. Based on PyMCA's HDF5Widget

findNodeIndex (*filename*, *nodename*)

classmethod getQtDesignerPluginInfo ()

neXusPreviewWidgetFactory (*ddict*)

returns a widget showing a preview of a node in a NeXus file

neXusWidget ()

onHDF5WidgetSignal (*ddict*)

openFile (*fname=None*)

setCurrentNode (*filename*, *nodename*)

- *TaurusNeXusBrowser*

taurus.qt.qtgui.graphic

This package contains a collection of taurus Qt graphics view widgets

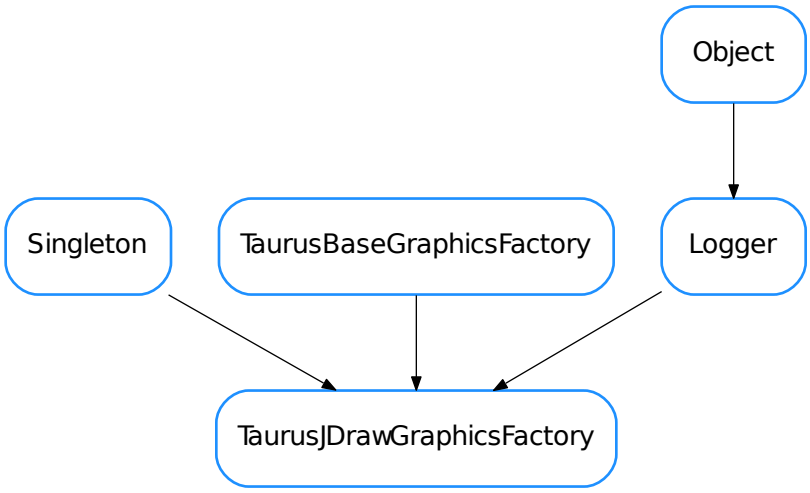
Modules

taurus.qt.qtgui.graphic.jdraw

This package contains the jdraw file format specific classes

Classes

TaurusJDrawGraphicsFactory



class TaurusJDrawGraphicsFactory (*parent*, *alias=None*, *delayed=False*)

Bases: `taurus.core.util.singleton.Singleton`, `taurus.qt.qtgui.graphic.taurusgraphic.TaurusBaseGraphicsFactory`, `taurus.core.util.log.Logger`

getEllipseObj (*params*)

getGroupObj(*params*)

getImageObj(*params*)

getLabelObj(*params*)

getLineObj(*params*)

getObj(*name*, *params*)

getPolylineObj(*params*)

getRectangleObj(*params*)

getRoundRectangleObj(*params*)

getSceneObj(*items*)

getSplineObj(*params*)

getSwingObjectObj(*params*)

getZBufferLevel()

incZBufferLevel()

init(**args*, ***kwargs*)
Singleton instance initialization.

readLabelObj(*item*, *params*)

readSimpleScalarViewerObj(*item*, *params*)

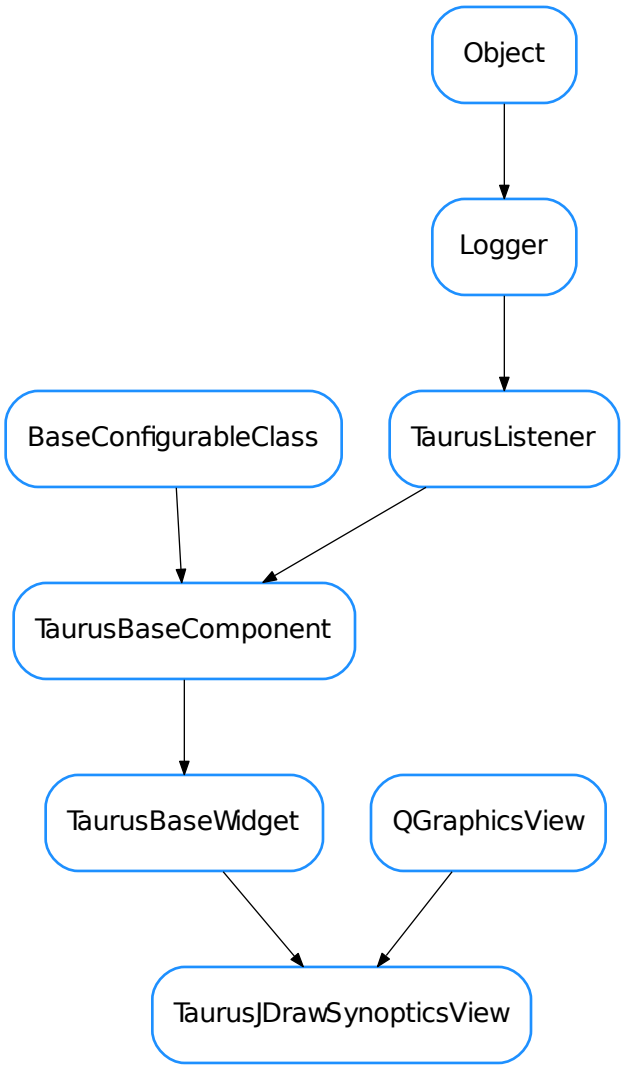
resetZBufferLevel()

setZBufferLevel(*level*)

set_common_params(*item*, *params*)

set_item_filling(*item*, *pattern=5*, *expand=False*)

TaurusJDrawSynopticsView



class **TaurusJDrawSynopticsView**(*parent=None*, *designMode=False*, *updateMode=None*, *alias=None*, *resizable=True*, *panelClass=None*)

Bases: `PyQt4.QtGui.QGraphicsView`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget`

Taurus Class that visualizes Synoptics drawn with the JDraw tool (by ESRF). It is equivalent to ATK Synoptic Player (Java).

After initialization call setModel('/your/file.jdw') to parse the synoptic file and connect to controlled objects.

Arguments to TaurusJDrawSynopticsView() creator are:

- designMode; used by Qt Designer

- •updateMode; controls Qt Viewport refresh (disabled by default)
- •alias; a dictionary of name replacements to be applied on graphical objects
- •resizable: whether to allow resizing or not
- •panelClass: class object, class name or shell command to be shown when an object is clicked (None will show default panel, '' or 'noPanel' will disable it)

TaurusJDrawSynopticsView and TaurusGraphicsScene signals/slots

External events:

```
Slot selectGraphicItem(const QString &) displays a selection
mark around the TaurusGraphicsItem that matches the argument passed.
```

Mouse Left-button events:

```
Signal graphicItemSelected(QString) is triggered, passing the
selected TaurusGraphicsItem.name() as argument.
```

Mouse Right-button events:

```
TaurusGraphicsItem.setContextMenu([(ActionName,ActionMethod(device_name))]
allows to configure custom context menus for graphic items using a list
of tuples. Empty tuples will insert separators in the menu.
```

closeEvent(*event=None*)

classmethod defaultPanelClass(*klass*)
This method assigns the Class used to open new object panels on double-click (TaurusDevicePanel by default) If an string is used it can be either a Taurus class or an OS launcher

defineStyle()

emitColors()
emit signal which is used to refresh the tree and colors of icons depend of the current status in jdrawSynoptic

fitting(*ADJUST_FRAME=False*)
Parent size is the size of the bigger panel (desn't keep ratio) Rect size never changes (fixed by the graphics objects) Size and SizeHint move one around the other

the method works well until an object is clicked, then the whole reference changes and doesn't work again.

getFramed()

getGraphicsFactory(*delayed=False*)

getModel()

getModelMimeData()
Used for drag events

classmethod getQtDesignerPluginInfo()

getSelectionStyle()

getSelectionStyleName()

get_device_list()

get_item_colors(*emit=False*)

get_item_list()

get_sizes()

graphicItemSelected

graphicSceneClicked

isReadOnly()

itemsChanged

model

modelsChanged

mousePressEvent(*event*)
Records last event position to use it for DragEvents

openJDraw()

panelClass()

refreshModel()

repaint()

resetSelectionStyle()

resizable()

resizeEvent(*event*)
It has been needed to reimplent size policies

selectGraphicItem

selectionStyle

setAlias(*alias*)
Assigning a dictionary like {'Tag':'Value'} with tags to be replaced in object names while parsing.

classmethod setDefaultPanelClass(*klass*, *other*)
This method returns the Class used to open new object panels on double-click (TaurusDevicePanel by default)

setModel

setModels()
This method triggers item.setModel(item._name) in all internal items.

setPanelClass(*widget*)

setResizable(*resizable*)

setSelectionStyle(*selectionStyle*)

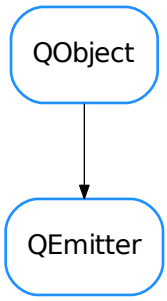
update()

updateStyle()

- *TaurusJDrawGraphicsFactory*
- *TaurusJDrawSynopticsView*

Classes

QEmitter

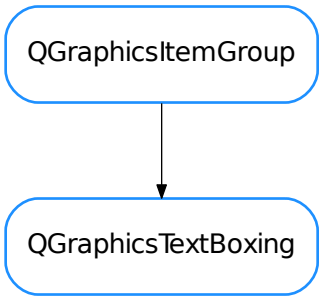


class **QEmitter**(**a*, ***kw*)

Bases: PyQt4.QtCore.QObject

updateView

QGraphicsTextBoxing



class **QGraphicsTextBoxing**(*parent=None*, *scene=None*)

Bases: PyQt4.QtGui.QGraphicsItemGroup

Display a text inside a virtual box. Support horizontal and vertical alignment

brush()

paint(*painter*, *option*, *widget*)

pen()

setAlignment(*alignment*)

setBrush(*brush*)

setDefaultTextColor(*color*)

setFont(*font*)

setHtml(*html*)

setPen(*pen*)

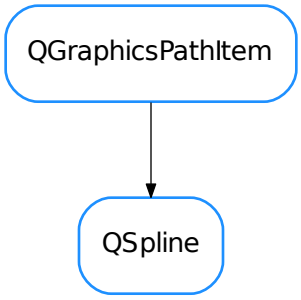
setPlainText(*text*)

setRect(*x*, *y*, *width*, *height*)

setValidBackground(*color*)

toPlainText()

QSpline



class QSpline(*parent=None*, *closed=False*, *control_points=None*)

Bases: `PyQt4.QtGui.QGraphicsPathItem`

setClose(*isClosed*)

setControlPoints(*control_points*)

updateSplinePath()

TaurusBaseGraphicsFactory

class `TaurusBaseGraphicsFactory`

`getEllipseObj`(*params*)

`getGraphicsClassItem`(*cls*, *type_*)

`getGraphicsItem`(*type_*, *params*)

`getGroupObj`(*params*)

`getImageObj`(*parms*)

`getLabelObj`(*params*)

`getLineObj`(*params*)

`getNameParam`()
Returns the name of the parameter which contains the name identifier. Default implementation returns 'name'. Overwrite has necessary.

`getObj`(*name*, *params*)

`getPolylineObj`(*params*)

`getRectangleObj`(*params*)

`getRoundRectangleObj`(*params*)

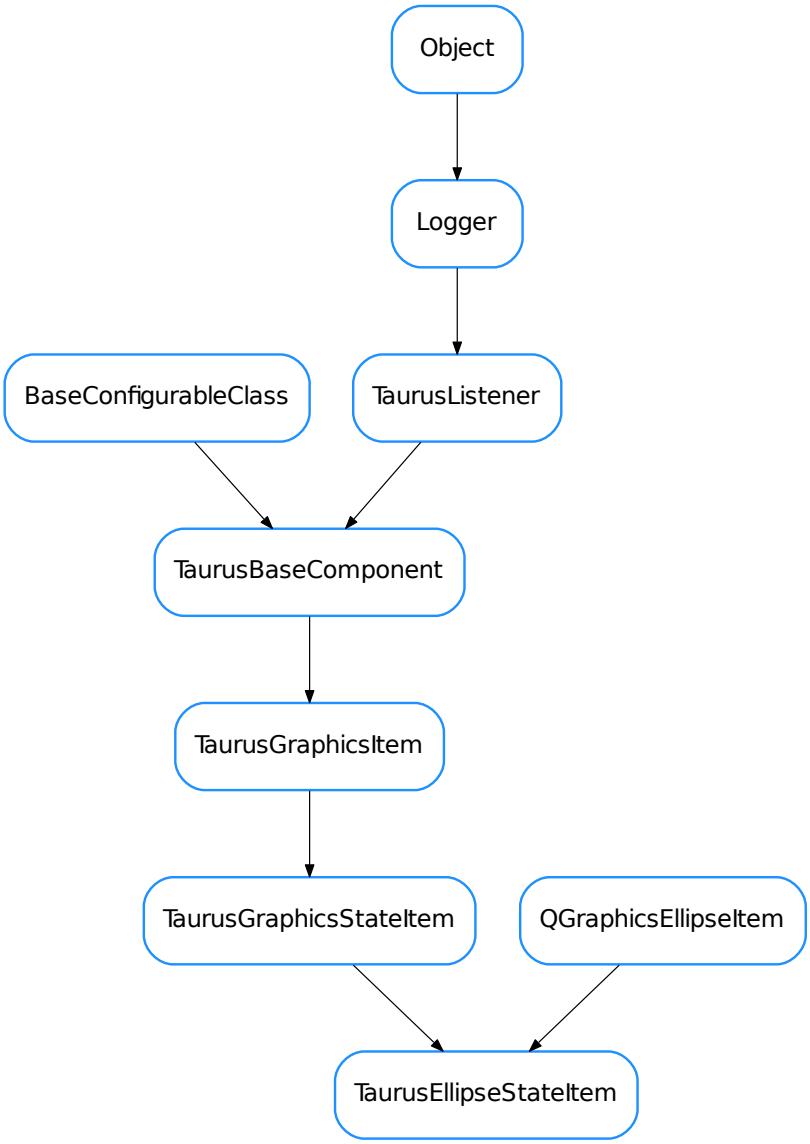
`getSceneObj`()

`getSplineObj`(*params*)

`getSwingObjectObj`(*params*)

`set_common_params`(*item*, *params*)
Sets the common parameters. Default implementation does nothing. Overwrite has necessary.

TaurusEllipseStateItem

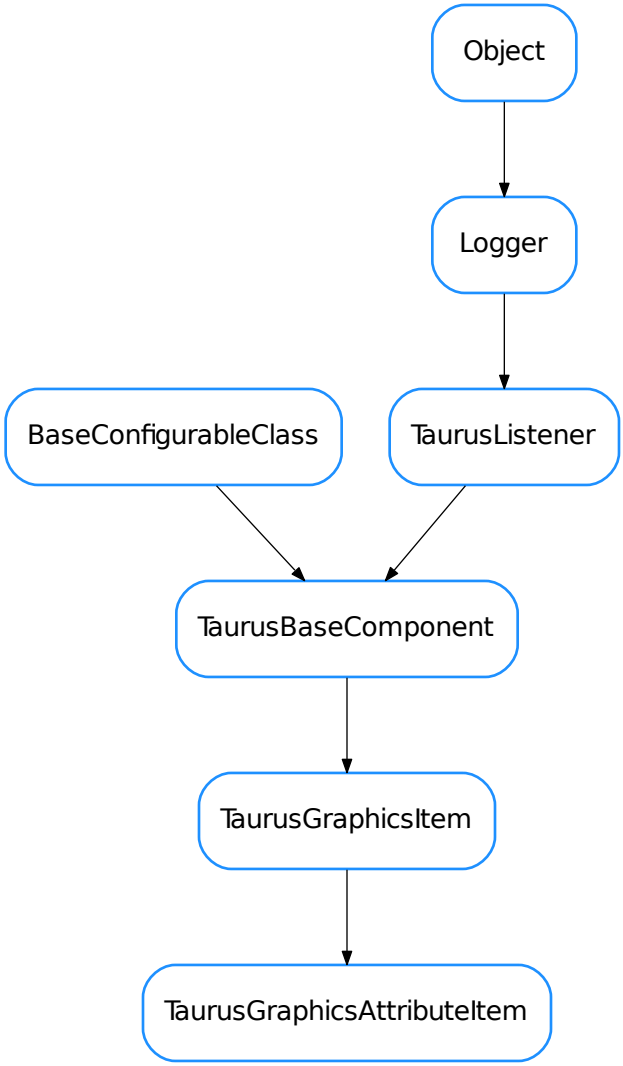


class **TaurusEllipseStateItem**(*name=None*, *parent=None*, *scene=None*)

Bases: `PyQt4.QtGui.QGraphicsEllipseItem`, `taurus.qt.qtgui.graphic.taurusgraphic.TaurusGraphicsStateItem`

paint(*painter*, *option*, *widget=None*)

TaurusGraphicsAttributeItem



class **TaurusGraphicsAttributeItem**(*name=None*, *parent=None*)

Bases: `taurus.qt.qtgui.graphic.taurusgraphic.TaurusGraphicsItem`

This class show value->text conversion in label widgets. Quality is shown in background

getUnit(**args*, ***kwargs*)

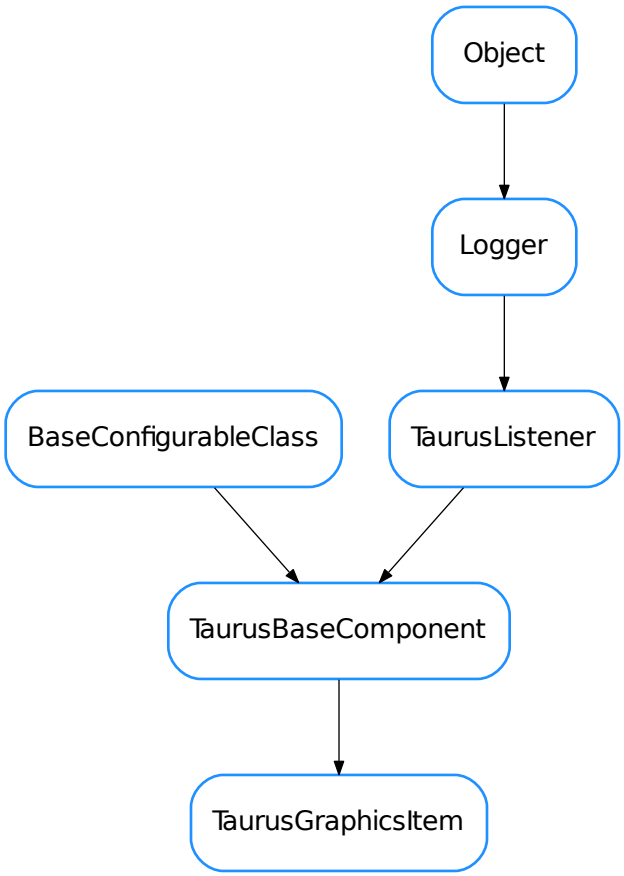
Deprecated since version 4.0.3: Use .getDisplayValue(fragmentName='rvalue.units') instead

setUnitVisible(*yesno*)

setUserFormat(*format*)

updateStyle()

TaurusGraphicsItem



class **TaurusGraphicsItem**(*name=None*, *parent=None*)

Bases: `taurus.qt.qtgui.base.taurusbase.TaurusBaseComponent`

Base class for all Taurus Graphics Items

contextMenu()

fireEvent(*evt_src=None*, *evt_type=None*, *evt_value=None*)

fires a value changed event to all listeners

getExtensions()

Any in ExtensionsList,noPrompt,standAlone,noTooltip,noSelect,ignoreRepaint,shellCommand,className,classParams

getModelClass()

getName()

getParentTaurusComponent()

Returns a parent Taurus component or None if no parent TaurusBaseComponent is found.

isReadOnly()

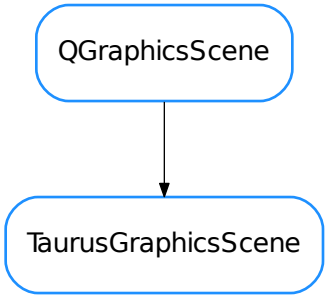
setContextMenu (*menu*)
Context Menu must be a list of tuples (ActionName,ActionMethod), empty tuples insert separators between options.

setModel (*model*)

setName (*name*)

updateStyle ()
Method called when the component detects an event that triggers a change in the style.

TaurusGraphicsScene



class TaurusGraphicsScene (*parent=None*, *strt=True*)
Bases: `PyQt4.QtGui.QGraphicsScene`

This class encapsulates TaurusJDrawSynopticsView and TaurusGraphicsScene signals/slots

External events:

```
Slot selectGraphicItem(const QString &) displays a selection
mark around the TaurusGraphicsItem that matches the argument passed.
```

Mouse Left-button events:

```
Signal graphicItemSelected(QString) is triggered, passing the
selected TaurusGraphicsItem.name() as argument.
```

Mouse Right-button events:

```
TaurusGraphicsItem.setContextMenu([(ActionName,ActionMethod(device_name))]
allows to configure custom context menus for graphic items using a list
of tuples. Empty tuples will insert separators in the menu.
```

ANY_ATTRIBUTE_SELECTS_DEVICE = True

TRACE_ALL = False

addItem (*item*)

addWidget (*item*, *flags=None*)

clearSelection()

closeAllPanels()
This method replaces killProcess, using taurus.qt.qtgui.util.ExternalAppAction instead!

drawSelectionMark(*x*, *y*, *w*, *h*, *oversize=1*)
If h or w are None the mark is drawn at x,y If h or w has a value the mark is drawn in the center of the region ((x,y)(x+w,y+h))

getAllChildren(*item*, *klass=None*)
Returns all children elements, filtering by klass if wanted

getClass(*clName*)

getItemByName(*item_name*, *strict=None*)
Returns a list with all items matching a given name.

Parameters **strict** (bool or None) – controls whether full_name (strict=True) or only device name (False) must match

Return type list

Returns items

getItemByPosition(*x*, *y*)
This method will try first with named objects; if failed then with itemAt

getItemClicked(*mouseEvent*)

getQueue()

getSelectionMark(*picture=None*, *w=10*, *h=10*)

getShellCommand(*obj*, *wait=False*)

static getTaurusParentItem(*item*, *top=True*)
Searches within a group hierarchy and returns a parent Taurus component or None if no parent TaurusBaseComponent is found.

graphicItemSelected

graphicSceneClicked

mouseDoubleClickEvent(*event*)

mousePressEvent(*mouseEvent*)

refreshTree2

selectGraphicItem(*item_name*)
A blue circle is drawn around the matching item name. If the item_name is empty, or it is a reserved keyword, or it has the "noSelect" extension, then the blue circle is removed from the synoptic.

setSelectionMark(*picture=None*, *w=10*, *h=10*)
This method allows to set a callable, graphic item or pixmap as selection mark (by default creates a blue circle). If picture is a callable, the object returned will be used as selection mark. If picture is a QGraphicsItem it will be used as selection mark. If picture is a QPixmap or a path to a pixmap a QGraphicsPixmapItem will be created. If no picture is provided, a blue ellipse will be drawn around the selected object. h/w will be used for height/width of the drawn object.

setSelectionStyle(*selectionStyle*)

showNewPanel(*args=None*, *standAlone=False*)

start()

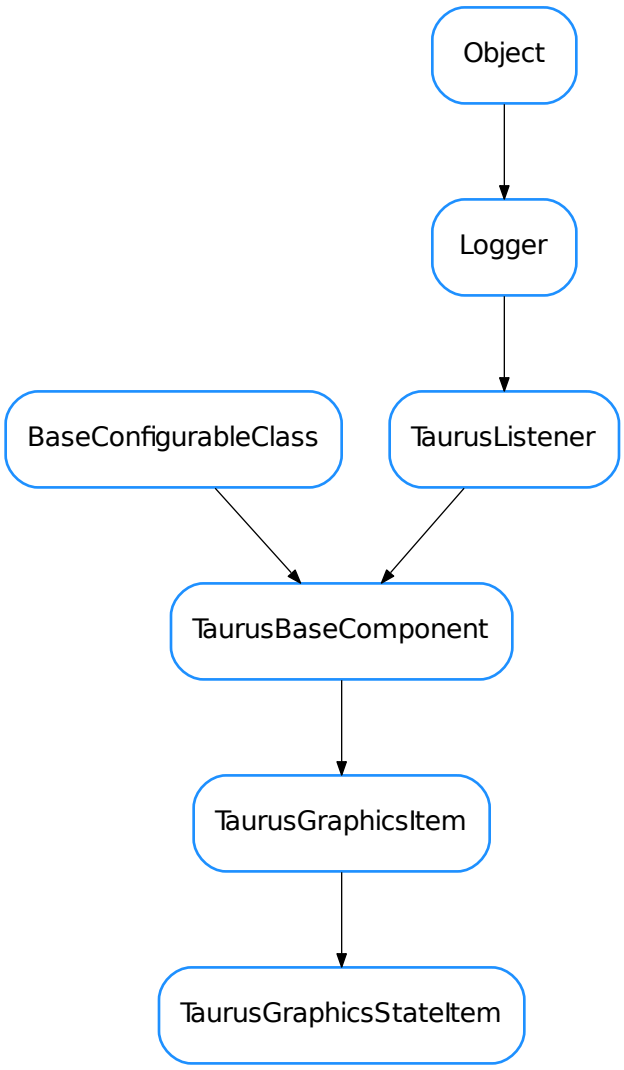
updateScene()

updateSceneItem(*item*)

updateSceneItems(*items*)

updateSceneViews()

TaurusGraphicsStateItem



class TaurusGraphicsStateItem(*name=None*, *parent=None*)

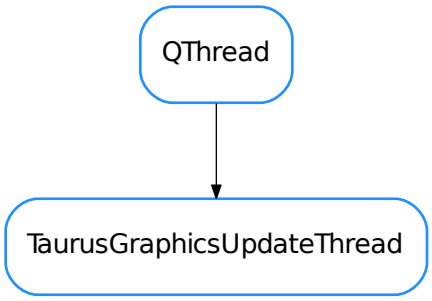
Bases: `taurus.qt.qtgui.graphic.taurusgraphic.TaurusGraphicsItem`

In State Item the displayValue should not override the label This item will modify only foreground/background

colors

updateStyle()

TaurusGraphicsUpdateThread

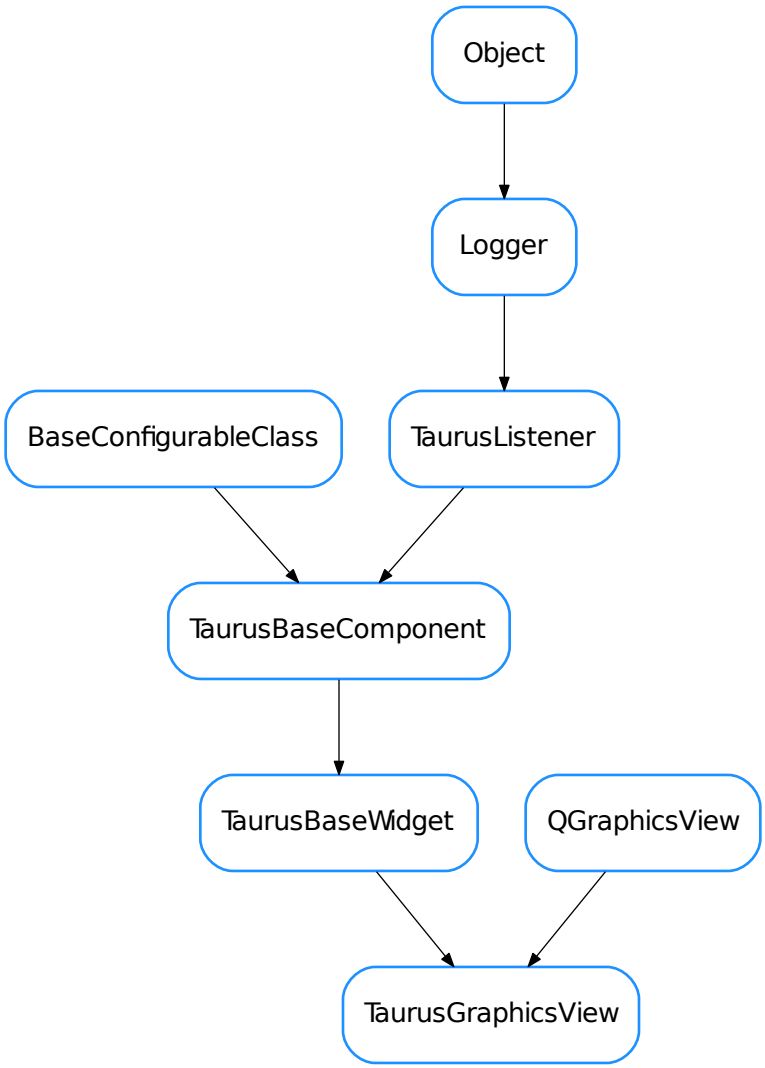


class TaurusGraphicsUpdateThread(*parent=None*, *period=3*)

Bases: PyQt4.QtCore.QThread

run()

TaurusGraphicsView



class TaurusGraphicsView (*parent=None*, *designMode=False*)

Bases: PyQt4.QtGui.QGraphicsView, taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget

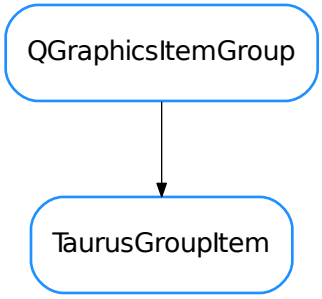
defineStyle()

classmethod getQtDesignerPluginInfo()

isReadOnly()

updateStyle()

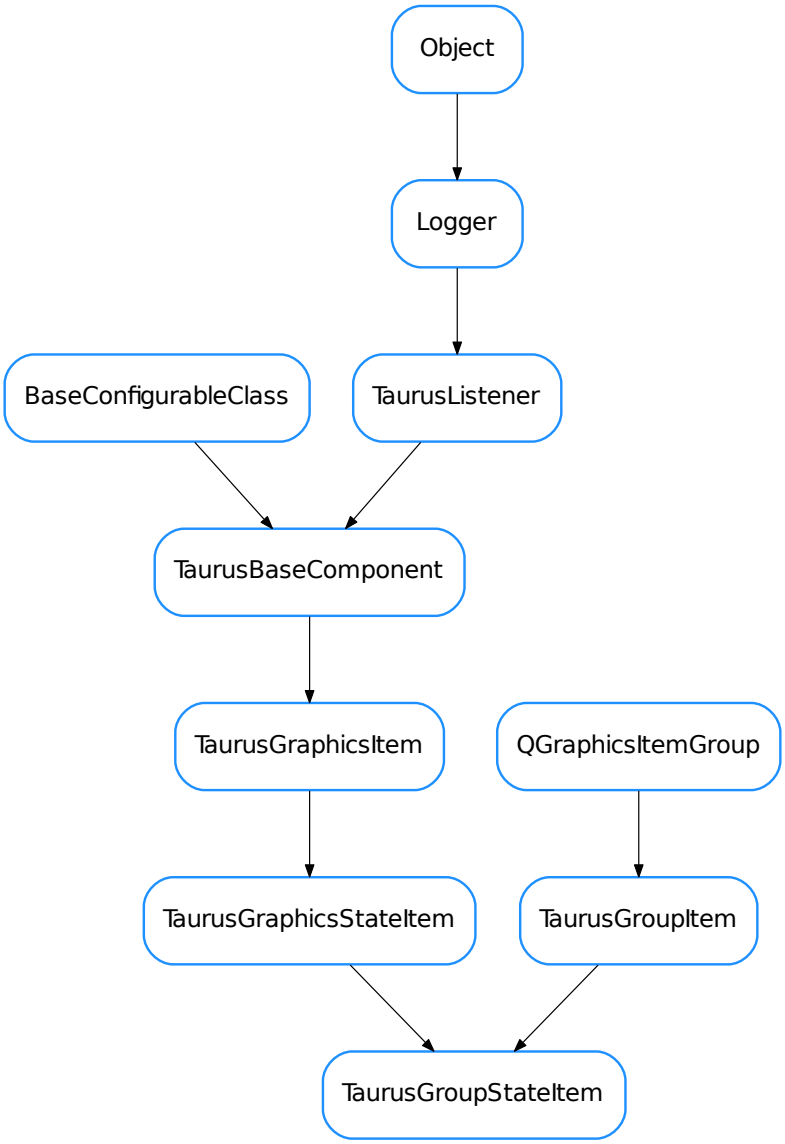
TaurusGroupItem



class **TaurusGroupItem**(*name=None*, *parent=None*, *scene=None*)

Bases: `PyQt4.QtGui.QGraphicsItemGroup`

TaurusGroupStateItem

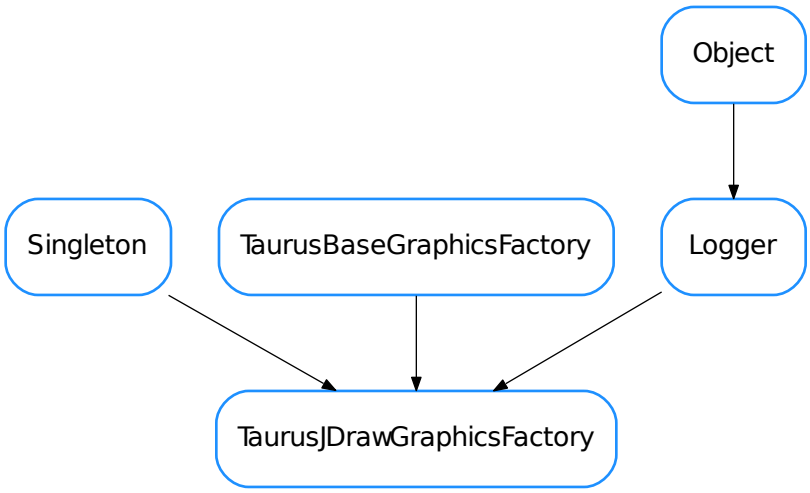


class `TaurusGroupStateItem`(*name=None*, *parent=None*, *scene=None*)

Bases: `taurus.qt.qtgui.graphic.taurusgraphic.TaurusGroupItem`, `taurus.qt.qtgui.graphic.taurusgraphic.TaurusGraphicsStateItem`

paint(*painter*, *option*, *widget*)

TaurusJDrawGraphicsFactory



class **TaurusJDrawGraphicsFactory**(*parent*, *alias=None*, *delayed=False*)

Bases: `taurus.core.util.singleton.Singleton`, `taurus.qt.qtgui.graphic.taurusgraphic.TaurusBaseGraphicsFactory`, `taurus.core.util.log.Logger`

getEllipseObj(*params*)

getGroupObj(*params*)

getImageObj(*params*)

getLabelObj(*params*)

getLineObj(*params*)

getObj(*name*, *params*)

getPolylineObj(*params*)

getRectangleObj(*params*)

getRoundRectangleObj(*params*)

getSceneObj(*items*)

getSplineObj(*params*)

getSwingObjectObj(*params*)

getZBufferLevel()

incZBufferLevel()

init(**args*, ***kwargs*)

Singleton instance initialization.

readLabelObj(*item*, *params*)

readSimpleScalarViewerObj(*item*, *params*)

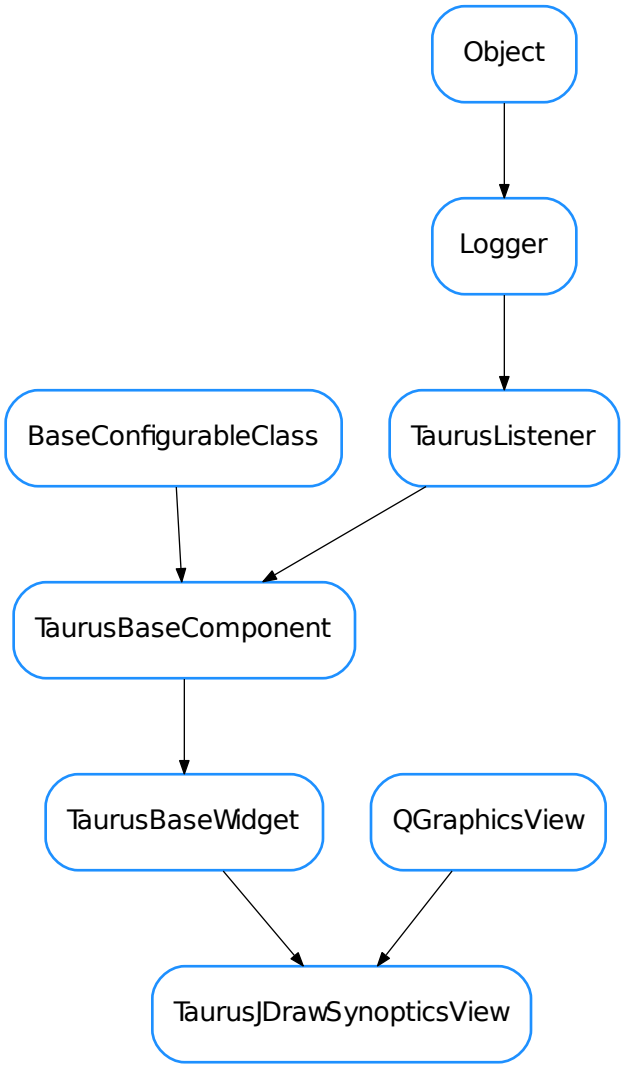
resetZBufferLevel()

setZBufferLevel(*level*)

set_common_params(*item*, *params*)

set_item_filling(*item*, *pattern=5*, *expand=False*)

TaurusJDrawSynopticsView



class TaurusJDrawSynopticsView(*parent=None*, *designMode=False*, *updateMode=None*, *alias=None*, *resizable=True*, *panelClass=None*)

Bases: PyQt4.QtGui.QGraphicsView, taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget

Taurus Class that visualizes Synoptics drawn with the JDraw tool (by ESRF). It is equivalent to ATK Synoptic Player (Java).

After initialization call setModel('/your/file.jdw') to parse the synoptic file and connect to controlled objects.

Arguments to TaurusJDrawSynopticsView() creator are:

- •designMode; used by Qt Designer
- •updateMode; controls Qt Viewport refresh (disabled by default)
- •alias; a dictionary of name replacements to be applied on graphical objects
- •resizable: whether to allow resizing or not
- •panelClass: class object, class name or shell command to be shown when an object is clicked (None will show default panel, '' or 'noPanel' will disable it)

TaurusJDrawSynopticsView and TaurusGraphicsScene signals/slots

External events:

```
Slot selectGraphicItem(const QString &) displays a selection
mark around the TaurusGraphicsItem that matches the argument passed.
```

Mouse Left-button events:

```
Signal graphicItemSelected(QString) is triggered, passing the
selected TaurusGraphicsItem.name() as argument.
```

Mouse Right-button events:

```
TaurusGraphicsItem.setContextMenu([(ActionName,ActionMethod(device_name))]
allows to configure custom context menus for graphic items using a list
of tuples. Empty tuples will insert separators in the menu.
```

closeEvent(*event=None*)

classmethod defaultPanelClass(*klass*)
This method assigns the Class used to open new object panels on double-click (TaurusDevicePanel by default) If an string is used it can be either a Taurus class or an OS launcher

defineStyle()

emitColors()
emit signal which is used to refresh the tree and colors of icons depend of the current status in jdrawSynoptic

fitting(*ADJUST_FRAME=False*)
Parent size is the size of the bigger panel (desn't keep ratio) Rect size never changes (fixed by the graphics objects) Size and SizeHint move one around the other

the method works well until an object is clicked, then the whole reference changes and doesn't work again.

getFramed()

getGraphicsFactory(*delayed=False*)

getModel()

getModelMimeData()
Used for drag events

classmethod **getQtDesignerPluginInfo**()

getSelectionStyle()

getSelectionStyleName()

get_device_list()

get_item_colors(*emit=False*)

get_item_list()

get_sizes()

graphicItemSelected

graphicSceneClicked

isReadOnly()

itemsChanged

model

modelsChanged

mousePressEvent(*event*)
Records last event position to use it for DragEvents

openJDraw()

panelClass()

refreshModel()

repaint()

resetSelectionStyle()

resizable()

resizeEvent(*event*)
It has been needed to reimplent size policies

selectGraphicItem

selectionStyle

setAlias(*alias*)
Assigning a dictionary like {'Tag':'Value'} with tags to be replaced in object names while parsing.

classmethod **setDefaultPanelClass**(*klass*, *other*)
This method returns the Class used to open new object panels on double-click (TaurusDevicePanel by default)

setModel

setModels()
This method triggers item.setModel(item._name) in all internal items.

setPanelClass(*widget*)

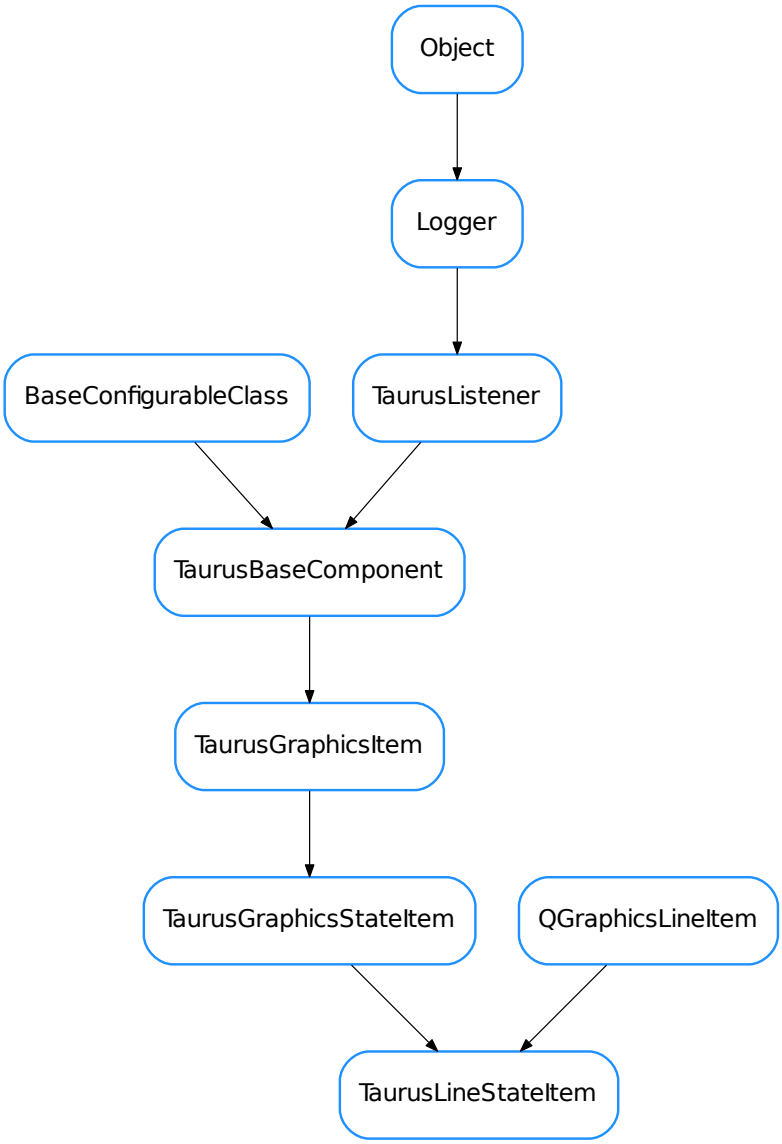
setResizable(*resizable*)

setSelectionStyle(*selectionStyle*)

update()

updateStyle()

TaurusLineStateItem

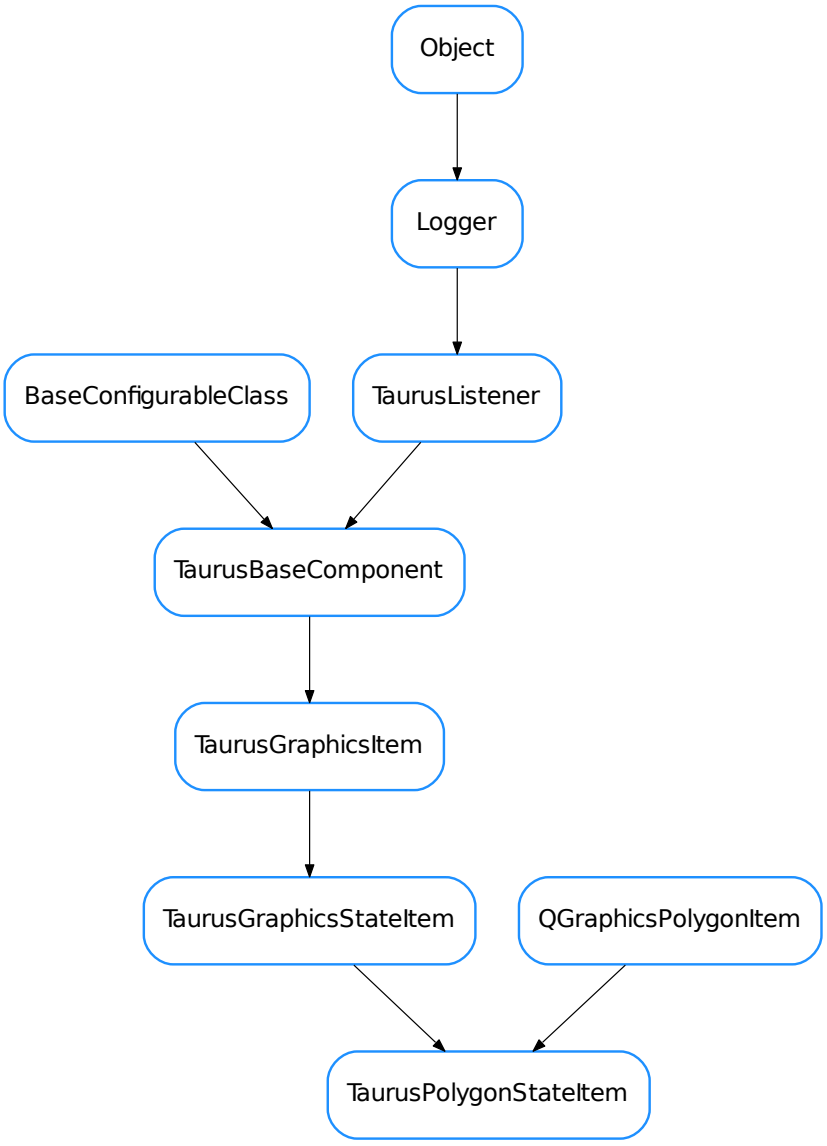


class **TaurusLineStateItem**(*name=None*, *parent=None*, *scene=None*)

Bases: `PyQt4.QtGui.QGraphicsLineItem`, `taurus.qt.qtgui.graphic.taurusgraphic.TaurusGraphicsStateItem`

paint(*painter*, *option*, *widget*)

TaurusPolygonStateItem

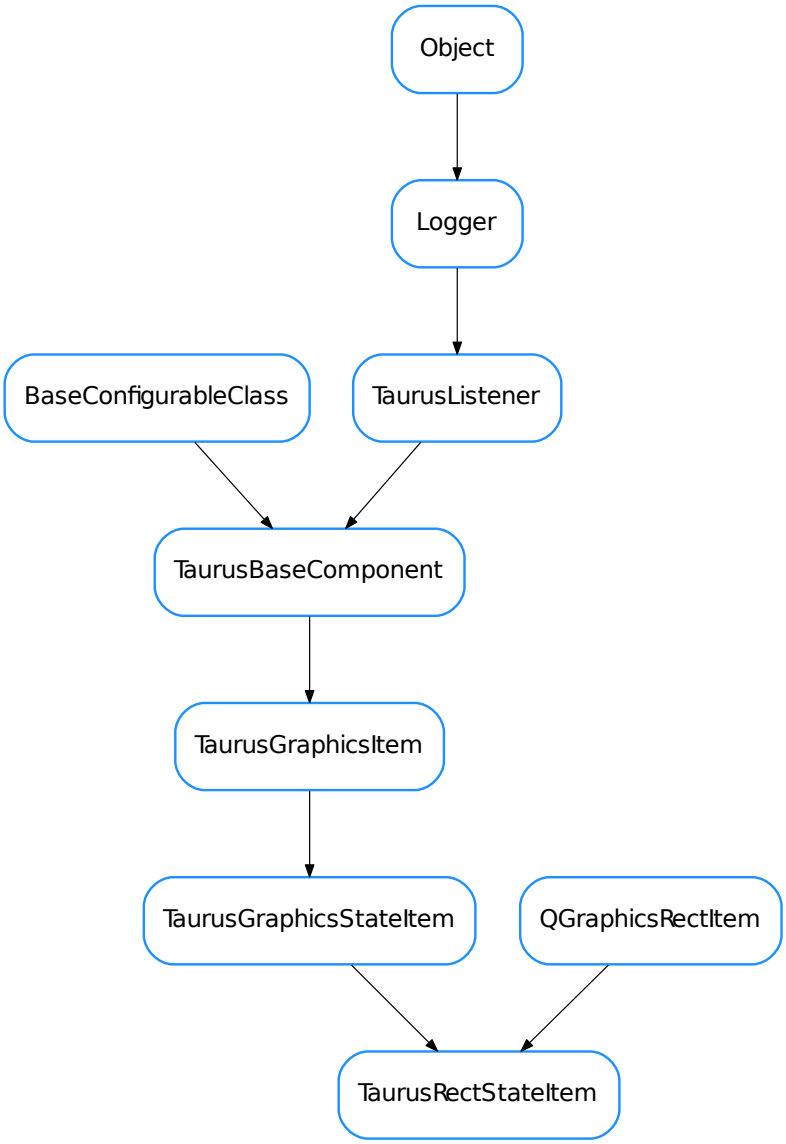


class TaurusPolygonStateItem(*name=None*, *parent=None*, *scene=None*)

Bases: PyQt4.QtGui.QGraphicsPolygonItem, taurus.qt.qtgui.graphic.taurusgraphic.TaurusGraphicsStateItem

paint(*painter*, *option*, *widget*)

TaurusRectStateItem

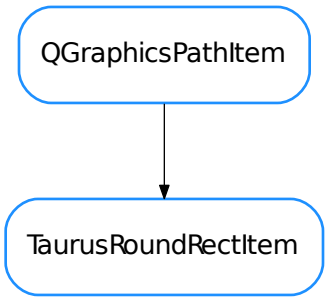


class **TaurusRectStateItem**(*name=None*, *parent=None*, *scene=None*)

Bases: `PyQt4.QtGui.QGraphicsRectItem`, `taurus.qt.qtgui.graphic.taurusgraphic.TaurusGraphicsStateItem`

paint(*painter*, *option*, *widget*)

TaurusRoundRectItem



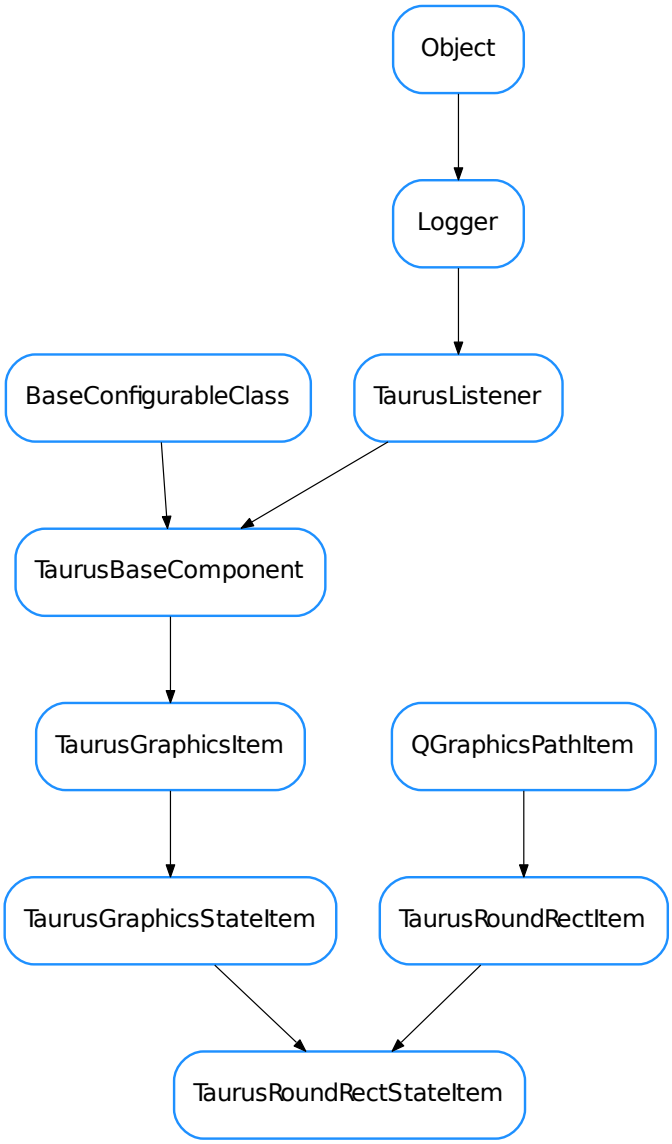
class TaurusRoundRectItem(*name=None*, *parent=None*, *scene=None*)

Bases: PyQt4.QtGui.QGraphicsPathItem

setCornerWidth(*width*, *nbPoints*)

setRect(*x*, *y*, *width*, *height*)

TaurusRoundRectStateItem

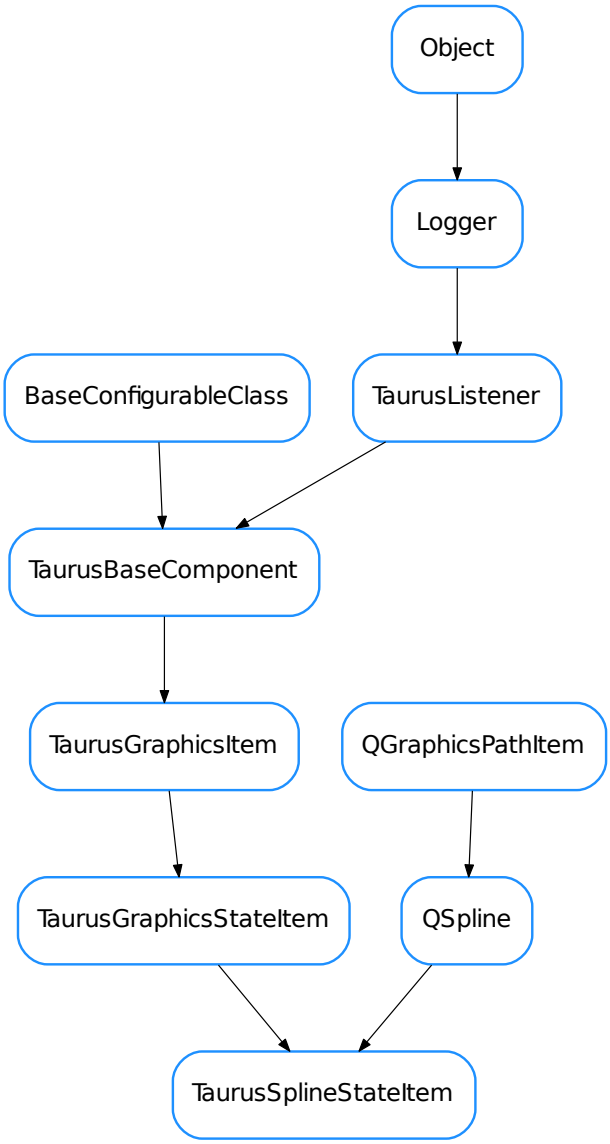


class **TaurusRoundRectStateItem**(*name=None*, *parent=None*, *scene=None*)

Bases: `taurus.qt.qtgui.graphic.taurusgraphic.TaurusRoundRectItem`, `taurus.qt.qtgui.graphic.taurusgraphic.TaurusGraphicsStateItem`

paint(*painter*, *option*, *widget*)

TaurusSplineStateItem

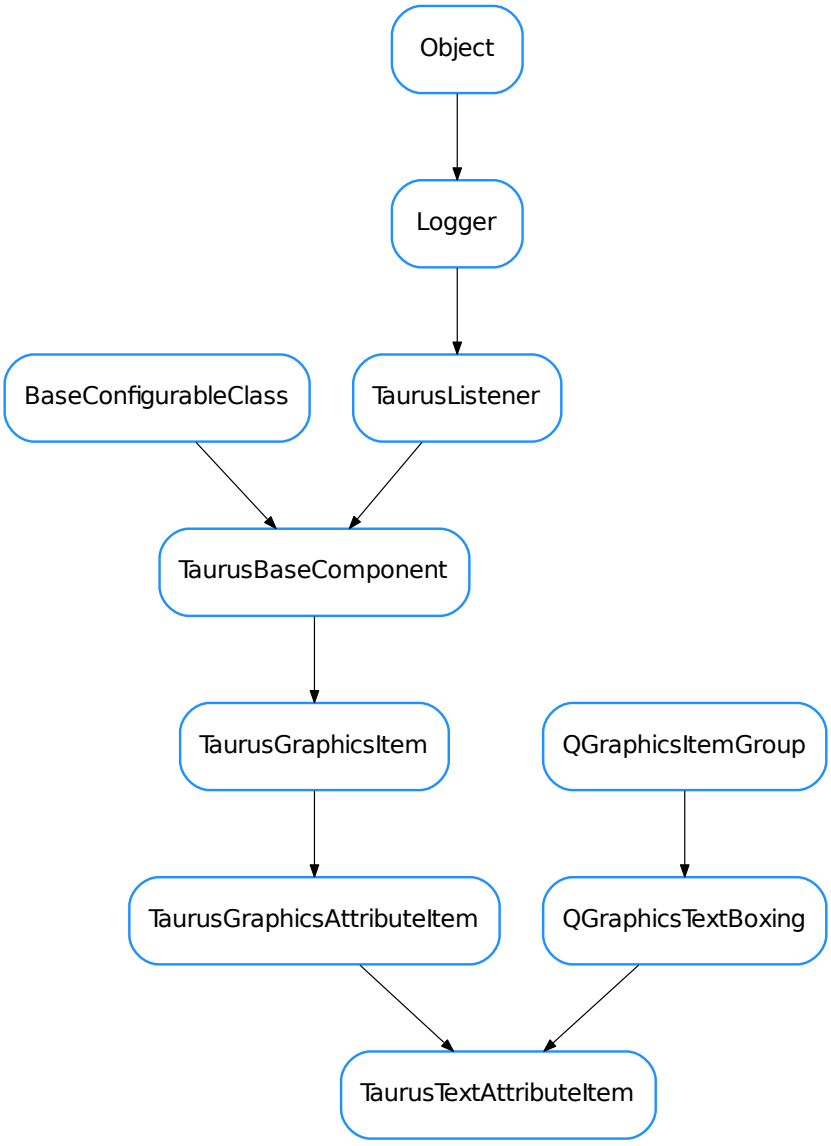


class TaurusSplineStateItem(*name=None*, *parent=None*, *scene=None*)

Bases: `taurus.qt.qtgui.graphic.taurusgraphic.QSpline`, `taurus.qt.qtgui.graphic.taurusgraphic.TaurusGraphicsStateItem`

paint(*painter*, *option*, *widget*)

TaurusTextAttributeItem



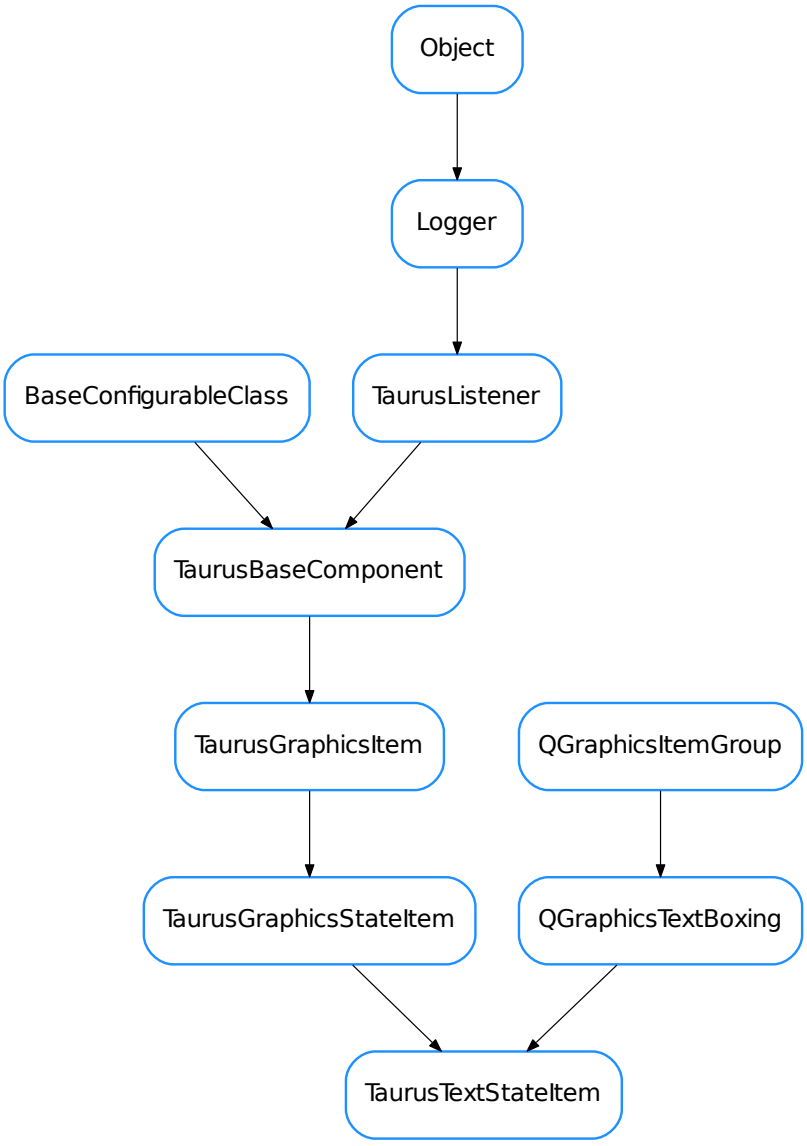
class `TaurusTextAttributeItem`(*name=None*, *parent=None*, *scene=None*)

Bases: `taurus.qt.qtgui.graphic.taurusgraphic.QGraphicsTextBoxing`, `taurus.qt.qtgui.graphic.taurusgraphic.TaurusGraphicsAttributeItem`

A QGraphicsItem that represents a text related to an attribute value

paint(*painter*, *option*, *widget*)

TaurusTextStateItem



class **TaurusTextStateItem**(*name=None*, *parent=None*, *scene=None*)

Bases: `taurus.qt.qtgui.graphic.taurusgraphic.QGraphicsTextBoxing`, `taurus.qt.qtgui.graphic.taurusgraphic.TaurusGraphicsStateItem`

A QGraphicsItem that represents a text related to a device state or attribute quality

paint(*painter*, *option*, *widget*)

- *QEmitter*
- *QGraphicsTextBoxing*

- *QSpline*
- *TaurusBaseGraphicsFactory*
- *TaurusEllipseStateItem*
- *TaurusGraphicsAttributeItem*
- *TaurusGraphicsItem*
- *TaurusGraphicsScene*
- *TaurusGraphicsStateItem*
- *TaurusGraphicsUpdateThread*
- *TaurusGraphicsView*
- *TaurusGroupItem*
- *TaurusGroupStateItem*
- *TaurusJDrawGraphicsFactory*
- *TaurusJDrawSynopticsView*
- *TaurusLineStateItem*
- *TaurusPolygonStateItem*
- *TaurusRectStateItem*
- *TaurusRoundRectItem*
- *TaurusRoundRectStateItem*
- *TaurusSplineStateItem*
- *TaurusTextAttributeItem*
- *TaurusTextStateItem*

Functions

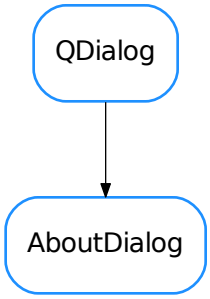
parseTangoUri(*name*)

taurus.qt.qtgui.help

This package contains a collection of taurus Qt widgets that provide a way to browse through the application help system

Classes

AboutDialog



class AboutDialog (*parent=None*)

Bases: PyQt4.QtGui.QDialog

Simple dialog to display typical About <application> dialog. It will create a Dialog with the title being *Dialog + <app name>* and a default text combining the application name and version, organization name and domain.

This behaviour can be changed by setting the dialog window title (`setWindowTitle()`) and content (*setText()*, *setHtml()*)

Example usage:

```
from taurus.external.qt import Qt
from taurus.qt.qtgui.help import AboutDialog

app = Qt.QApplication([])
app.setApplicationName("Example GUI")
app.setApplicationVersion("1.2.3")
app.setOrganizationName("Taurus")
app.setOrganizationDomain("http://www.taurus-scada.org/")
about_dialog = AboutDialog()
pixmap = Qt.QIcon.fromTheme("folder-open").pixmap(64, 64)
about_dialog.setPixmap(pixmap)
about_dialog.exec_()
```

getHtml()

Gets the current dialog HTML text.

Returns the current dialog HTML text.

Return type str

getPixmap()

Gets the current pixmap.

Returns the current dialog pixmap

Return type Qt.QPixmap

classmethod getQtDesignerPluginInfo()

getSource()

Gets the current dialog document source.

Returns the current dialog document source.

Return type Qt.QUrl

html

This property holds the current dialog HTML

Access functions:

- *getHtml()*
- *setHtml()*
- *resetHtml()*

loadUi (*filename=None*, *path=None*)

pixmap

This property holds the current dialog pixmap

Access functions:

- *getPixmap()*

- *setPixmap()*
- *resetPixmap()*

resetHtml()
Resets the dialog HTML to an empty HTML document

resetPixmap()
Resets the dialog pixmap to a Null pixmap.

setHtml

setPixmap

setSource

setText(*text*)
Sets the dialog text.

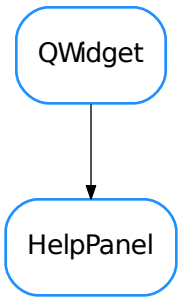
Parameters **text** (*str*) – new text

source
This property holds the current dialog document source

Access functions:

- *getSource()*
- *setSource()*

HelpPanel



class HelpPanel(*collection_file=None*, *parent=None*)
Bases: PyQt4.QtGui.QWidget

Simple widget to display application help system. Usage:

```
from taurus.external.qt import Qt
from taurus.qt.qtgui.help import HelpPanel

app = Qt.QApplication([])
help_panel = HelpPanel()
```

```
help_panel.setCollectionFile("help_file.qhc")
help_panel.show()
app.exec_()
```

collectionFile
This property holds the current collection file name

Access functions:

- *HelpPanel.getCollectionFile()*
- *HelpPanel.setCollectionFile()*
- *HelpPanel.resetCollectionFile()*

getCollectionFile()
Returns the name of the current collection file or empty string if no collection file is active

Returns the name of the current collection file

Return type str

classmethod getQtDesignerPluginInfo()

resetCollectionFile()
Resets the collection file

setCollectionFile

- *AboutDialog*
- *HelpPanel*

Functions

Assistant(*collection_file*, *auto_create=True*, *parent=None*)
The *Assistant()* will create a subprocess displaying the help system for the given QtHelp collection file (.qhc). Example usage:

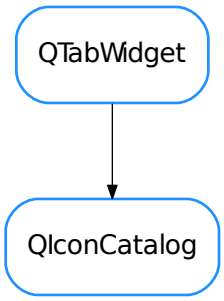
```
from taurus.external.qt import Qt
from taurus.qt.qtgui.help import Assistant

app = Qt.QApplication([])
qas = Assistant("my_app_help.qhc")
qas.start()
app.exec_()
```

taurus.qt.qtgui.icon

Utilities for using the bundled icons in Taurus and for registering external sources of icons.

Classes

QIconCatalog



class QIconCatalog (*parent=None*)

Bases: PyQt4.QtGui.QTabWidget

A widget that shows a tab for each registered search path prefix. In each tab, all icons available for the corresponding prefix are displayed. Clicking on an icon provides info on how to use it from a taurus application.

iconSelected

- *QIconCatalog*

Functions

getCachedPixmap (*key*, *size=None*)

Returns a PyQt4.QtGui.QPixmap object for the given key and size. The key argument supports QDir's searchPath prefixes (see QDir.setSearchPaths()).

Parameters

- **key** (str) – the pixmap key., e.g.: 'status:folder-open.svg'
- **size** (int) – the pixmap size in pixels (will get a square pixmap). If None is passed it will return the original size

Return type QPixmap

Returns

getDevStateIcon (*state*, *fallback=None*)

Gets a PyQt4.QtGui.QIcon object for the given taurus.core.taurusbasetypes.TaurusDevState.

If an icon cannot be found for the given state, fallback is returned.

Parameters

- **state** (TaurusDevState) – the taurus device state
- **fallback** (QIcon) – the fallback icon. Default is None.

Return type QIcon or None

Returns

getDevStatePixmap(*state*, *size=None*)

Gets a PyQt4.QtGui.QPixmap object for the given `taurus.core.taurusbasetypes.TaurusDevState`.

Parameters

- **state** (`TaurusDevState`) – the taurus software device state
- **size** (`int`) – the pixmap size in pixels (will get a square pixmap). Default is None meaning it will return the original size.

Return type `QPixmap` or `None`

Returns

getDevStateToolTip(*state*)

getElementTypeIcon(*elemType*, *fallback=None*)

Gets a PyQt4.QtGui.QIcon object for the given `taurus.core.taurusbasetypes.TaurusElementType`.

If an icon cannot be found for the given TaurusElementType, fallback is returned.

Parameters

- **elemType** (`TaurusElementType`) – the taurus element type
- **fallback** (`QIcon`) – the fallback icon. Default is None.

Return type `QIcon`

Returns

getElementTypeIconName(*elemType*)

Gets an icon name string for the given `taurus.core.taurusbasetypes.TaurusElementType`.

If an icon name cannot be found for elemType, None is returned.

Parameters **elemType** (`TaurusElementType`) – the taurus element type

Return type `str`

Returns a string representing the icon name for the given `taurus.core.taurusbasetypes.TaurusElementType`

getElementTypePixmap(*elemType*, *size=None*)

Gets a PyQt4.QtGui.QPixmap object for the given `taurus.core.taurusbasetypes.TaurusElementType`.

Parameters

- **elemType** (`TaurusElementType`) – the taurus element type
- **size** (`int`) – the pixmap size in pixels (will get a square pixmap). Default is None meaning it will return the original size.

Return type `QPixmap` or `None`

Returns

getElementTypeSize(*elemType*)

getElementTypeToolTip(*elemType*)

getStandardIcon(*key*, *widget=None*)
Returns a PyQt4.QtGui.QIcon object for the given key. Key should be a QStyle.StandardPixmap enumeration member. The widget argument is optional and can also be used to aid the determination of the icon.

Parameters

- **key** (StandardPixmap) – a standard pixmap which can follow some existing GUI style or guideline
- **widget** (QWidget) – the widget argument (optional) can also be used to aid the determination of the icon.

Return type QIcon

Returns

registerPathFiles(*pathfilenames*)
Use given .path files to update Qt's search path Each path file contains a json-encoded list of (prefix,path) tuples. This function will call Qt.QDir.addSearchPath with each of the tuples from the path files (prefix values will be sanitized first, and relative path values will be made relative to the dir containing the .path file)

Parameters **pathfilenames** (list <str>) – list of .path file names

registerTheme(*name='Tango'*, *path=''*, *force=False*)
Use bundled them if OS does not define a theme (non-X11 systems)

Parameters

- **name** (str) – icon theme name (default=Tango)
- **path** (str) – path to dir containing the theme (absolute or relative to dir of taurus.qt.qtgui.icon). Default = ''
- **force** (bool) – Force to set path even if a theme is already set

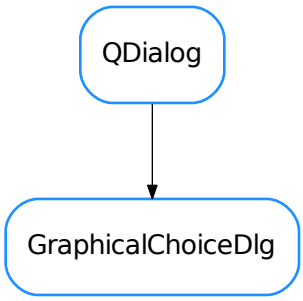
sanitizePrefix(*prefix*)
strips any leading '/' and substitutes non alphanumeric characters by '_'

taurus.qt.qtgui.input

This package contains a collection of taurus Qt widgets that typically interact with the user. Examples are line edits, comboboxes and checkboxes

Classes

GraphicalChoiceDlg



class **GraphicalChoiceDlg**(*parent=None*, *designMode=False*, *choices=None*, *pixmaps=None*, *iconSize=128*, *defaultPixmap=None*, *horizontalScrollBarPolicy=0*, *verticalScrollBarPolicy=0*)

Bases: `PyQt4.QtGui.QDialog`

A generic dialog for choosing among a set of choices which are presented as an array of, each with a given pixmap.

The *`getChoice()`* static method is provided for convenience so that the dialog can be invoked wit a single line:

```
chosen,ok = GraphicalChoiceDlg.getChoice(parent, title, msg, choices, pixmaps,␣
↪size, defpixmap, horizontalScrollBarPolicy, verticalScrollBarPolicy)
```

static **getChoice**(*parent=None*, *title=''*, *msg=''*, *choices=None*, *pixmaps=None*, *iconSize=128*, *defaultPixmap=None*, *horizontalScrollBarPolicy=0*, *verticalScrollBarPolicy=0*)

Static method which launches a GraphicalChoiceDlg with the given options and returns the result

Parameters

- **parent** (`QWidget`) – The parent of the dialog (it will be centered on it)
- **title** (`str`) – the text which is displayed in the title bar of the dialog
- **msg** (`str`) – the text which is shown to the user in the dialog, above the choices.
- **choices** (`list <list>`) – a list of lists of strings to be used as choices names. The (possibly sparse) 2D array defined by the nested lists will be used to present the choices in a grid. The choice names will be used as keys for pixmaps
- **pixmaps** (`dict <str, QPixmap>`) – dictionary mapping the choices text to corresponding pixmap. If no valid pixmap is provided for a given choice, the defaultPixmap will be used
- **iconSize** (`int`) – size of the icons to be displayed (128px by default)
- **defaultPixmap** (`QPixmap`) – Default Pixmap to use if none passed for a given choice. No Pixmap will be used if None passed.
- **horizontalScrollBarPolicy** (`ScrollBarPolicy`) – defines the mode of the horizontal scroll bar. The default mode is ScrollBarAsNeeded.

- **verticalScrollBarPolicy** (ScrollBarPolicy) – defines the mode of the vertical scroll bar. The default mode is ScrollBarAsNeeded

Return type tuple <str, bool>

Returns A tuple containing choice,ok. choice is the name of the chosen option. ok is true if the user pressed OK and false if the user pressed Cancel.

getChosen()
returns the choice :rtype: str :return:

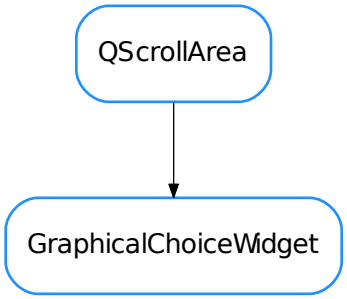
onChoiceMade(*chosen*)
slot called when the user chooses an option

setHorizontalScrollBarPolicy(*policy*)
sets horizontal scrollbar policy of scrollArea

setMessage(*msg*)
sets the text which is shown to the user in the dialog

setVerticalScrollBarPolicy(*policy*)
sets vertical scrollbar policy of scrollArea

GraphicalChoiceWidget



class GraphicalChoiceWidget(*parent=None*, *designMode=False*, *choices=None*, *pixmaps=None*, *iconSize=128*, *defaultPixmap=None*, *horizontalScrollBarPolicy=0*, *verticalScrollBarPolicy=0*)
Bases: PyQt4.QtGui.QScrollArea

A widget that presents a 2D grid of buttons

choiceMade

getChosen()
returns the choice :rtype: str :return:

classmethod getQtDesignerPluginInfo()
Returns pertinent information in order to be able to build a valid QtDesigner widget plugin

The dictionary returned by this method should contain *at least* the following keys and values: - 'module' : a string representing the full python module name (ex.: 'taurus.qt.qtgui.base') - 'icon' : a string representing

valid resource icon (ex.: 'designer:combobox.png') - 'container' : a bool telling if this widget is a container widget or not.

This default implementation returns the following dictionary:

```
{ 'group'     : 'Taurus Widgets',
  'icon'      : 'logos:taurus.png',
  'container' : False }
```

Return type dict

Returns a map with pertinent designer information

onClick()
slot called when a button is clicked

setChoice(*row*, *col*, *text*, *pixmap=None*, *tooltip=None*)
sets the option for a given row,column coordinate in the grid

Parameters

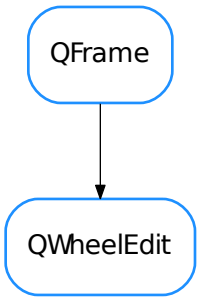
- **row** (int) – row in the grid for this option
- **col** (int) – column in the grid for this option
- **text** (str) – name for this option
- **pixmap** (QPixmap or None) – If no valid pixmap is provided for a given choice, the default one will be used
- **tooltip** (str) – tooltip for this option (if None given, the *text* is used)

setChoices(*choices*, *pixmaps=None*)
sets the available options

Parameters

- **choices** (list <list>) – a list of lists of strings to be used as choices names. The (possibly sparse) 2D array defined by the nested lists will be used to present the choices in a grid. The choice names will be used as keys for pixmaps
- **pixmaps** (dict <str, QPixmap>) – dictionary mapping the choices text to corresponding pixmap. If no valid pixmap is provided for a given choice, a default pixmap will be used

QWheelEdit



class QWheelEdit (*parent=None*)

Bases: PyQt4.QtGui.QFrame

A widget designed to handle numeric scalar values. It allows interaction based on single digit as well as normal value edition.

DefaultDecDigitCount = 2

DefaultIntDigitCount = 6

autoRepeat

autoRepeatDelay

autoRepeatInterval

buttonPressed (*self*, *b*) → None

Slot executed when an arrow button is pressed from the button group

@param[in] b (_ArrowButton) the button which was pressed

clearWarning (*self*) → None

Clears the warning style. If not in warning mode, nothing is done.

decimalDigits

editingFinished (*self*) → None

Slot called when the user finishes editing

getAutoRepeat ()

getAutoRepeatDelay ()

getAutoRepeatInterval ()

getDecDigitCount (*self*) → int

Gets the number of decimal digits this widget displays

@return (int) the number of decimal digits this widget displays

getDigitCount (*self*) → int

Gets the total number of digits this widget displays

@return (int) the total number of digits this widget displays

getEditWidget (*self*) → QWidget
Gets the widget object used when the user manually sets the value

@return (QWidget) the widget used for editing

getIntDigitCount (*self*) → int
Gets the number of integer digits this widget displays

@return (int) the number of integer digits this widget displays

getMaxValue (*self*) → float
Gets the maximum allowed value

@return (float) the maximum allowed value

getMinValue (*self*) → float
Gets the minimum allowed value

@return (float) the minimum allowed value

getPreviousValue (*self*) → float
Gives the previous value of this widget

@return (float) the previous value of this widget

getShowArrowButtons ()

getValue (*self*) → float
Gets the current value of this widget

@return (float) the value currently displayed by the widget

getValueStr (*self*) → str
Gets the current value string of this widget

@return (str) the value currently displayed by the widget

hideEditWidget (*self*) → None
Forces the edition widget to be hidden

integerDigits

keyPressEvent (*self*, *key_event*) → None
Exectuted when the user presses a key. F2 enters/leaves edition mode. ESC leaves edition mode

maxValue

minValue

mouseDoubleClickEvent (*self*, *mouse_event*)
Executed when user presses double click. This widget shows the edition widget when this happens

numberChanged

numberEdited

resetAutoRepeat ()

resetDecDigitCount (*self*) → None
Resets the number of decimal digits this widget displays to DefaultDecDigitCount

resetIntDigitCount (*self*) → None
Resets the number of integer digits this widget displays to DefaultIntDigitCount

resetMaxValue (*self*) → None
Resets the maximum allowed value to the maximum possible according to the current total number of digits

resetMinValue (*self*) → None
Resets the minimum allowed value to the minimum possible according to the current total number of digits

resetShowArrowButtons ()

resetValue (*self*) → None
Resets the value of this widget to 0.0

returnPressed

setAutoRepeat (*v*)

setAutoRepeatDelay (*milisecs*)

setAutoRepeatInterval (*milisecs*)

setDecDigitCount (*self*, *n*) → None
Sets the number of decimal digits this widget displays

@param[in] n (int) the number of decimal digits to display

setDigitCount (*self*, *int_nb*, *dec_nb*) → None
Updates the displayed digits.

@param[in] int_nb(int) number of integer digits @param[in] dec_nb(int) number of decimal digits

setIntDigitCount (*self*, *n*) → None
Sets the number of integer digits this widget displays

@param[in] n (int) the number of integer digits to display

setMaxValue (*self*, *v*) → None
Sets the maximum allowed value for the widget

@param[in] v (float) the new maximum allowed value

setMinValue (*self*, *v*) → None
Sets the minimum allowed value for the widget

@param[in] v (float) the new minimum allowed value

setRoundFunc (*self*, *roundFunc*) → None
Sets the rounding function to use when calling _setValue(). This allows you to filter invalid user input

@param[in] roundFunc (callable) the rounding function to use

setShowArrowButtons (*yesno*)

setValue (*self*, *v*) → None
Sets the value of this widget. Send a 'valueChanged(double)' Qt signal

@param[in] v (float/Quantity) the value to be set

setWarning (*self*, *msg*) → None
Activates the warning style for this widget. This means a violet border and a tooltip with the given message.

@param[in] msg (str) the message to be displayed as tooltip

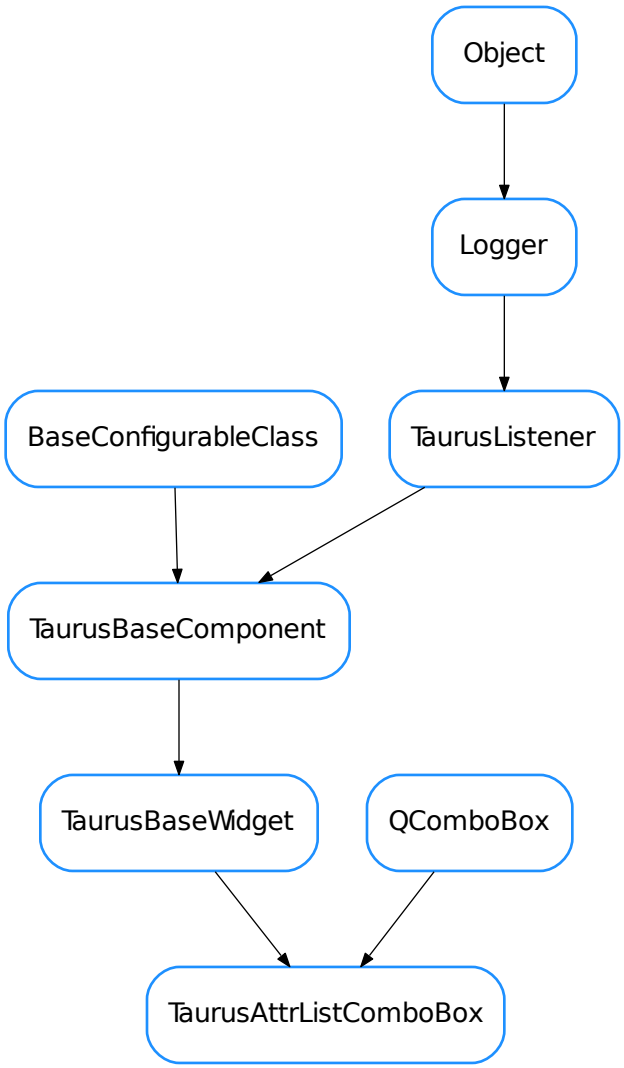
showArrowButtons

showEditWidget (*self*) → None
Forces the edition widget to be displayed

value

wheelEvent (*evt*)

TaurusAttrListComboBox



class TaurusAttrListComboBox(*parent=None*, *designMode=False*)

Bases: `PyQt4.QtGui.QComboBox`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget`

Combobox whose items reflect the items read from a 1D attribute of dtype str

defineStyle()

Defines the initial style for the widget

getModelClass()

reimplemented from `TaurusBaseWidget`

classmethod getQtDesignerPluginInfo()

reimplemented from `TaurusBaseWidget`

handleEvent(*evt_src*, *evt_type*, *evt_value*)
reimplemented from `TaurusBaseWidget`

model

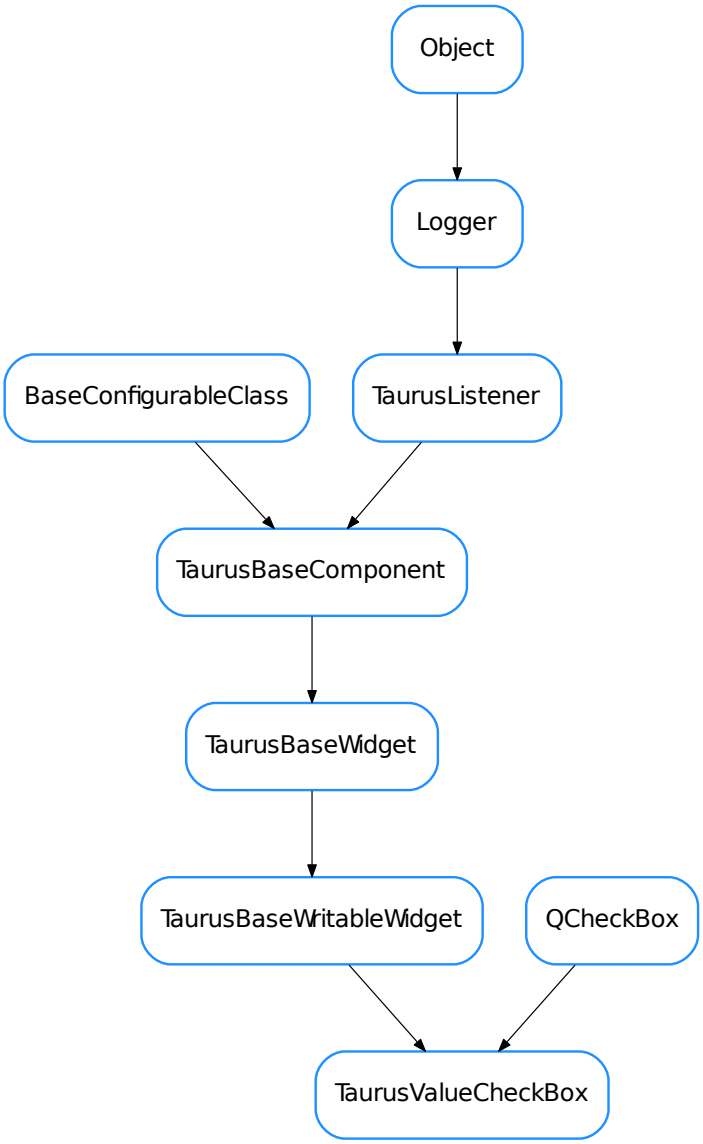
setModel(*m*)
reimplemented from `TaurusBaseWidget`

setQModel(**args*, ***kwargs*)
access to `QAbstractItemView.setModel()`

updateStyle()
reimplemented from `TaurusBaseWidget`

useParentModel

TaurusValueCheckBox



class TaurusValueCheckBox(*qt_parent=None*, *designMode=False*)

Bases: `PyQt4.QtGui.QCheckBox`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseWritableWidget`

A QCheckBox connected to a boolean writable attribute model

autoApply

forcedApply

classmethod **getQtDesignerPluginInfo**()

getValue()

keyPressEvent(*event*)

minimumSizeHint()

model

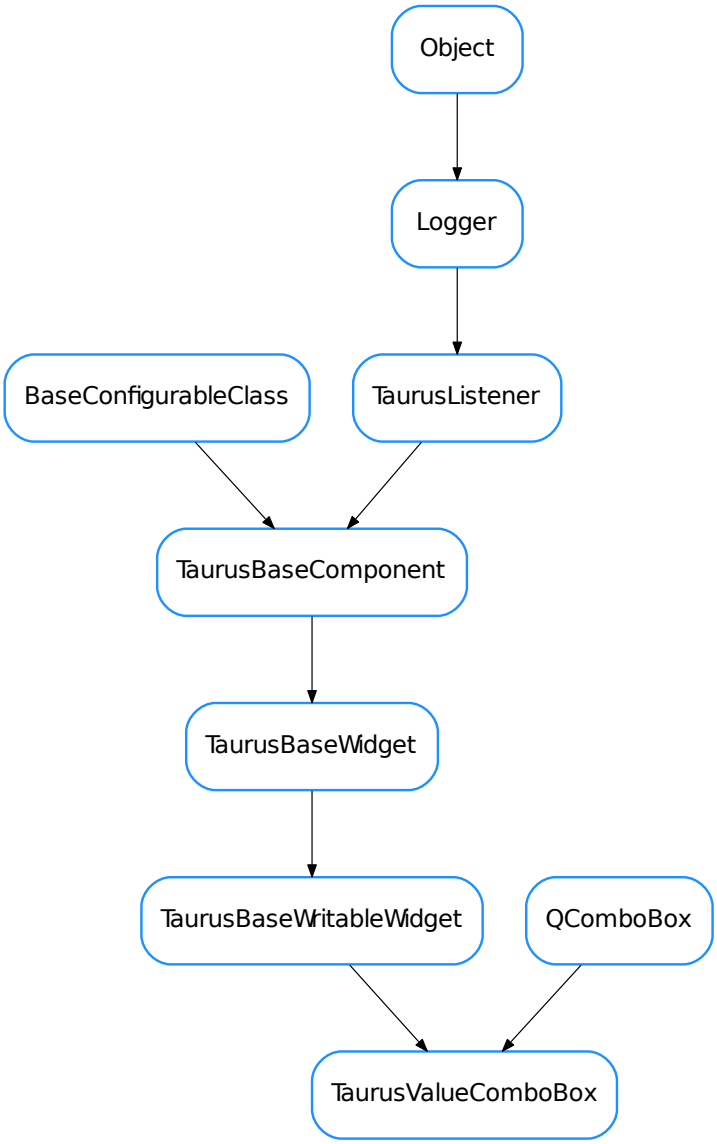
setValue(*v*)

showText

updateStyle()

useParentModel

TaurusValueComboBox



class TaurusValueComboBox(*parent=None*, *designMode=False*)

Bases: `PyQt4.QtGui.QComboBox`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseWritableWidget`

This widget shows a combobox that offers a limited choice of values that can be set on an attribute.

addValueNames(*names*)

Add new value-name associations to the combobox.

... seealso: *setValueNames()*

Parameters **names** (sequence <tuple>) – A sequence of (name,value) tuples, where each attribute value gets a name for display

autoApply

forcedApply

classmethod **getQtDesignerPluginInfo**()
reimplemented from TaurusBaseWritableWidget

getValue()
Get the value that the widget is displaying now, not the value of the attribute.

getValueString(*value*, *default='UNKNOWN(%s)'*)
Returns the corresponding name in the combobox out of a value (or a default value if not found).

Parameters

- **value** – value to look up
- **default** (str) – value in case it is not found. It accepts a '%s' placeholder which will be substituted with str(value). It defaults to 'UNKNOWN(%s)'.

keyPressEvent(*event*)
reimplemented to emit an 'applied()' signal when Enter (or Return) key is pressed

model

postDetach()
reimplemented from TaurusBaseWritableWidget

preAttach()
reimplemented from TaurusBaseWritableWidget

setModel(*m*)
Reimplemented from TaurusBaseWritableWidget.setModel()

setQModel(**args*, ***kwargs*)
access to QCombobox.setModel()

setValue(*value*)
Set the value for the widget to display, not the value of the attribute.

setValueNames(*names*)
Sets the correspondence between the values to be applied and their associated text to show in the combobox.

Parameters **names** (sequence <tuple>) – A sequence of (name,value) tuples, where each attribute value gets a name for display

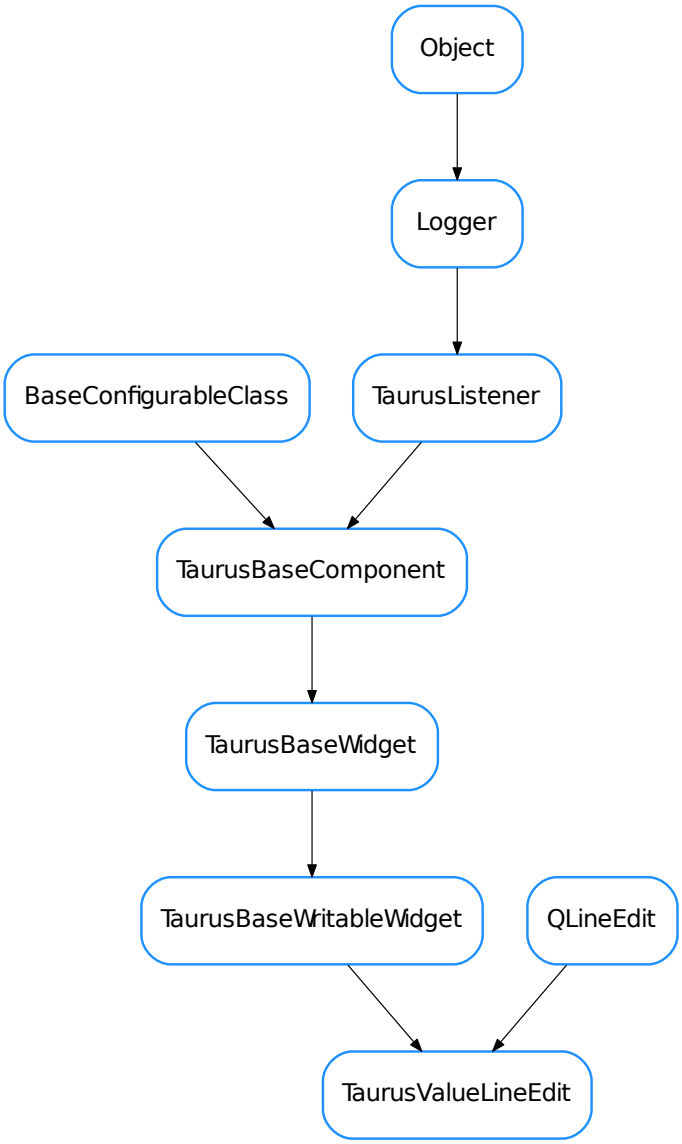
teachDisplayTranslationToWidget(*widget*, *default='UNKNOWN(%s)'*)
Makes a label object change the displayed text by the corresponding value of the combobox. This is implemented for the general case and may be not what you expect in some cases (as for example, it fires a fake periodic event which may be problematic if these are being filtered out).

updateStyle()
reimplemented from TaurusBaseWritableWidget

useParentModel

writeIndexValue

TaurusValueLineEdit



class TaurusValueLineEdit (*qt_parent=None*, *designMode=False*)

Bases: `PyQt4.QtGui.QLineEdit`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseWritableWidget`

autoApply

enableWheelEvent

forcedApply

getEnableWheelEvent()

classmethod getQtDesignerPluginInfo()

getValue()

handleEvent(*evt_src*, *evt_type*, *evt_value*)

isTextValid()
Validates current text

Return type bool

Returns Returns False if there is a validator and the current text is not Acceptable. Returns True otherwise.

keyPressEvent(*evt*)

model

notifyValueChanged(**args*)
reimplement to avoid autoapply on every partial edition

resetEnableWheelEvent()

setEnableWheelEvent(*b*)

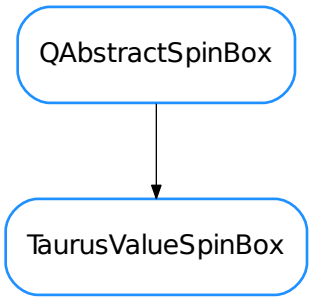
setValue(*v*)

updateStyle()

useParentModel

wheelEvent(*evt*)

TaurusValueSpinBox



class TaurusValueSpinBox(*qt_parent=None*, *designMode=False*)
Bases: PyQt4.QtGui.QAbstractSpinBox

autoApply

forcedApply

getAutoApply()

getForcedApply()

getModel()

classmethod getQtDesignerPluginInfo()

getSingleStep()

getUseParentModel()

getValue()

keyPressEvent(*evt*)

model

modelChanged

resetAutoApply()

resetForcedApply()

resetModel()

resetSingleStep()

resetUseParentModel()

setAutoApply(*model*)

setForcedApply(*model*)

setModel(*model*)

setSingleStep

setUseParentModel(*model*)

setValue(*v*)

singleStep

stepBy(*steps*)

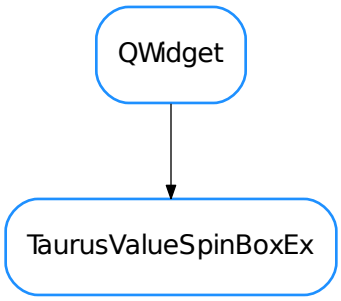
stepEnabled()

useParentModel

validate(*input*, *pos*)

Overloaded to use the current validator from the TaurusValueLineEdit, instead of the default QAbstractSpinBox validator. If no validator is set in the LineEdit, it falls back to the original behaviour

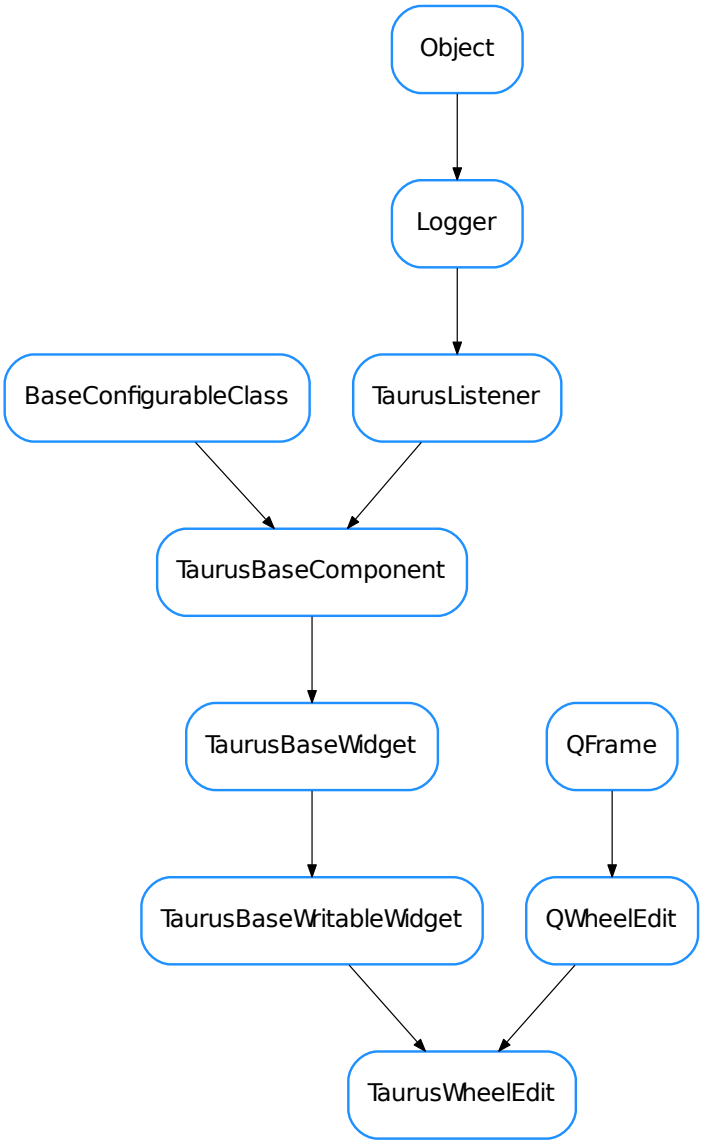
TaurusValueSpinBoxEx



class **TaurusValueSpinBoxEx**(*qt_parent=None*, *designMode=False*)

Bases: `PyQt4.QtGui.QWidget`

TaurusWheelEdit



class TaurusWheelEdit(*qt_parent=None*, *designMode=False*)

Bases: `taurus.qt.qtgui.input.qwheel.QWheelEdit`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseWritableWidget`

autoApply

forcedApply

classmethod getQtDesignerPluginInfo()

handleEvent(*evt_src*, *evt_type*, *evt_value*)

model

updateStyle()

useParentModel

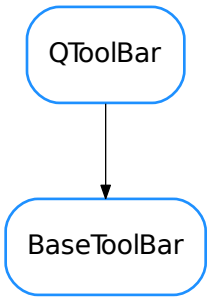
- *GraphicalChoiceDlg*
- *GraphicalChoiceWidget*
- *QWheelEdit*
- *TaurusAttrListComboBox*
- *TaurusValueCheckBox*
- *TaurusValueComboBox*
- *TaurusValueLineEdit*
- *TaurusValueSpinBox*
- *TaurusValueSpinBoxEx*
- *TaurusWheelEdit*

taurus.qt.qtgui.model

This package provides the set of base model widget classes.

Classes

BaseToolBar

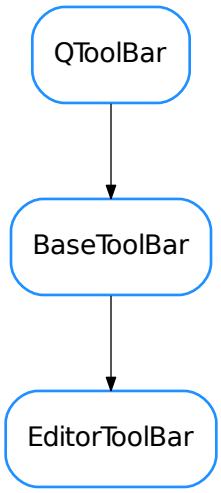


class BaseToolBar(*name=None*, *view=None*, *parent=None*, *designMode=False*)

Bases: PyQt4.QtGui.QToolBar

viewWidget()

EditorToolBar



class **EditorToolBar**(*view=None*, *parent=None*, *designMode=False*)

Bases: `taurus.qt.qtgui.model.qbasemodel.BaseToolBar`

Internal widget to be placed in a _QToolArea providing buttons for moving, adding and removing items from a view based widget

addTriggered

moveBottomTriggered

moveDownTriggered

moveTopTriggered

moveUpTriggered

onAdd()

onMoveBottom()

onMoveDown()

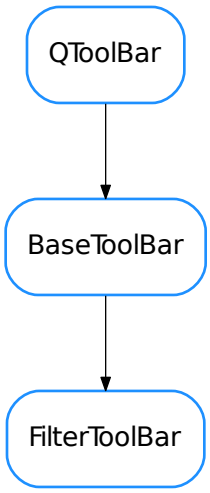
onMoveTop()

onMoveUp()

onRemove()

removeTriggered

FilterToolBar



class **FilterToolBar**(*view=None*, *parent=None*, *designMode=False*)

Bases: taurus.qt.qtgui.model.qbasemodel.BaseToolBar

Internal widget providing quick filter to be placed in a _QToolArea

clearFilterTriggered

filterChanged

filterEdited

getFilterLineEdit()

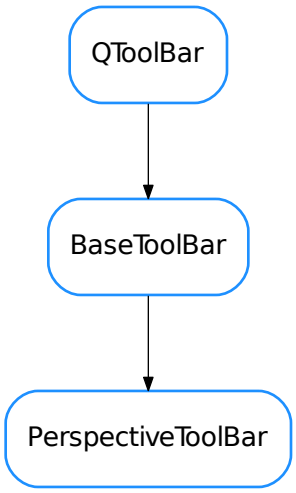
onClearFilter()

onFilterChanged(*text=None*)

onFilterEdited(*text=None*)

setFilterText(*text*)

PerspectiveToolBar



class **PerspectiveToolBar**(*perspective*, *view=None*, *parent=None*, *designMode=False*)

Bases: `taurus.qt.qtgui.model.qbasemodel.BaseToolBar`

onSwitchPerspective()

perspective()

perspectiveChanged

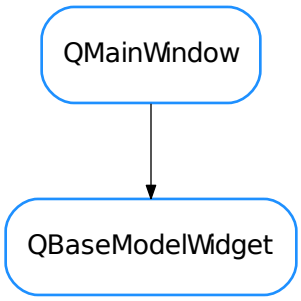
switchPerspectiveButton()

Returns the QToolButton that handles the switch perspective.

Return type `QToolButton`

Returns the switch perspective tool button

QBaseModelWidget



class QBaseModelWidget (*parent=None*, *designMode=False*, *with_filter_widget=True*, *with_selection_widget=True*, *with_refresh_widget=True*, *perspective=None*, *proxy=None*)

Bases: `PyQt4.QtGui.QMainWindow`

A pure Qt widget designed to display a Qt view widget (QTreeView for example), envolved by optional toolbar and statusbar. The Qt model associated with the internal Qt view widget should be a *taurus.qt.qtcore.model.TaurusBaseModel*

DftPerspective = None

KnownPerspectives = {}

addToolBar (*toolbar*)

createStatusBar ()

createToolArea ()

createViewWidget (*klass=None*)

currentItemChanged

getBaseQModel ()

getFilterBar ()

getPerspectiveBar ()

getQModel ()

getRefreshBar ()

getSelectionBar ()

insertToolBar (*before*, *toolbar*)

itemClicked

itemDoubleClicked

itemSelectionChanged

onClearSelection ()

onFilterChanged(*filter*)

onRefreshModel()

onSelectAll()

onSwitchPerspective(*perspective*)

perspective()

refresh()

selectedItems()
Returns a list of all selected non-hidden items

Return type list <TaurusTreeItem>

Returns

setQModel(*qmodel*)

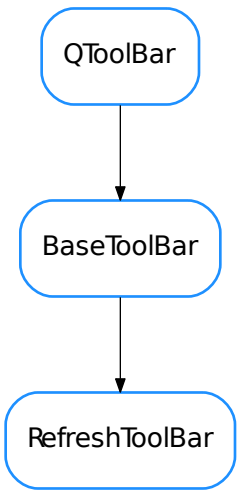
usesProxyQModel()

viewCurrentIndexChanged(*current*, *previous*)

viewSelectionChanged(*selected*, *deselected*)

viewWidget()

RefreshToolBar

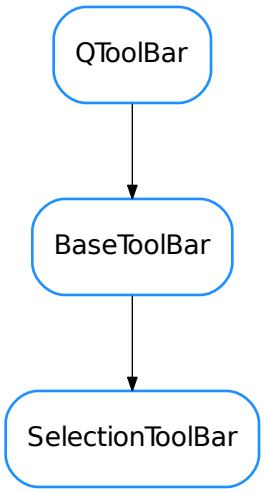


class RefreshToolBar(*view=None*, *parent=None*, *designMode=False*)
Bases: taurus.qt.qtgui.model.qbasemodel.BaseToolBar

onRefresh()

refreshTriggered

SelectionToolBar



class **SelectionToolBar**(*view=None*, *parent=None*, *designMode=False*)

Bases: taurus.qt.qtgui.model.qbasemodel.BaseToolBar

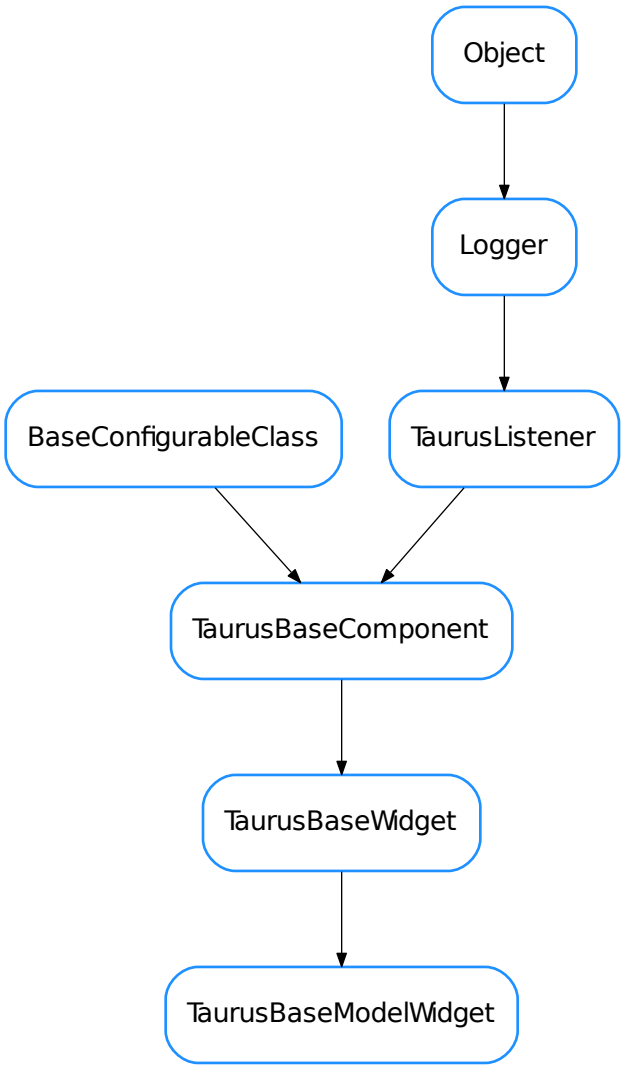
clearSelectionTriggered

onSelectAll()

onclearSelection()

selectAllTriggered

TaurusBaseModelWidget



class TaurusBaseModelWidget (*designMode=False*)

Bases: `taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget`

A class:*taurus.qt.qtgui.base.TaurusBaseWidget* that connects to a taurus model. It must be used together with class:*~taurus.qt.qtgui.base.QBaseModelWidget*

model

This property holds the unique URI string representing the model name with which this widget will get its data from. The convention used for the string can be found *here*.

In case the property `useParentModel` is set to True, the model text must start with a '/' followed by the attribute name.

Access functions:

- `TaurusBaseWidget.getModel()`
- `TaurusBaseWidget.setModel()`
- `TaurusBaseWidget.resetModel()`

See also:

Model concept

setModel(*m*)

- *BaseToolBar*
- *EditorToolBar*
- *FilterToolBar*
- *PerspectiveToolBar*
- *QBaseModelWidget*
- *RefreshToolBar*
- *SelectionToolBar*
- *TaurusBaseModelWidget*

taurus.qt.qtgui.panel

This package contains a collection of taurus Qt widgets representing various panels like forms or panels to be inserted in dialogs

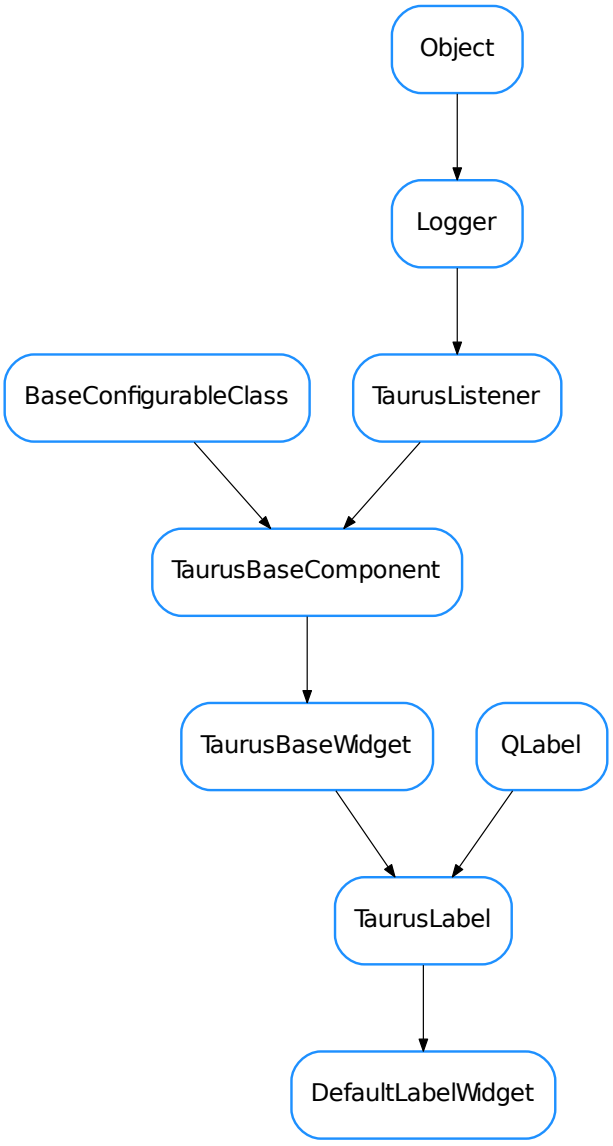
Modules

taurus.qt.qtgui.panel.report

__init__.py:

Classes

DefaultLabelWidget



class DefaultLabelWidget(**args*)

Bases: `taurus.qt.qtgui.display.tauruslabel.TaurusLabel`

The base class used by default for showing the label of a TaurusValue.

Note: It only makes sense to use this class as a part of a TaurusValue, since it assumes that it can get a reference to a TaurusValue via the `getTaurusValueBuddy()` member

---

contextMenuEvent(*event*)
The label widget will be used for handling the actions of the whole TaurusValue

see `QWidget.contextMenuEvent()`

getDisplayValue(*cache=True*, *fragmentName=None*)

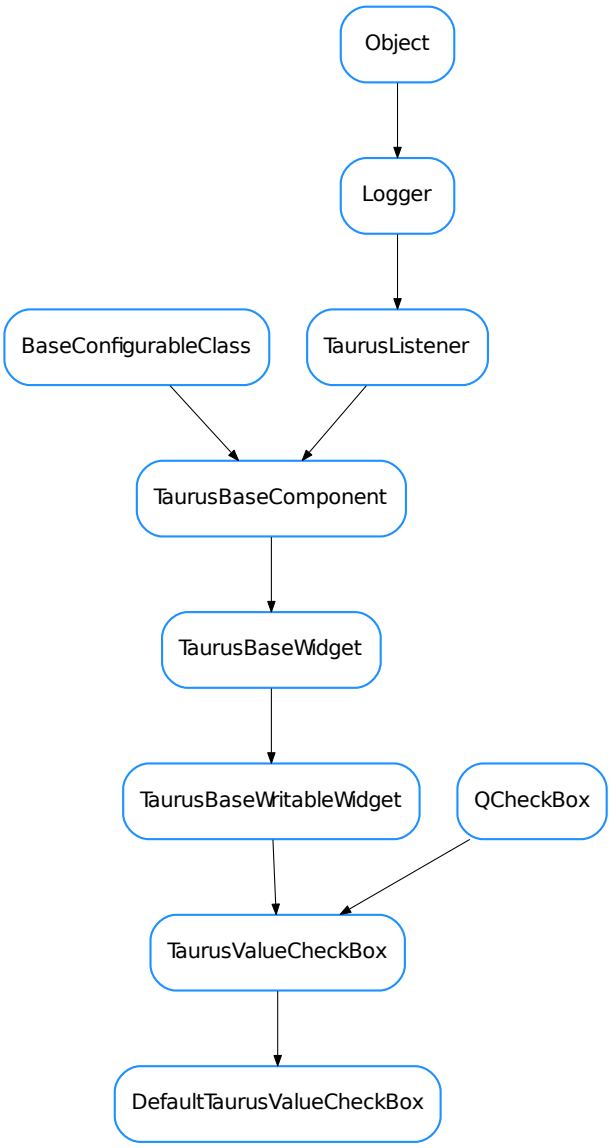
getModelMimeData()
reimplemented to use the taurusValueBuddy model instead of its own model

classmethod getQtDesignerPluginInfo()

setModel(*model*)

sizeHint()

DefaultTaurusValueCheckBox

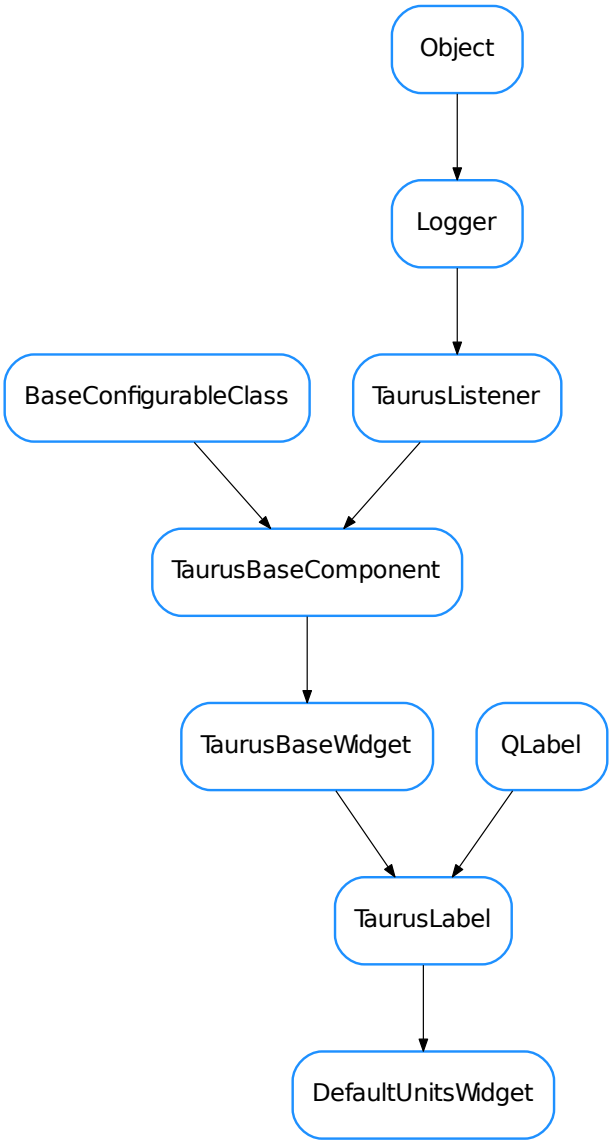


class DefaultTaurusValueCheckBox(**args*)

Bases: `taurus.qt.qtgui.input.tauruscheckbox.TaurusValueCheckBox`

classmethod getQtDesignerPluginInfo()

DefaultUnitsWidget



class **DefaultUnitsWidget**(**args*)

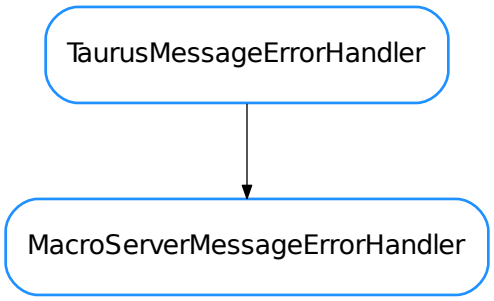
Bases: `taurus.qt.qtgui.display.tauruslabel.TaurusLabel`

classmethod **getQtDesignerPluginInfo**()

setModel(*model*)

sizeHint()

MacroServerMessageErrorHandler



class **MacroServerMessageErrorHandler**(*msgbox*)

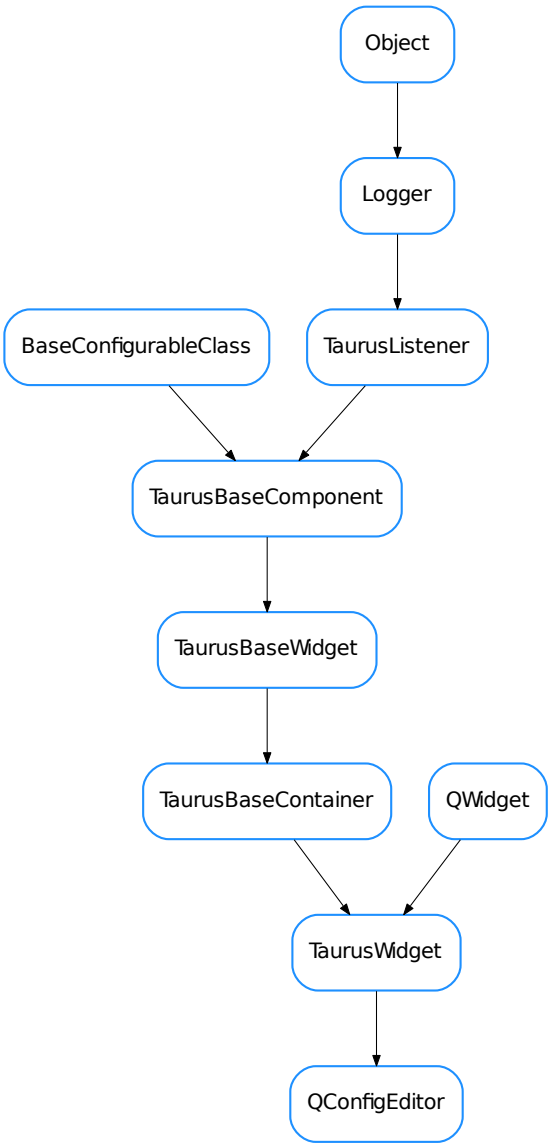
Bases: `taurus.qt.qtgui.panel.taurusmessagepanel.TaurusMessageErrorHandler`

setError(*err_type=None*, *err_value=None*, *err_traceback=None*)

Translates the given error object into an HTML string and places it in the message panel

Parameters **error** (*object*) – an error object (typically an exception object)

QConfigEditor



class `QConfigEditor`(*parent=None*, *designMode=False*)

Bases: `taurus.qt.qtgui.container.tauruswidget.TaurusWidget`

A widget that shows a tree view of the contents of Taurus configuration files saved by TaurusMainWindow and lets the user edit the values of the configuration keys

`contextMenuEvent`(*event*)

Reimplemented from `QWidget.contextMenuEvent()`

`loadFile`(*iniFileName=None*)

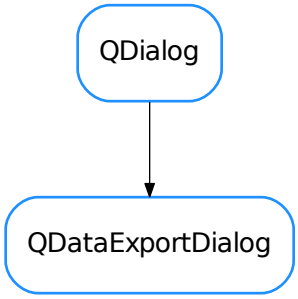
Loads a configuration stored in a file and creates the tree.

IniFileName (str) Name of the file. If None is given the user is prompted for a file.

restoreOriginal()
Replaces temporary file with the original file and builds again the configuration tree.

saveFile()
Replaces original file with temporary file (where changes were being saved).

QDataExportDialog



class QDataExportDialog(*parent=None*, *datadict=None*, *sortedNames=None*)
Bases: PyQt4.QtGui.QDialog

This creates a Qt dialog for showing and exporting x-y Ascii data from one or more curves The data sets are passed (by calling setDataSets() or at instantiation time) as a dictionary:

```
datadict={name:(x,y),...}
```

where name is the curve name and x,y are iterable containers (e.g., lists, tuple, arrays...) of data to be exported

@TODO: It would be nice if the textedit scrolled to the start ***also for the first set loaded***

allInMultipleFiles = 'All set in multiple files'

allInSingleFile = 'All sets in a single file (table like)'

exportAllData(*preffix=None*)
Exports all sets using a common preffix and appending 'XXX.dat', where XXX is a number starting at 001 if preffix is not given, the user is prompted for a directory path

exportCurrentData(*set=None*, *ofile=None*, *verbose=True*, *AllowCloseAfter=True*)
Exports data Arguments: set: the curve name. If none is passed, it uses the one selected by dataSetCB ofile: output file name or file handle. It will prompt if not provided verbose: set this to False to disable information popups AllowCloseAfter: set this to false if you want to ignore the checkbox in the dialog

exportData()

loadUi(*filename=None*, *path=None*)

onDataSetCBChange(*key*)

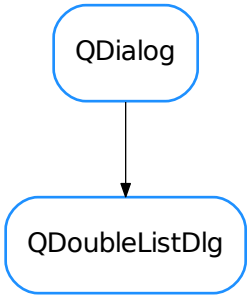
setDataSets (*datadict*, *sortedNames=None*)
Used to set the sets that are to be offered for exporting. It overwrites previous values.

setXIsTime (*xIsTime*)

updateText (*key=None*)
update the text edit that shows the preview of the data

xIsTime ()

QDoubleListDlg



class QDoubleListDlg (*parent=None*, *designMode=False*, *winTitle=''*, *mainLabel=''*, *label1=''*, *label2=''*, *list1=None*, *list2=None*)
Bases: `PyQt4.QtGui.QDialog`

Generic dialog providing two lists. Items can be moved from one to the other

getAll1 ()
returns a copy the items in the first list

Return type `list <str>`

Returns

getAll2 ()
returns a copy the items in the second list

Return type `list <str>`

Returns

loadUi (*filename=None*, *path=None*)

onTo1 (**args*)
slot to be called when the "To1" button is pressed

onTo2 (**args*)
slot to be called when the "To2" button is pressed

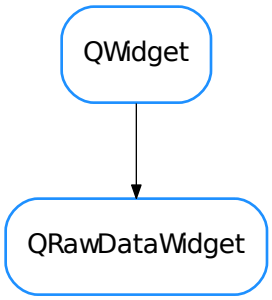
setList1 (*list1*)
sets the items to be present in the first list

Parameters **list2** (`seq <str>`) – a sequence of strings

setList2 (*list2*)
sets the items to be present in the second list

Parameters **list2** (seq <str>) – a sequence of strings

QRawDataWidget



class **QRawDataWidget** (*parent=None*)
Bases: PyQt4.QtGui.QWidget

AddCurve

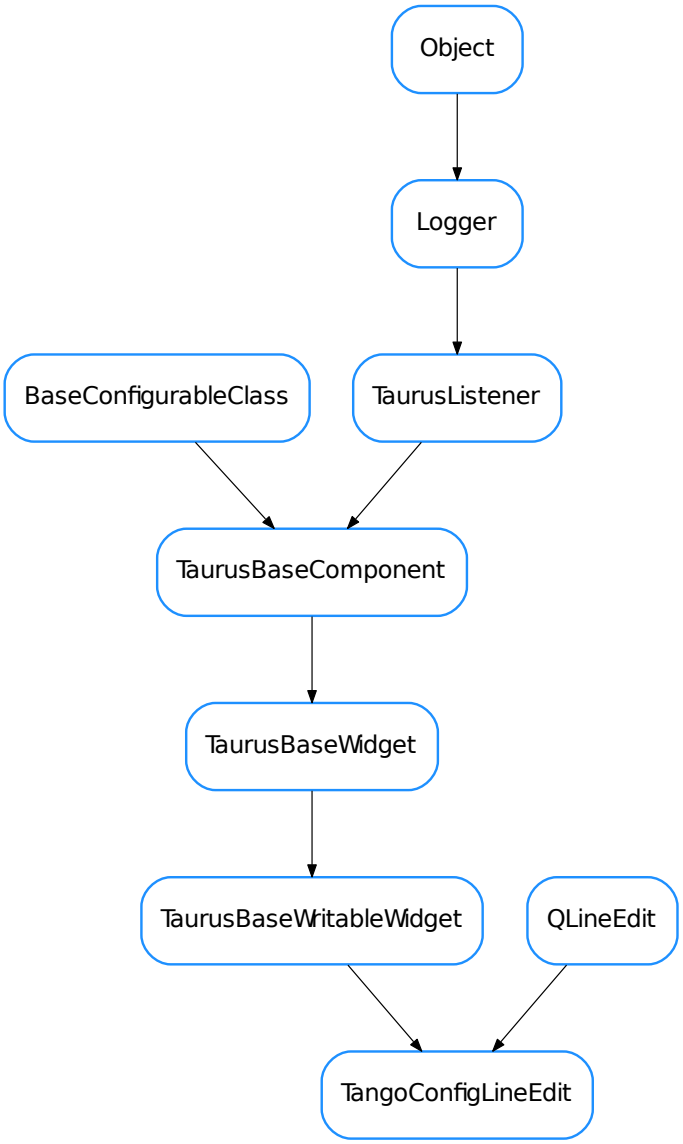
ReadFromFiles

loadUi (*filename=None*, *path=None*)

onAddCurveButtonClicked()
Emit a AddCurve signal with a rawdata dictionary as a parameter. The rawdata dictionary is prepared from the from the GUI's selection.

onOpenFilesButtonClicked()
Emit a ReadFromFiles signal with the selected xcol and skiprows as parameters

TangoConfigLineEdit



class TangoConfigLineEdit(*qt_parent=None*, *designMode=False*)

Bases: PyQt4.QtGui.QLineEdit, taurus.qt.qtgui.base.taurusbase.TaurusBaseWritableWidget

autoApply

forcedApply

getModelClass()

classmethod getQtDesignerPluginInfo()

getValue()

handleEvent(*evt_src*, *evt_type*, *evt_value*)

model

notifyValueChanged()

postAttach()

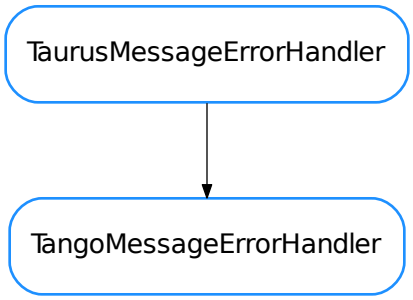
setModel(*model*)

setValue(*v*)

updatePendingOperations()

writeValue()

TangoMessageErrorHandler



class TangoMessageErrorHandler(*msgbox*)

Bases: `taurus.qt.qtgui.panel.taurusmessagepanel.TaurusMessageErrorHandler`

This class is designed to handle `PyTango.DevFailed` error into a *TaurusMessagePanel*

setError(*err_type=None*, *err_value=None*, *err_traceback=None*)

Translates the given error object into an HTML string and places it it the message panel

Parameters **error** (*object*) – an error object (typically an exception object)

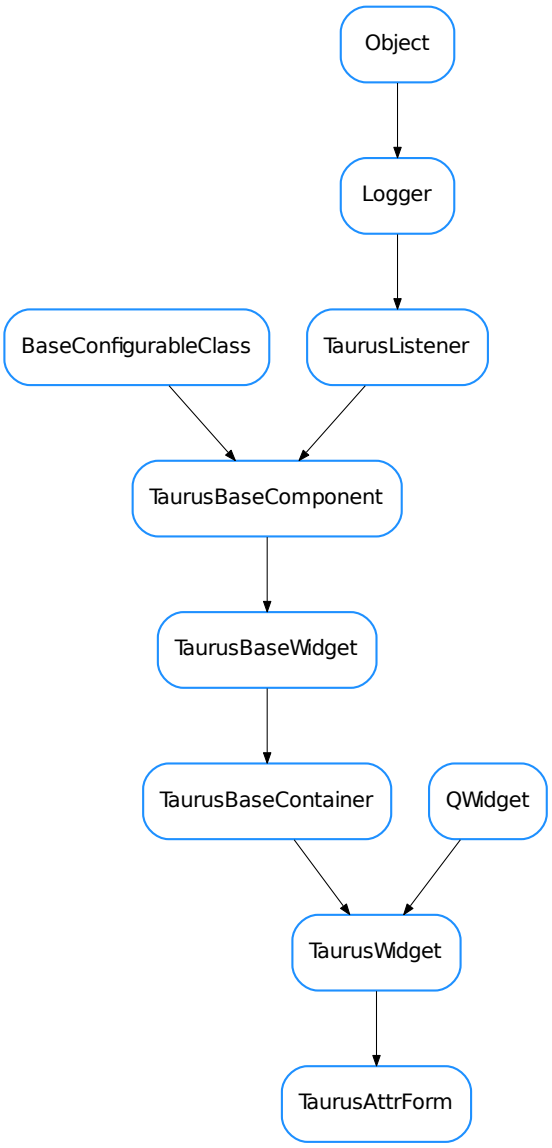
TaurusArrayEditorButton

class TaurusArrayEditorButton (*parent=None*, *designMode=False*)

Bases: taurus.qt.qtgui.panel.taurusvalue._AbstractTaurusValueButton

A button that launches a TaurusArrayEditor

TaurusAttrForm



class **TaurusAttrForm**(*parent=None*, *designMode=False*)

Bases: `taurus.qt.qtgui.container.tauruswidget.TaurusWidget`

A form that displays the attributes of a Device Server

getModelClass()

see `TaurusBaseComponent.getModelClass()`

classmethod **getQtDesignerPluginInfo**()

getViewFilters()
returns the filters used in deciding which attributes are displayed

Return type sequence <callable>

Returns a sequence of filters

model

setExpertView

setSortKey(*sortkey*)
sets the key function used to sort the attributes in the form

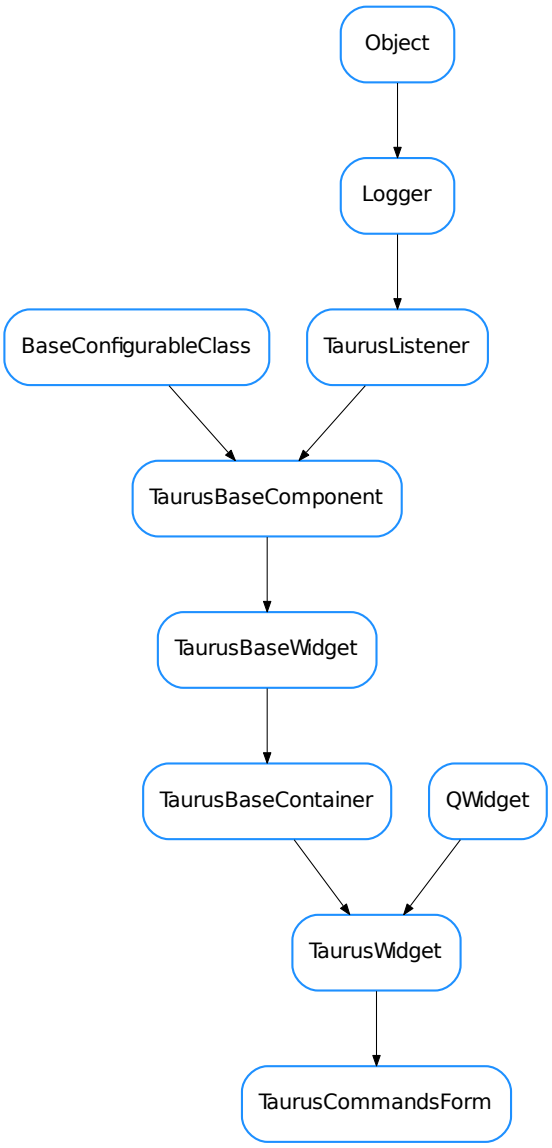
Parameters **sortkey** (callable) – a function that takes a AttributeInfo as argument and returns a key to use for sorting purposes (e.g. the default sortKey is lambda x:x.name)

setViewFilters(*filterlist*)
sets the filters to be applied when displaying the attributes

Parameters **filterlist** (sequence <callable>) – a sequence of attr filters. All filters will be applied to each attribute name to decide whether to display it or not. for an attribute to be plotted, the following condition must be true for all filters: bool(filter(attrname))==True

useParentModel

TaurusCommandsForm



class **TaurusCommandsForm** (*parent=None*, *designMode=False*)

Bases: `taurus.qt.qtgui.container.tauruswidget.TaurusWidget`

A form that shows commands available for a Device Server

applyConfig (*configdict*, ***kwargs*)

extending `TaurusBaseWidget.applyConfig()` to restore the splitter config

Parameters **configdict** (`dict`) –

See also:

`TaurusBaseWidget.applyConfig()`, *createConfig()*

createConfig(*allowUnpickable=False*)
extending `TaurusBaseWidget.createConfig()` :type alllowUnpickable: `bool` :param allowUnpickable:

Return type `dict <str, object>`

Returns configurations (which can be loaded with *applyConfig()*).

getModelClass()
see `TaurusBaseComponent.getModelClass()`

classmethod getQtDesignerPluginInfo()

getSplitter()
returns the splitter that separates the command buttons area from the output area

Return type `QSplitter`

Returns

getViewFilters()
returns the filters used in deciding which commands are displayed

Return type `sequence <callable>`

Returns a sequence of filters

setDefaultParameters(*params*)
sets the values that will appear by default in the parameters combo box, the command combo box for the command will be editable only if the first parameter is an empty string

Parameters **params** (`dict <str, list>`) – { 'cmd_name': ['parameters string 1', 'parameters string 2'] }

setExpertView

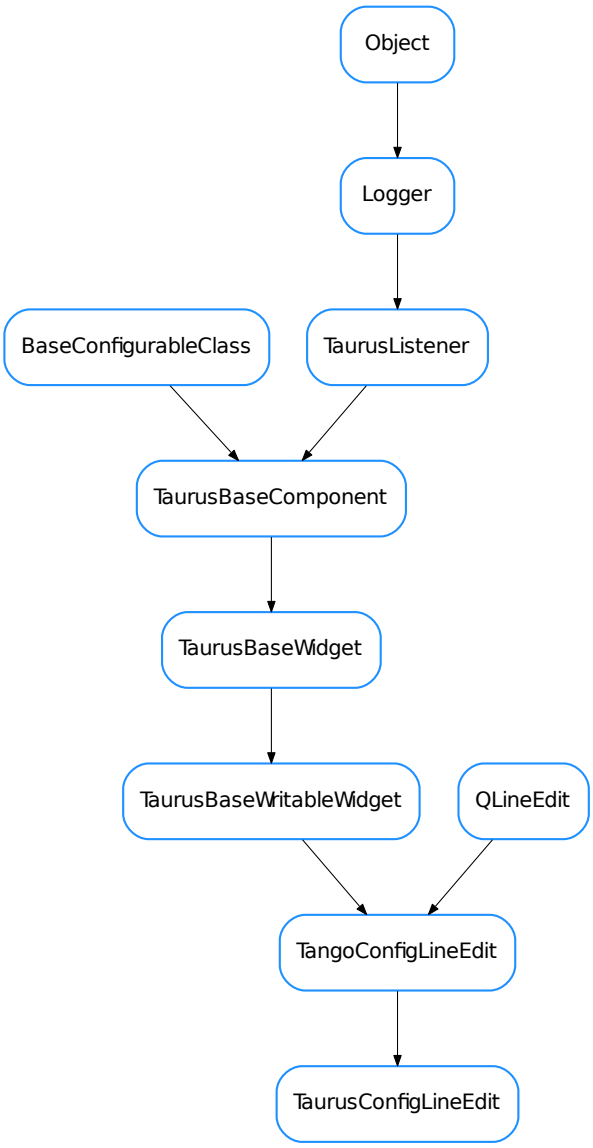
setSortKey(*sortkey*)
sets the method used to sort the commands (cmd_name by default)

Parameters **sortkey** (`callable`) – a function that takes a `CommandInfo` as argument and returns a key to use for sorting purposes (e.g. the default sortKey is `lambda x:x.cmd_name`)

setViewFilters(*filterlist*)
sets the filters to be applied when displaying the commands

Parameters **filterlist** (`sequence <callable>`) – a sequence of command filters. All filters will be applied to each command to decide whether to display it or not. for a command to be plotted, the following condition must be true for all filters: bool(filter(command))==True

TaurusConfigLineEdit

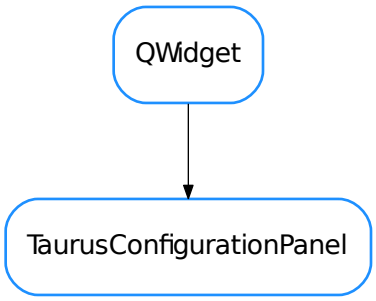


class **TaurusConfigLineEdit**(**args*, ***kwargs*)

Bases: `taurus.qt.qtgui.panel.taurusconfigurationpanel.TangoConfigLineEdit`

classmethod **getQtDesignerPluginInfo**()

TaurusConfigurationPanel



class **TaurusConfigurationPanel** (*parent=None*, *designMode=False*)

Bases: `PyQt4.QtGui.QWidget`

loadUi (*filename=None*, *path=None*)

setModel (*model*)

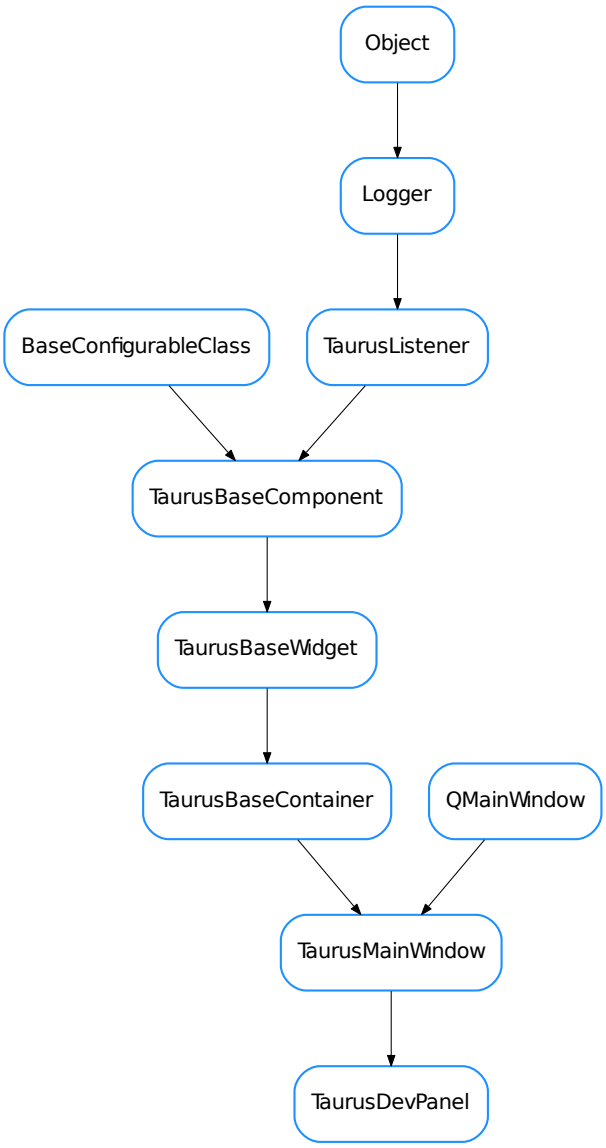
TaurusDevButton

class **TaurusDevButton** (*parent=None*, *designMode=False*)

Bases: `taurus.qt.qtgui.panel.taurusvalue._AbstractTaurusValueButton`

A button that launches a TaurusAttrForm

TaurusDevPanel



class TaurusDevPanel(*parent=None*, *designMode=False*)

Bases: taurus.qt.qtgui.container.taurusmainwindow.TaurusMainWindow

TaurusDevPanel is a Taurus Application inspired in Jive and Atk Panel.

It Provides a Device selector and several dockWidgets for interacting and displaying information from the selected device.

createActions()

create actions

classmethod **getQtDesignerPluginInfo**()

loadUi(*filename=None*, *path=None*)

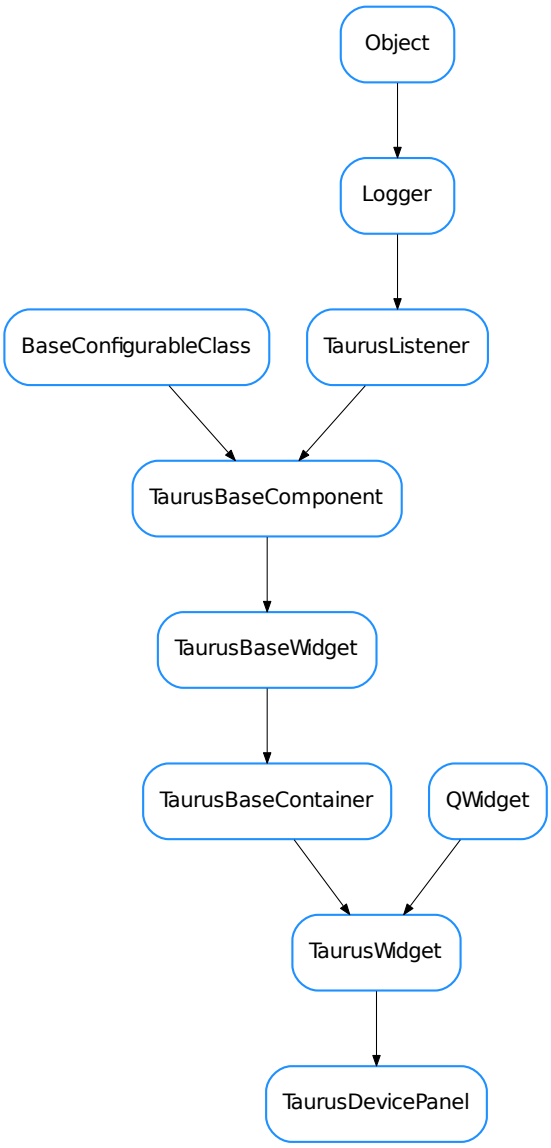
onDeviceSelected(*devinfo*)

onItemSelectionChanged(*current*, *previous*)

setDevice(*devname*)

setTangoHost(*host*)
extended from :class:setTangoHost

TaurusDevicePanel



class TaurusDevicePanel (*parent=None*, *model=None*, *palette=None*, *bound=True*)

Bases: `taurus.qt.qtgui.container.tauruswidget.TaurusWidget`

TaurusDevPanel is a Taurus Application inspired in Jive and Atk Panel.

It Provides a Device selector and a panel for displaying information from the selected device.

READ_ONLY = False

detach()

duplicate()

classmethod getAttributeFilters(*klass*)

classmethod getCommandFilters(*klass*)

classmethod getIconMap(*klass*)

get_attrs_form(*device*, *form=None*, *filters=None*, *parent=None*)

get_comms_form(*device*, *form=None*, *parent=None*)

loadConfigFile(*ifile=None*)

classmethod setAttributeFilters(*klass*, *filters*)
It will set the attribute filters filters will be like: {device_regexp:[attribute_regexps]} example: {'.*/VGCT-.*': ['ChannelState','p[0-9]']}

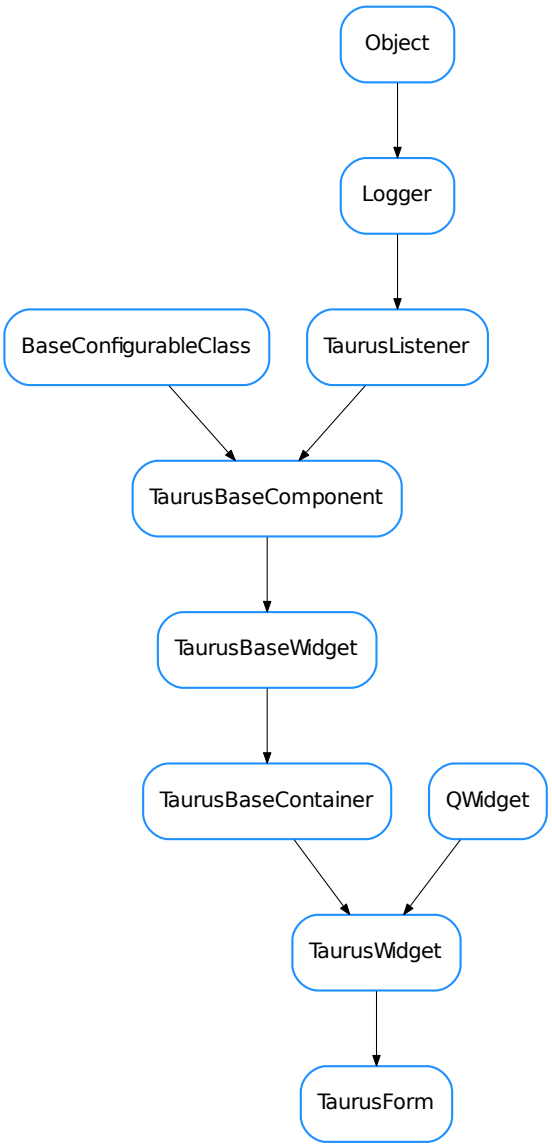
classmethod setCommandFilters(*klass*, *filters*)
It will set the command filters filters will be like: {device_regexp:[command_regexps]} example:

```
{'.*/IPCT-.*': (
                  ('setmode',('SERIAL','LOCAL','STEP','FIXED','START','PROTECT
↪')),
                  ('onhv1',()), ('offhv1',()), ('onhv2',()), ('offhv2',()),
                  ('sendcommand',())
                ),}
```

classmethod setIconMap(*klass*, *filters*)
A dictionary like {device_regexp:pixmap_url}

setModel

TaurusForm



class **TaurusForm**(*parent=None*, *formWidget=None*, *buttons=None*, *withButtons=True*, *designMode=False*)

Bases: `taurus.qt.qtgui.container.tauruswidget.TaurusWidget`

A form containing specific widgets for interacting with a given list of taurus attributes and/or devices.

Its model is a list of attribute and/or device names to be shown. Each item is represented in a row consisting of a label, a read widget, a write widget, a units widget and an "extra" widget (some of them may not be shown) which are vertically aligned with their counterparts from other items.

By default a *TaurusValue* object is used for each item, but this can be changed and specific mappings can be defined using the *setCustomWidgetMap()* method.

Item objects can be accessed by index using a list-like notation:

```
form = TaurusForm()
form.model = ['sys/tg_test/1'+a for a in ('short_image','/float_scalar','/double_
↪scalar')]
form[0].labelConfig = 'dev_alias'
form[-1].writeWidgetClass = 'TaurusWheelEdit'
print(len(form))  # --> outputs '3' (the length of the form is the number of␣
↪items)
```

By default, the form provides global Apply and Cancel buttons.

You can also see some code that exemplifies the use of TaurusForm in *Taurus coding examples*

addModels

apply

chooseAttrs()

chooseModels()
launches a model chooser dialog to modify the contents of the form

compact

destroyChildren()

dropEvent(*event*)
reimplemented to support dropping of modelnames in forms

fillWithChildren()

getCustomWidgetMap()
Returns the map used to create custom widgets.

Return type dict <str, tuple>

Returns a dictionary whose keys are device type strings (i.e. see PyTango.DeviceInfo) and whose values are tuples of classname,args,kwargs

getFormWidget(*model=None*)
Returns a tuple that can be used for creating a widget for a given model.

Parameters **model** (str) – a taurus model name for which the new item of the form will be created

Return type tuple <type, list, dict>

Returns a tuple containing a class, a list of args and a dict of keyword args. The args and the keyword args can be passed to the class constructor

getItemByIndex(*index*)
returns the child item with at the given index position.

getItemByModel(*model*, *index=0*)
returns the child item with given model. If there is more than one item with the same model, the index parameter can be used to distinguish among them Please note that his index is only relative to same-model items!

getItems()
returns a list of the objects that have been created as childs of the form

getModel()

classmethod getQtDesignerPluginInfo()

getRegExp()

isCompact()

isWithButtons()

model

modifiableByUser

onChangeLabelsAction()
changes the labelConfig of all its items

parentModelChanged

removeModels

reset

resetCompact()

resetFormWidget()

resetModel()

resetWithButtons()

setCompact(*compact*)

setCustomWidgetMap(*cwmap*)
Sets a map map for custom widgets.

Parameters **cwmap** (dict <str, tuple>) – a dictionary whose keys are device type strings (i.e. see PyTango.DeviceInfo) and whose values are tuples of classname,args,kwargs

setFormWidget(*formWidget*)

setModelCheck(*model*, *check=True*)

setModifiableByUser(*modifiable*)
sets whether the user can change the contents of the form (e.g., via Modify Contents in the context menu)
Reimplemented from TaurusWidget.setModifiableByUser()

Parameters **modifiable** (bool) –

See also:

TaurusWidget.setModifiableByUser()

setRegExp(*regExp*)

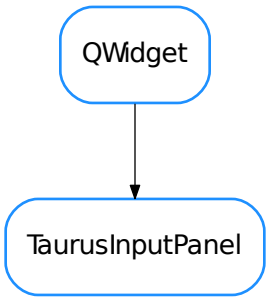
setWithButtons(*trueFalse*)

sizeHint()

useParentModel

withButtons

TaurusInputPanel



class **TaurusInputPanel**(*input_data*, *parent=None*)

Bases: PyQt4.QtGui.QWidget

A panel design to get an input from the user.

The input_data is a dictionary which contains information on how to build the input dialog. It **must** contains the following keys:

- *prompt* <str>: message to be displayed

The following are optional keys (and their corresponding default values):

- *title* <str> (doesn't have default value)
- *key* <str> (doesn't have default value): a label to be presented left to the input box represeting the label
- *unit* <str> (doesn't have default value): a label to be presented right to the input box representing the units
- *data_type* <str or sequence> ('String'): type of data to be requested. Standard accepted data types are 'String', 'Integer', 'Float', 'Boolean', 'Text'. A list of elements will be interpreted as a selection. Default TaurusInputPanel class will interpret any custom data types as 'String' and will display input widget accordingly. Custom data types can be handled differently by supplying a different input_panel_klass.
- *minimum* <int/float>: minimum value (makes sense when data_type is 'Integer' or 'Float')
- *maximum* <int/float>: maximum value (makes sense when data_type is 'Integer' or 'Float')
- *step* <int/float> (1): step size value (makes sense when data_type is 'Integer' or 'Float')
- *decimals* <int> (1): number of decimal places to show (makes sense when data_type is 'Float')
- *default_value* <obj> (doesn't have default value): default value
- *allow_multiple* <bool> (False): allow more than one value to be selected (makes sense when data_type is a sequence of possibilities)

Example:

```
app = Qt.QApplication([])

class Listener(object):
    def on_accept(self):
        print "user selected", self.panel.value()
```

```
d = dict(prompt="What's your favourite car brand?",
         data_type=["Mazda", "Skoda", "Citroen", "Mercedes", "Audi", "Ferrari"],
         default_value="Mercedes")
w = TaurusInputPanel(d)
l = Listener()
l.panel = w
w.connect(w.buttonBox(), Qt.SIGNAL("accepted()"), l.on_accept)
w.show()
app.exec_()
```

addButton(*button*, *role=3*)
Adds the given button with the given to the button box

Parameters

- **button** (*PyQt4.QtGui.QPushButton*) – the button to be added
- **role** (*PyQt4.Qt.QDialogButtonBox.ButtonRole*) – button role

buttonBox()
Returns the button box from this panel

Returns the button box from this panel

Return type PyQt4.Qt.QDialogButtonBox

create_boolean_panel(*input_data*)

create_custom_panel(*input_data*)

create_float_panel(*input_data*)

create_integer_panel(*input_data*)

create_selection_panel(*input_data*)

create_single_input_panel(*input_data*)

create_string_panel(*input_data*)

create_text_panel(*input_data*)

fill_main_panel(*panel*, *input_data*)

getText()
Returns the current text of this panel

Returns the text for this panel

Return type str

inputPanel()

loadUi(*filename=None*, *path=None*)

setIconPixmap(*pixmap*)
Sets the icon to the dialog

Parameters **pixmap** (*PyQt4.Qt.QPixmap*) – the icon pixmap

setInputFocus()

setText(*text*)
Sets the text of this panel

Parameters **text** (*str*) – the new text

TaurusMessageErrorHandler

class TaurusMessageErrorHandler(*msgbox*)

Bases: object

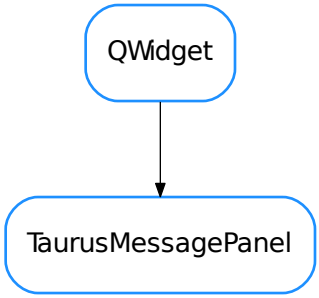
This class is designed to handle a generic error into a *TaurusMessagePanel*

setError(*err_type=None*, *err_value=None*, *err_traceback=None*)

Translates the given error object into an HTML string and places it in the message panel

Parameters **error** (*object*) – an error object (typically an exception object)

TaurusMessagePanel



class TaurusMessagePanel(*err_type=None*, *err_value=None*, *err_traceback=None*, *parent=None*, *designMode=False*)

Bases: PyQt4.QtGui.QWidget

A panel intended to display a taurus error. Example:

```
dev = taurus.Device("sys/tg_test/1")
try:
    print dev.read_attribute("throw_exception")
except PyTango.DevFailed, df:
    msgbox = TaurusMessagePanel()
    msgbox.show()
```

You can show the error outside the exception handling code. If you do this, you should keep a record of the exception information as given by `sys.exc_info()`:

```
dev = taurus.Device("sys/tg_test/1")
exc_info = None
try:
    print dev.read_attribute("throw_exception")
except PyTango.DevFailed, df:
    exc_info = sys.exc_info()

if exc_info:
    msgbox = TaurusMessagePanel(*exc_info)
    msgbox.show()
```

ErrorHandlers = {<class 'PyTango.DevFailed'>: <class 'taurus.qt.qtgui.panel.taurusmessagepanel.TangoMessageErro

addButton(*button*, *role=3*)
Adds the given button with the given to the button box

Parameters

- **button** (*PyQt4.QtGui.QPushButton*) – the button to be added
- **role** (*PyQt4.Qt.QDialogButtonBox.ButtonRole*) – button role

buttonBox()
Returns the button box from this panel

Returns the button box from this panel

Return type PyQt4.Qt.QDialogButtonBox

checkBox()
Returns the check box from this panel

Returns the check box from this panel

Return type PyQt4.Qt.QCheckBox

checkBoxState()
Returns the check box state

Returns the check box state

Return type PyQt4.Qt.CheckState

checkBoxText()
Returns the check box text

Returns the check box text

Return type str

classmethod findErrorHandler(*klass*, *err_type*)
Finds the proper error handler class for the given error

Parameters **err_type** (*class object*) – error class

Returns a message box error handler

Return type TaurusMessageBoxErrorHandler class object

getDetailedHtml()
Returns the current detailed HTML of this panel

Returns the detailed HTML for this panel

Return type str

getDetailedText()
Returns the current detailed text of this panel

Returns the detailed text for this panel

Return type str

getError()
Returns the current exception information of this panel

Returns the current exception information (same as type as returned by `sys.exc_info()`)

Return type tuple<type, *value*, traceback>

getOriginHtml()
Returns the current origin HTML of this panel

Returns the origin HTML for this panel

Return type str

getOriginText()
Returns the current origin text of this panel

Returns the origin text for this panel

Return type str

getText()
Returns the current text of this panel

Returns the text for this panel

Return type str

loadUi(*filename=None*, *path=None*)

classmethod registerErrorHandler(*klass*, *err_type*, *err_handler*)

reportComboBox()

setCheckBoxState(*state*)
Sets the checkbox state.

Parameters **text** (*PyQt4.Qt.CheckState*) – new checkbox state

setCheckBoxText(*text*)
Sets the checkbox text.

Parameters **text** (*str*) – new checkbox text

setCheckBoxVisible(*visible*)
Sets the checkbox visibility.

Parameters **visible** (*bool*) – True makes checkbox visible, False hides it

setDetailedHtml(*html*)
Sets the detailed HTML of the dialog

Parameters **html** (*str*) – the new HTML text

setDetailedText(*text*)
Sets the detailed text of the dialog

Parameters **text** (*str*) – the new text

setError(*err_type=None*, *err_value=None*, *err_traceback=None*)
Sets the exception object. Example usage:

```
dev = taurus.Device("sys/tg_test/1")
exc_info = None
msgbox = TaurusMessagePanel()
try:
    print dev.read_attribute("throw_exception")
except PyTango.DevFailed, df:
    exc_info = sys.exc_info()

if exc_info:
    msgbox.setError(*exc_info)
    msgbox.show()
```

Parameters

- **err_type** – the exception type of the exception being handled (a class object)
- **err_value** (*object*) – exception object
- **err_traceback** (*TracebackType*) – a traceback object which encapsulates the call stack at the point where the exception originally occurred

setIconPixmap(*pixmap*)
Sets the icon to the dialog

Parameters **pixmap** (*PyQt4.Qt.QPixmap*) – the icon pixmap

setOriginHtml(*html*)
Sets the origin HTML of the dialog

Parameters **html** (*str*) – the new HTML text

setOriginText(*text*)
Sets the origin text of the dialog

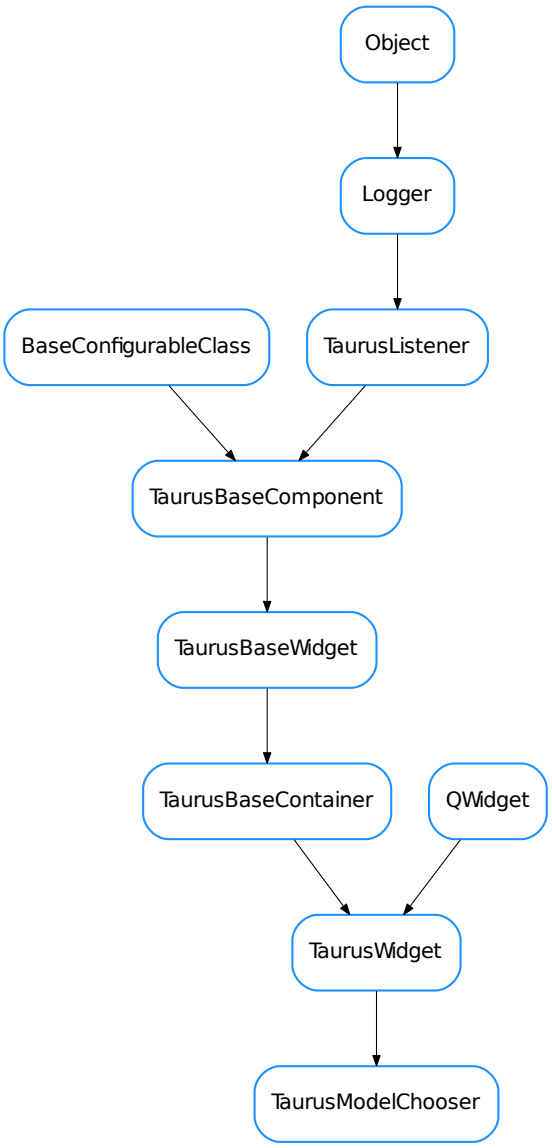
Parameters **text** (*str*) – the new text

setText(*text*)
Sets the text of this panel

Parameters **text** (*str*) – the new text

toggledDetails

TaurusModelChooser



class **TaurusModelChooser**(*parent=None*, *selectables=None*, *host=None*, *designMode=None*, *singleModel=False*)

Bases: `taurus.qt.qtgui.container.tauruswidget.TaurusWidget`

A widget that allows the user to select a list of models from a tree representing devices and attributes from a Tango server.

The user selects models and adds them to a list. Then the user should click on the update button to notify that the selection is ready.

signals::

- "updateModels" emitted when the user clicks on the update button. It passes a list<str> of models that have been selected.

UpdateAttrs

addModels(*models*)
Add given models to the selected models list

getListedModels(*asMimeData=False*)
returns the list of models that have been added

Parameters **asMimeData** (bool) – If False (default), the return value will be a list of models. If True, the return value is a *QMimeData* containing at least *TAURUS_MODEL_LIST_MIME_TYPE* and *text/plain* MIME types. If only one model was selected, the mime data also contains a TAURUS_MODEL_MIME_TYPE.

Return type list <str> or QMimeData

Returns the type of return depends on the value of *asMimeData*

classmethod getQtDesignerPluginInfo()

isSingleModelMode()
returns True if the selection is limited to just one model. Returns False otherwise.

Return type bool

Returns

static modelChooserDlg(*parent=None*, *selectables=None*, *host=None*, *asMimeData=False*, *singleModel=False*, *windowTitle='Model Chooser'*)
Static method that launches a modal dialog containing a TaurusModelChooser

Parameters

- **parent** (QObject) – parent for the dialog
- **selectables** (list <TaurusElementType>) – if passed, only elements of the tree whose type is in the list will be selectable.
- **host** (QObject) – Tango host to be explored by the chooser
- **asMimeData** (bool) – If False (default), a list of models will be. returned. If True, a *QMimeData* object will be returned instead. See *getListedModels()* for a detailed description of this QMimeData object.
- **singleModel** (bool) – If True, the selection will be of just one model. Otherwise (default) a list of models can be selected
- **windowTitle** (str) – Title of the dialog (default="Model Chooser")

Return type list,bool or QMimeData,bool

Returns Returns a models,ok tuple. models can be either a list of models or a QMimeData object, depending on *asMimeData*. ok is True if the dialog was accepted (by clicking on the "update" button) and False otherwise

onRemoveSelected()
Remove the list-selected models from the list

resetListedModels()
equivalent to setListedModels([])

resetSingleModelMode()
equivalent to setSingleModelMode(False)

setListedModels(*models*)
adds the given list of models to the widget list

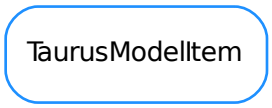
setSingleModelMode(*single*)
sets whether the selection should be limited to just one model (single=True) or not (single=False)

singleModelMode

updateList(*attrList*)
for backwards compatibility with AttributeChooser only. Use *setListedModels()* instead

updateModels

TaurusModelItem



class TaurusModelItem(*src=None*, *display=None*)
Bases: object

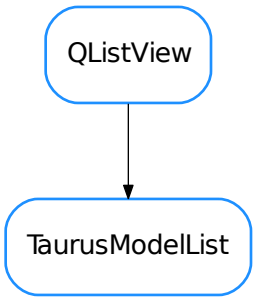
An item object for *TaurusModelModel*. Exposes *display icon* and *ok* attributes which are calculated and kept in synch with the property *src*

getSrc()
getter for src.

setSrc(*src*)
processes the src and sets the values of _src, display, icon and ok attributes

src
getter for src.

TaurusModelList



class TaurusModelList(*parent=None*, *items=None*, *designMode=False*)

Bases: PyQt4.QtGui.QListView

A list view widget to display and manage a list of models

Tries to identify the type of model and show the state of the device/attr associated with it. It also allows drag and drop of models and sorting.

addModels(*models*)

adds models to the list

Parameters **models** (list <str>) – sequence of model names to be added

clear()

removes all items from the list

contextMenuEvent(*event*)

see QWidget.contextMenuEvent()

dataChangedSignal

getModelItems()

returns the model item objects

Return type list <*TaurusModelItem*>

Returns

See also:

getModelList()

getModelList()

returns a the model names corresponding to the items in the list

Return type list <str>

Returns

See also:

getModelItems()

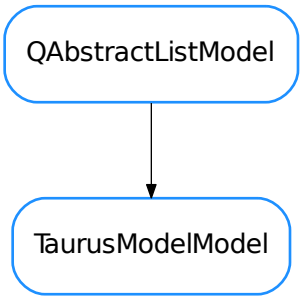
classmethod **getQtDesignerPluginInfo**()

newRow(*position=None*)
adds an empty row *before* the given position

Parameters **position** (int or None) – position at which the new row will be added. If None passed, it will be added at the end.

removeSelected()
removes selected items from the list

TaurusModelModel



class TaurusModelModel(*items=None*)
Bases: PyQt4.QtCore.QAbstractListModel

A Qt data model for describing taurus models

addItem(*item*)
appends an item to the internal list

Parameters **item** (*TaurusModelItem*) –

clearAll()
clears all rows

data(*index*, *role=0*)
reimplemented from Qt.QAbstractListModel

dropMimeData(*data*, *action*, *row*, *column*, *parent*)
reimplemented from Qt.QAbstractListModel

dumpData()
returns a deep copy of the internal item list representation

flags(*index*)
reimplemented from Qt.QAbstractListModel

insertItems(*row*, *items*)
convenience method to add new rows by passing a list of strings ()

Parameters

- **row** (int) – the row of the list at which the item insertion starts, if row==-1, items will be appended to the list

- **items** (seq) – a sequence items to add to the list. The objects in the sequence can be either strings, *TaurusModelItem* objects or tuples of valid arguments for initializing *TaurusModelItem* objects

insertRows(*position=None*, *rows=1*, *parentindex=None*, *items=None*)
reimplemented from Qt.QAbstractListModel

mimeData(*indexes*)
reimplemented from Qt.QAbstractListModel

mimeTypes()
reimplemented from Qt.QAbstractListModel

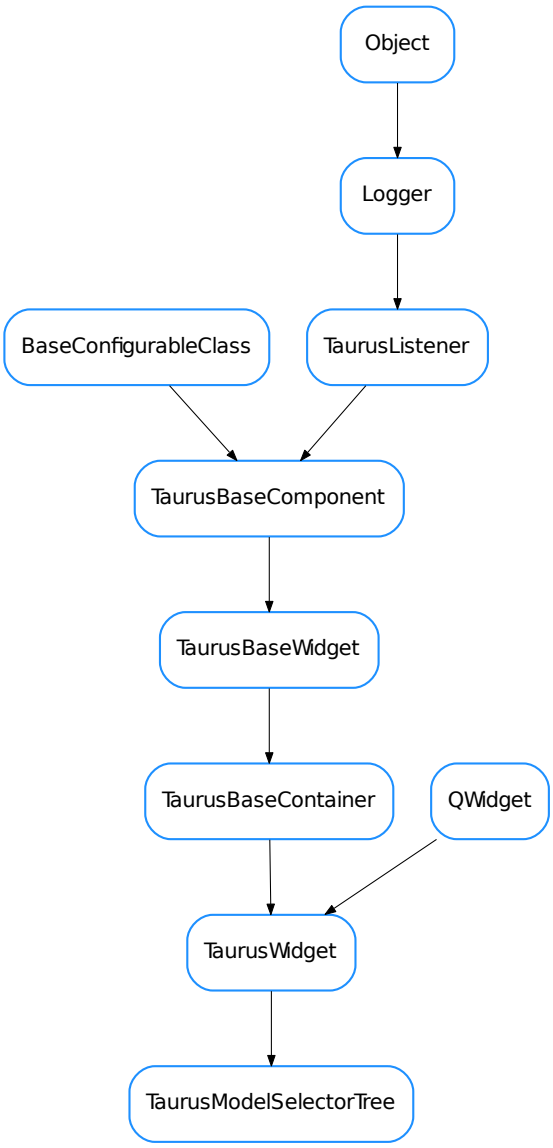
removeRows(*position*, *rows=1*, *parentindex=None*)
reimplemented from Qt.QAbstractListModel

rowCount(*index=<PyQt4.QtCore.QModelIndex object>*)
reimplemented from Qt.QAbstractListModel

setData(*index*, *value=None*, *role=2*)
reimplemented from Qt.QAbstractListModel

swapItems(*index1*, *index2*)
swap the items described by index1 and index2 in the list

TaurusModelSelectorTree



class **TaurusModelSelectorTree**(*parent=None*, *selectables=None*, *buttonsPos=None*, *designMode=None*)

Bases: `taurus.qt.qtgui.container.tauruswidget.TaurusWidget`

addModels

classmethod **getQtDesignerPluginInfo**()

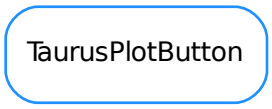
getSelectedModels()

onAddSelected()

setButtonsPos(*buttonsPos*)

treeView()

TaurusPlotButton

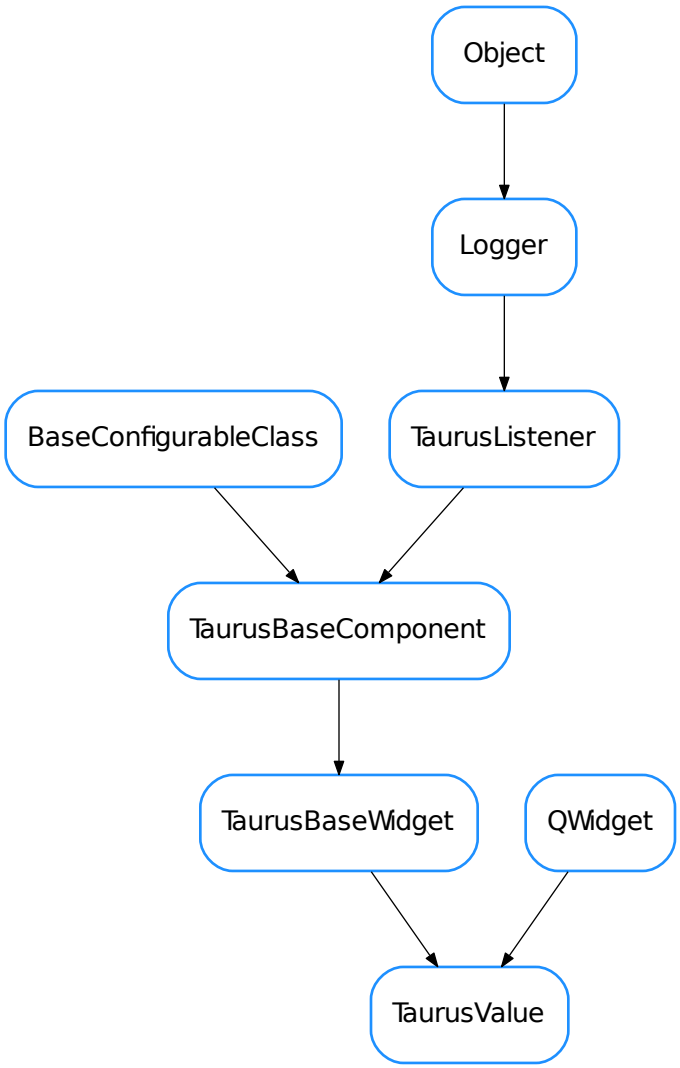


class TaurusPlotButton(*parent=None*, *designMode=False*)

Bases: `taurus.qt.qtgui.panel.taurusvalue._AbstractTaurusValueButton`

A button that launches a TaurusPlot

TaurusValue



class TaurusValue (*parent=None*, *designMode=False*, *customWidgetMap=None*)

Bases: `PyQt4.QtGui.QWidget`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget`

Internal TaurusValue class

> **Warning:** *TaurusValue* (and any derived class from it) should never be instantiated directly. It is designed to be instantiated by a *TaurusForm* class, since it breaks some conventions on the way it manages layouts of its parent model.

addCustomWidgetToLayout()

addExtraWidgetToLayout()

addLabelWidgetToLayout()

addReadWidgetToLayout()

addUnitsWidgetToLayout()

addWriteWidgetToLayout()

allowWrite

applyConfig(*configdict*, ***kwargs*)
extending `TaurusBaseWidget.applyConfig()` to restore the subwidget's classes

Parameters **configdict** (`dict`) –

See also:

`TaurusBaseWidget.applyConfig()`, *createConfig()*

createConfig(*allowUnpickable=False*)
extending `TaurusBaseWidget.createConfig()` to store also the class names for subwidgets

Parameters **alllowUnpickable** (`bool`) –

Return type `dict <str, object>`

Returns configurations (which can be loaded with *applyConfig()*).

customWidget()
Returns the custom widget

customWidgetClassFactory(*classID*)

extraWidget()
Returns the extra widget

extraWidgetClass

extraWidgetClassFactory(*classID*)

getAllowWrite()

getCustomWidgetClass()

getCustomWidgetMap()
Returns the map used to create custom widgets.

Return type `dict <str, QWidget>`

Returns a dictionary whose keys are device type strings (i.e. see `PyTango.DeviceInfo`) and whose values are widgets to be used

getDefaultCustomWidgetClass()

getDefaultExtraWidgetClass()

getDefaultLabelWidgetClass()

getDefaultReadWidgetClass(*returnAll=False*)
Returns the default class (or classes) to use as read widget for the current model.

Parameters **returnAll** (`bool`) – if True, the return value is a list of valid classes instead of just one class

Return type `class` or `list <class>`

Returns the default class to use for the read widget (or, if returnAll==True, a list of classes that can show the attribute). If a list is returned, it will be loosely ordered by preference, being the first element always the default one.

getDefaultUnitsWidgetClass()

getDefaultWriteWidgetClass(*returnAll=False*)

Returns the default class (or classes) to use as write widget for the current model.

Parameters **returnAll** (bool) – if True, the return value is a list of valid classes instead of just one class

Return type class or list <class>

Returns the default class to use for the write widget (or, if returnAll==True, a list of classes that can show the attribute). If a list is returned, it will be loosely ordered by preference, being the first element always the default one.

getExtraWidgetClass()

getLabelConfig()

getLabelWidgetClass()

getModelClass()

getPreferredRow()

classmethod getQtDesignerPluginInfo()

getReadWidgetClass()

getRow()

getSwitcherClass()

Returns the TaurusValue switcher class (used in compact mode). Override this method if you want to use a custom switcher in TaurusValue subclasses.

getUnitsWidgetClass()

getWriteWidgetClass()

handleEvent(*evt_src*, *evt_type*, *evt_value*)

Reimplemented from TaurusBaseWidget.handleEvent() to update subwidgets on config events

hasPendingOperations()

self.getPendingOperations will always return an empty list, but still self.hasPendingOperations will look at the writeWidget's operations. If you want to ask the TaurusValue for its pending operations, call self.writeWidget().getPendingOperations()

isCompact()

isReadOnly()

isValueChangedByUser()

labelConfig

labelWidget()

Returns the label widget

labelWidgetClass

labelWidgetClassFactory(*classID*)

minimumHeight()

model

modifiableByUser

onChangeLabelConfig()

onChangeReadWidget()

onChangeWriteWidget()

parentModelChanged

preferredRow

readWidget(*followCompact=False*)
Returns the read widget. If followCompact=True, and compact mode is used, it returns the switcher's readWidget instead of the switcher itself.

readWidgetClass

readWidgetClassFactory(*classID*)

resetAllowWrite()

resetCustomWidgetClass()

resetExtraWidgetClass()

resetLabelConfig()

resetLabelWidgetClass()

resetPreferredRow()

resetReadWidgetClass()

resetUnitsWidgetClass()

resetWriteWidgetClass()

setAllowWrite

setCompact(*compact*)

setCustomWidgetClass

setCustomWidgetMap(*cwmap*)
Sets a map map for custom widgets.

Parameters **cwmap** (dict <str, QWidget>) – a dictionary whose keys are device class strings (see PyTango.DeviceInfo) and whose values are widget classes to be used

setDangerMessage(*dangerMessage=None*)

setExtraWidgetClass

setForceDangerousOperations(*yesno*)

setLabelConfig

setLabelWidgetClass

setMinimumHeight(*minimumHeight*)

setModel

setParent(*parent*)

setPreferredRow

setReadWidgetClass

setUnitsWidgetClass

setVisible(*visible*)

setWriteWidgetClass

unitsWidget()
Returns the units widget

unitsWidgetClass

unitsWidgetClassFactory(*classID*)

updateCustomWidget()

updateExtraWidget()

updateLabelWidget()

updatePendingOpsStyle()

updateReadWidget()

updateUnitsWidget()

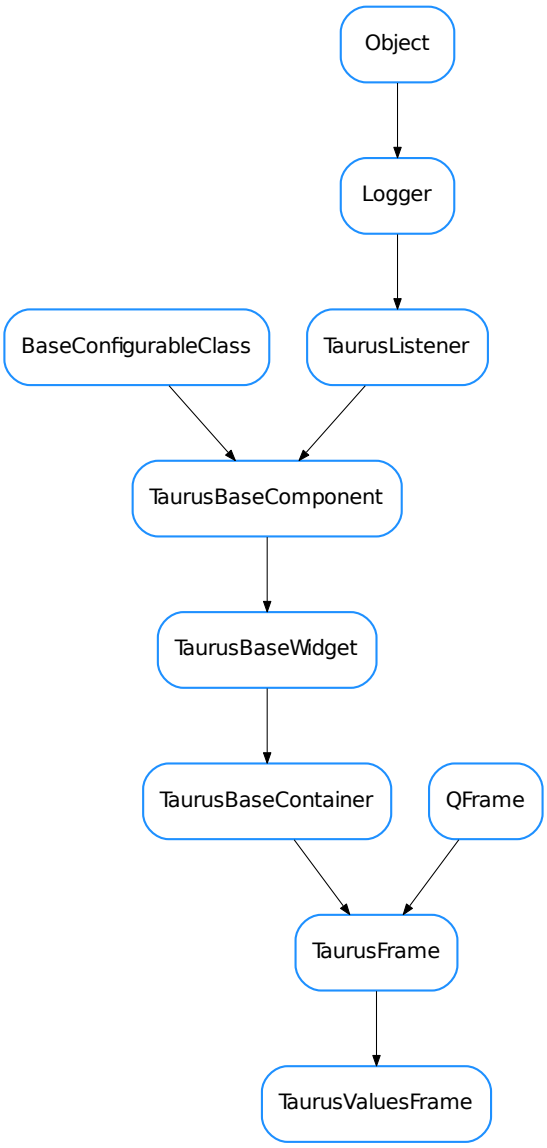
updateWriteWidget()

writeWidget(*followCompact=False*)
Returns the write widget. If followCompact=True, and compact mode is used, it returns the switcher's writeWidget instead of None.

writeWidgetClass

writeWidgetClassFactory(*classID*, *ignoreCompact=False*)

TaurusValuesFrame



class **TaurusValuesFrame** (*parent=None*, *designMode=False*)

Bases: `taurus.qt.qtgui.container.taurusframe.TaurusFrame`

This is a container specialized into containing TaurusValue widgets. It should be used Only for TaurusValues

getModel ()

classmethod **getQtDesignerPluginInfo** ()

we don't want this widget in designer

getTaurusValueByIndex(*index*)
returns the TaurusValue item at the given index position

getTaurusValues()
returns the list of TaurusValue Objects contained by this frame

model

resetModel()

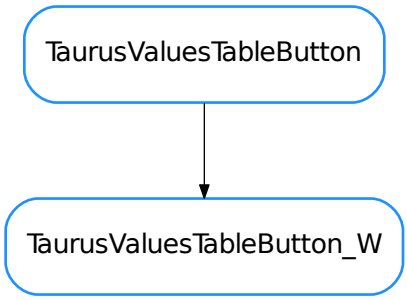
setModel

TaurusValuesTableButton

class TaurusValuesTableButton(*parent=None*, *designMode=False*)
Bases: `taurus.qt.qtgui.panel.taurusvalue._AbstractTaurusValueButton`

A button that launches a TaurusValuesTable

TaurusValuesTableButton_W



class TaurusValuesTableButton_W(*parent=None*, *designMode=False*)
Bases: `taurus.qt.qtgui.panel.taurusvalue.TaurusValuesTableButton`

A button that launches a TaurusValuesTable

- *DefaultLabelWidget*
- *DefaultTaurusValueCheckBox*

- *DefaultUnitsWidget*
- *MacroServerMessageErrorHandler*
- *QConfigEditor*
- *QDataExportDialog*
- *QDoubleListDlg*
- *QRawDataWidget*
- *TangoConfigLineEdit*
- *TangoMessageErrorHandler*
- *TaurusArrayEditorButton*
- *TaurusAttrForm*
- *TaurusCommandsForm*
- *TaurusConfigLineEdit*
- *TaurusConfigurationPanel*
- *TaurusDevButton*
- *TaurusDevPanel*
- *TaurusDevicePanel*
- *TaurusForm*
- *TaurusInputPanel*
- *TaurusMessageErrorHandler*
- *TaurusMessagePanel*
- *TaurusModelChooser*
- *TaurusModelItem*
- *TaurusModelList*
- *TaurusModelModel*
- *TaurusModelSelectorTree*
- *TaurusPlotButton*
- *TaurusValue*
- *TaurusValuesFrame*
- *TaurusValuesTableButton*
- *TaurusValuesTableButton_W*

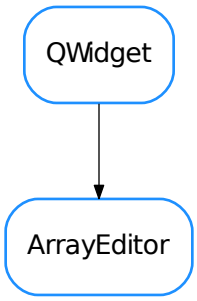
taurus.qt.qtgui.plot

Taurus Widget Plot module

This module is part of Taurus Widgets. It contains specialized widgets for 2D plotting in Taurus. It depends on the PyQwt module

Classes

ArrayEditor



class ArrayEditor(*parent=None*)

Bases: `PyQt4.QtGui.QWidget`

changeCPointSelection(*newpos*)

connectToController(*ctrl*)

delController(*index*)

getCorrected()

returns x,c where x and c are numpy arrays representing the abcissas and ordinates for the corrected curve, respectively

getCorrection()

returns xp,cp where xp and cp are numpy arrays representing the abcissas and ordinates for the correction points, respectively

getMaster()

returns x,m where x and m are numpy arrays representing the abcissas and ordinates for the master, respectively

insertController(*xp*, *index=None*)

insertControllers(*xplist*)

loadUi(*filename=None*, *path=None*)

makeControllerVisible(*ctrl=None*)

onAddRegEspCPointsBT()

onAddSingleCPointBT()

onCorrSBChanged(*value=None*)

recalculates the position and value of the control points (self.xp and self.corrp) as well as the correction curve (self.corr)

onLCopy(*checked*)

onLScale(*checked*)

onRCopy(*checked*)

onRScale(*checked*)

plot1MouseDoubleClickEvent(*event*)

plot1MousePressEvent(*event*)

plot1MouseReleaseEvent(*event*)

plot2MouseDoubleClickEvent(*event*)

plot2MousePressEvent(*event*)

plot2MouseReleaseEvent(*event*)

plotMouseDoubleClickEvent(*event*, *taurusplot*)

plotMousePressEvent(*event*, *taurusplot*)

plotMouseReleaseEvent(*event*, *taurusplot*)

resetCorrection()

resetMaster()

setCorrection(*xp=None*, *corrp=None*)
sets control points at the points specified by xp and with the values specified by corrp. Example:

```
setCorrection([1,2,8,9], [0,0,0,0])
```

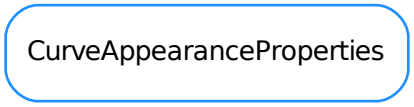
would set 4 control points with initial value 0 at x=1, 2, 8 and 9s

setMaster(*x*, *y*, *keepCP=False*, *keepCorr=False*)

showEditCPointsDialog()

updatePlots()

CurveAppearanceProperties



class CurveAppearanceProperties(*sStyle=None*, *sSize=None*, *sColor=None*, *sFill=None*, *lStyle=None*, *lWidth=None*, *lColor=None*, *cStyle=None*, *yAxis=None*, *cFill=None*, *title=None*, *visible=None*)

Bases: object

An object describing the appearance of a TaurusCurve

applyToCurve(*curve*)
applies the current properties to a given curve If a property is set to None, it is not applied to the curve

conflictsWith(*other*, *strict=True*)
returns a list of attribute names that are in conflict between this self and other

static inConflict_none(*a*, *b*)
In case of conflict, returns None

static inConflict_update_a(*a*, *b*)
This function can be passed to CurvesAppearance.merge() if one wants to update prop1 with prop2 except for those attributes of prop2 that are set to None

classmethod merge(*plist*, *attributes=None*, *conflict=None*)
returns a CurveAppearanceProperties object formed by merging a list of other CurveAppearanceProperties objects

Note: This is a class method, so it can be called without previously instantiating an object

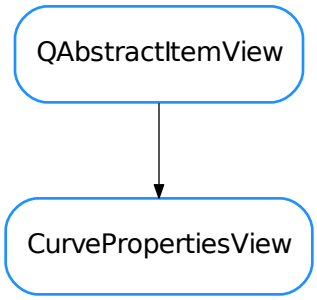
Parameters

- **plist** (sequence <*CurveAppearanceProperties*>) – objects to be merged
- **attributes** (sequence <str>) – the name of the attributes to consider for the merge. If None, all the attributes will be merged
- **conflict** (callable) – a function that takes 2 objects (having a different attribute)and returns a value that solves the conflict. If None is given, any conflicting attribute will be set to None.

Return type *CurveAppearanceProperties*

Returns merged properties

CurvePropertiesView



class CurvePropertiesView(*parent=None*, *designMode=False*)
Bases: PyQt4.QtGui.QAbstractItemView

This widget is a view on a CurvesTableModel. It displays and allows to change the properties of selected curve(s). Note that this widget does not allow to change selection by itself, but rather relies on some other view on the same model (like a QTableView) to alter the selection.

blockControlsSignals(*block*)
blocks/unblocks the signals from all the properties controls

Parameters **block** (bool) – If True, signals are blocked. If False they are unblocked

dataChanged(*topleft*, *bottomright*)
Reimplemented. See `Qt.QAbstractItemView.dataChanged()`

getShownProperties()
Returns a copy of the currently shown properties

Return type *CurveAppearanceProperties*

Returns

horizontalOffset(**args*, ***kwargs*)
dummy reimplementation

indexAt(**args*, ***kwargs*)
dummy reimplementation

loadUi(*filename=None*, *path=None*)

onPropertyControlChanged(**args*)
slot called whenever one of the controls is changed

scrollTo(**args*, ***kwargs*)
dummy reimplementation

selectionChanged(*selected*, *deselected*)
Reimplemented. See `Qt.QAbstractItemView.selectionChanged()`

showProperties(*prop*, *blockSignals=True*)
Updates the control widgets to show the given properties.

..note:: that the signals of the controls may be temporally blocked to prevent loops. See the *blockSignals* parameter.

Parameters

- **prop** (*CurveAppearanceProperties*) – the properties object containing what should be shown. If a given property is set to None, the corresponding widget will show a "neutral" display
- **blockSignals** (`bool`) – If True (default) the signals of the control widgets are blocked while updating them to avoid loops.

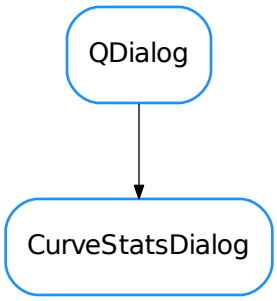
updateControls()
Updates the state of the properties controls to reflect the selection

verticalOffset(**args*, ***kwargs*)
dummy reimplementation

visualRect(**args*, ***kwargs*)
dummy reimplementation

visualRegionForSelection(**args*, ***kwargs*)
dummy reimplementation

CurveStatsDialog



class **CurveStatsDialog**(*parent=None*)

Bases: PyQt4.QtGui.QDialog

A dialog for configuring and displaying statistics from curves of a plot

closeEvent(*event*)

See Qwidget.closeEvent()

closed

getSelectedRows()

returns a list of row numbers corresponding to the selected rows of the table

loadUi(*filename=None*, *path=None*)

maxSelected(*pos*)

slot called when the user has selected a min value from the plot

minSelected(*pos*)

slot called when the user has selected a min value from the plot

onCalculate()

slot called when the calculate button is pressed. Performs the calculation of stats in the current limits for the currently selected curves (or for all if none selected) and fills the table.

onMaxChanged(*x*)

slot called when the max value is changed

onMinChanged(*x*)

slot called when the min value is changed

onSelectMax()

slot called when the user clicks on the selectMax button

onSelectMin()

slot called when the user clicks on the selectMin button

onStatToggled(*checked*)

slot called when any of the stat checkboxes is toggled

refreshCurves()

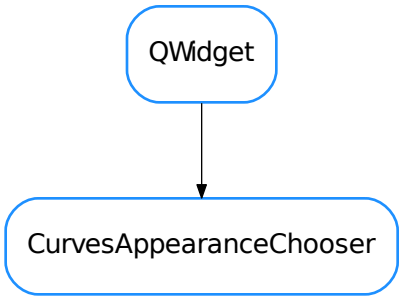
resets the table re-reading the curves from the plot

restorePlot (*keepMarkers=False*)
leaves the parent plot in its original state

showEvent (*event*)
See `Qwidget.showEvent()`

statColumns = ('points', 'min', 'max', 'mean', 'std', 'rms')

CurvesAppearanceChooser



class CurvesAppearanceChooser (*parent=None*, *curvePropDict={}*, *showButtons=False*, *autoApply=False*, *designMode=False*)
Bases: `PyQt4.QtGui.QWidget`

A widget for choosing plot appearance for one or more curves. The current curves properties are passed using the setCurves() method using a dictionary with the following structure:

```
curvePropDict={name1:prop1, name2:prop2,...}
```

where propX is an instance of *CurveAppearanceProperties* When applying, a signal is emitted and the chosen properties are made available in a similar dictionary.

CurveTitleEdited

NAME_ROLE = 32

controlChanged

curveAppearanceChanged

getSelectedCurveNames()
Returns the curve names for the curves selected at the curves list.

Note: The names may differ from the displayed text, which corresponds to the curve titles (this method is what you likely need if you want to get keys to use in curves or curveProp dicts).

Return type `string_list`

Returns the names of the selected curves

getShownProperties()
Returns a copy of the currently shown properties and updates self._shownProp

Return type *CurveAppearanceProperties*

Returns

loadUi (*filename=None*, *path=None*)

onApply ()
Apply does 2 things:

- It updates *self.curvePropDict* using the current values choosen in the dialog
- It emits a curveAppearanceChanged signal that indicates the names of the curves that changed and the new properties. (The names and the properties are returned by the function as well)

Return type tuple <*CurveAppearanceProperties*, list>

Returns a tuple containing the curve properties and a list of the selected curve names (as a list<str>)

onControlChanged (**args*)
slot to be called whenever a control widget is changed. It emmits a 'controlChanged signal and applies the change if in autoapply mode. It ignores any arguments passed

onItemChanged (*item*)
slot used when an item data has changed

onReset ()
slot to be called when the reset action is triggered. It reverts to the original situation

onSelectedCurveChanged ()
Updates the shown properties when the curve selection changes

setCurves (*curvePropDict*)
Populates the list of curves from the properties dictionary. It uses the curve title for display, and stores the curve name as the item data (with role=CurvesAppearanceChooser.NAME_ROLE)

Parameters **curvePropDict** (dict) – a dictionary whith keys=curvenames and values= *CurveAppearanceProperties* object

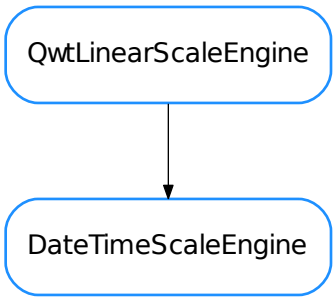
showProperties (*prop=None*)
Updates the dialog to show the given properties.

Parameters **prop** (*CurveAppearanceProperties*) – the properties object containing what should be shown. If a given property is set to None, the corresponding widget will show a "neutral" display

updateTitles (*newTitlesDict=None*)
Updates the titles of the curves that are displayed in the curves list.

Parameters **newTitlesDict** (dict <str, str>) – dictionary with key=curve_name and value=title

DateTimeScaleEngine



class **DateTimeScaleEngine**(*scaleDraw=None*)

Bases: `PyQt4.Qwt5.QwtLinearScaleEngine`

static **disableInAxis**(*plot*, *axis*, *scaleDraw=None*, *scaleEngine=None*)

convenience method that will disable this engine in the given axis. Note that it changes the ScaleDraw as well.

Parameters

- **plot** (`QwtPlot`) – the plot to change
- **axis** (`Axis`) – the id of the axis
- **scaleDraw** (`QwtScaleDraw`) – Scale draw to use. If None given, a *FancyScaleDraw* will be set
- **scaleEngine** (`QwtScaleEngine`) – Scale draw to use. If None given, a `Qwt5.QwtLinearScaleEngine` will be set

divideScale(*x1*, *x2*, *maxMajSteps*, *maxMinSteps*, *stepSize*)

Reimplements Qwt5.QwtLinearScaleEngine.divideScale

Important: The stepSize parameter is **ignored**.

Return type `QwtScaleDiv`

Returns a scale division whose ticks are aligned with the natural time units

static **enableInAxis**(*plot*, *axis*, *scaleDraw=None*, *rotation=None*)

convenience method that will enable this engine in the given axis. Note that it changes the ScaleDraw as well.

Parameters

- **plot** (`QwtPlot`) – the plot to change
- **axis** (`Axis`) – the id of the axis
- **scaleDraw** (`QwtScaleDraw`) – Scale draw to use. If None given, the current ScaleDraw for the plot will be used if possible, and a *TaurusTimeScaleDraw* will be set if not
- **rotation** (`float` or `None`) – The rotation of the labels (in degrees, clockwise-positive)

static getDefaultAxisLabelsAlignment(*axis*, *rotation*)
return a "smart" alignment for the axis labels depending on the axis and the label rotation

Parameters

- **axis** (Axis) – the axis
- **rotation** (float) – The rotation (in degrees, clockwise-positive)

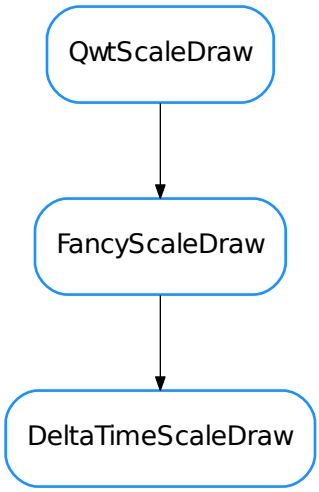
Return type Alignment

Returns an alignment

scaleDraw()

setScaleDraw(*scaleDraw*)

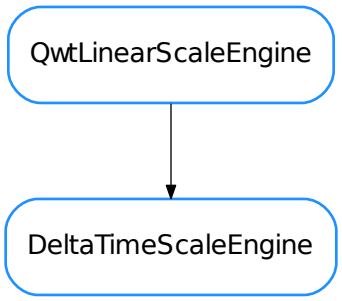
DeltaTimeScaleDraw



class DeltaTimeScaleDraw(**args*)
Bases: taurus.qt.qtgui.plot.scales.FancyScaleDraw

label(*val*)

DeltaTimeScaleEngine



class **DeltaTimeScaleEngine**(*scaleDraw=None*)

Bases: PyQt4.Qwt5.QwtLinearScaleEngine

static **disableInAxis**(*plot*, *axis*, *scaleDraw=None*, *scaleEngine=None*)

convenience method that will disable this engine in the given axis. Note that it changes the ScaleDraw as well.

Parameters

- **plot** (QwtPlot) – the plot to change
- **axis** (Axis) – the id of the axis
- **scaleDraw** (QwtScaleDraw) – Scale draw to use. If None given, a *FancyScaleDraw* will be set
- **scaleEngine** (QwtScaleEngine) – Scale draw to use. If None given, a Qwt5.QwtLinearScaleEngine will be set

divideScale(*x1*, *x2*, *maxMajSteps*, *maxMinSteps*, *stepSize*)

Reimplements Qwt5.QwtLinearScaleEngine.divideScale

Return type QwtScaleDiv

Returns a scale division whose ticks are aligned with the natural delta time units

static **enableInAxis**(*plot*, *axis*, *scaleDraw=None*, *rotation=None*)

convenience method that will enable this engine in the given axis. Note that it changes the ScaleDraw as well.

Parameters

- **plot** (QwtPlot) – the plot to change
- **axis** (Axis) – the id of the axis
- **scaleDraw** (QwtScaleDraw) – Scale draw to use. If None given, the current ScaleDraw for the plot will be used if possible, and a *TaurusTimeScaleDraw* will be set if not
- **rotation** (float or None) – The rotation of the labels (in degrees, clockwise-positive)

static getDefaultAxisLabelsAlignment(*axis*, *rotation*)
return a "smart" alignment for the axis labels depending on the axis and the label rotation

Parameters

- **axis** (Axis) – the axis
- **rotation** (float) – The rotation (in degrees, clockwise-positive)

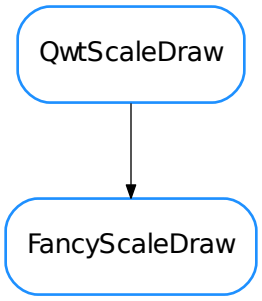
Return type Alignment

Returns an alignment

scaleDraw()

setScaleDraw(*scaleDraw*)

FancyScaleDraw



class FancyScaleDraw(*format=None*, *palette=None*)
Bases: PyQt4.Qwt5.QwtScaleDraw

This is a scaleDraw with a tuneable palette and label formats

draw(*painter*, *palette*)

getLabelFormat()
pass a format string (e.g. "%g") or None to use default (it uses the locale)

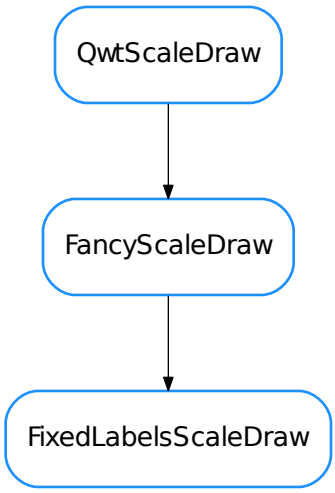
getPalette()

label(*val*)

setLabelFormat(*format*)
pass a format string (e.g. "%g") or None to use default (it uses the locale)

setPalette(*palette*)
pass a QPalette or None to use default

FixedLabelsScaleDraw

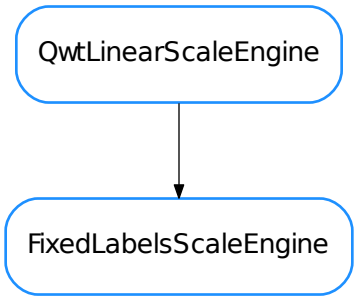


class FixedLabelsScaleDraw (*positions*, *labels*)

Bases: `taurus.qt.qtgui.plot.scales.FancyScaleDraw`

label (*val*)

FixedLabelsScaleEngine



class FixedLabelsScaleEngine (*positions*)

Bases: `PyQt4.Qwt5.QwtLinearScaleEngine`

static disableInAxis (*plot*, *axis*, *scaleDraw=None*, *scaleEngine=None*)

convenience method that will disable this engine in the given axis. Note that it changes the ScaleDraw as

well.

Parameters

- **plot** (QwtPlot) – the plot to change
- **axis** (Axis) – the id of the axis
- **scaleDraw** (QwtScaleDraw) – Scale draw to use. If None given, a *FancyScaleDraw* will be set
- **scaleEngine** (QwtScaleEngine) – Scale draw to use. If None given, a Qwt5.QwtLinearScaleEngine will be set

divideScale(*x1*, *x2*, *maxMajSteps*, *maxMinSteps*, *stepSize=0.0*)

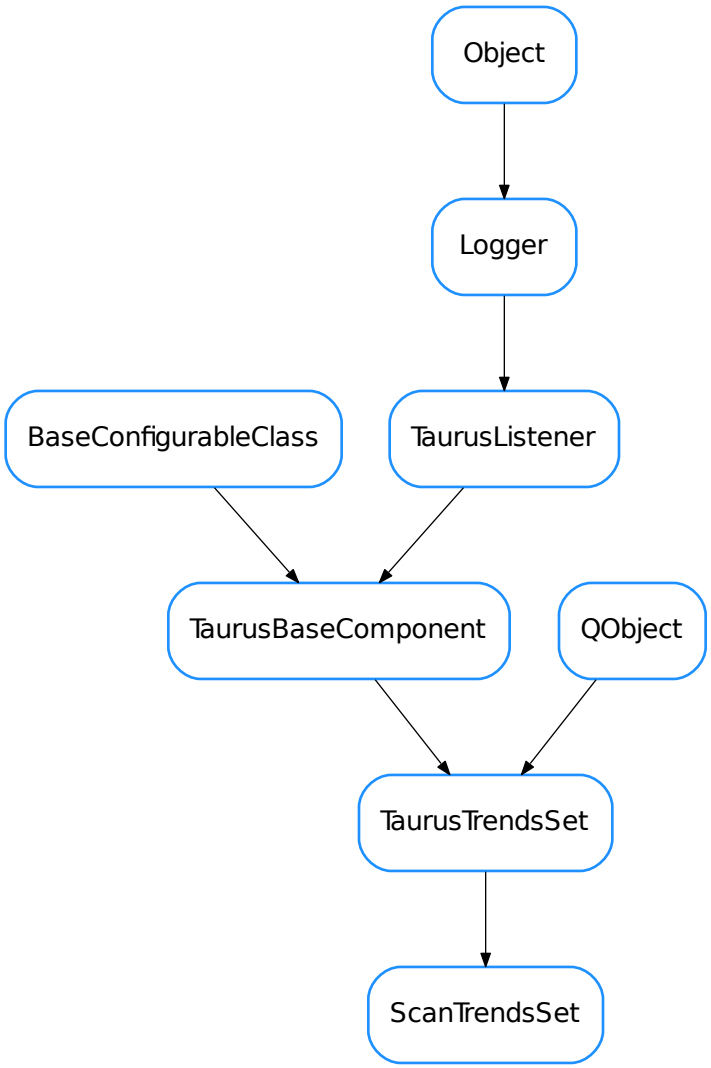
static **enableInAxis**(*plot*, *axis*, *scaleDraw=None*)

convenience method that will enable this engine in the given axis. Note that it changes the ScaleDraw as well.

Parameters

- **plot** (QwtPlot) – the plot to change
- **axis** (Axis) – the id of the axis
- **scaleDraw** (QwtScaleDraw) – Scale draw to use. If None given, the current ScaleDraw for the plot will be used if possible, and a *FixedLabelsScaleDraw* will be set if not

ScanTrendsSet



class ScanTrendsSet (*name*, *parent=None*, *autoClear=True*, *xDataKey=None*)

Bases: `taurus.qt.qtgui.plot.taurustrend.TaurusTrendsSet`

An specialized TaurusTrendSet that instead of being updated via events, it receives new data directly via a PyQt slot

receives signal containing record data from a scan.

When an event is received, all curves belonging to a TaurusTrendSet are updated.

Note that internally each curve is treated as a RawData curve (i.e., it is not aware of events by itself, but it relies on the ScanTrendSet object to update its values)

See also:

`TaurusTrendSet`

DEFAULT_X_DATA_KEY = 'point_nb'

clearTrends(*replot=True*)
Reimplemented from *TaurusTrendsSet.clearTrends()*.

Note: If the autoClear property is True for this trend set, this method is called automatically every time a data_desc package is received.

---

connectWithQDoor(*qdoor*)
connects this ScanTrendsSet to a QDoor

Parameters **qdoor** (`QDoor` or `str`) – either a QDoor instance or the QDoor name

dataChanged

disconnectQDoor(*qdoor*)
connects this ScanTrendsSet to a QDoor

Parameters **qdoor** (`QDoor` or `str`) – either a QDoor instance or the QDoor name

getDataDesc()

getModel()

onPlotablesFilterChanged(*flt*)
slot to be called whenever the plotables filter is changed. It will call *clearTrends()* if flt is None

Parameters **flt** (`list <method>`) –

scanDataReceived(*packet*)
packet is a dict with {type:str, "data":object} and the accepted types are: data_desc, record_data, record_end and the data objects are: seq<ColumnDesc.Todict()>, record.data dict and dict , respectively

setAutoClear(*enable*)

setEndMacroMarkerEnabled(*enable*)
Sets whether a marker should be put at the end of each macro or not

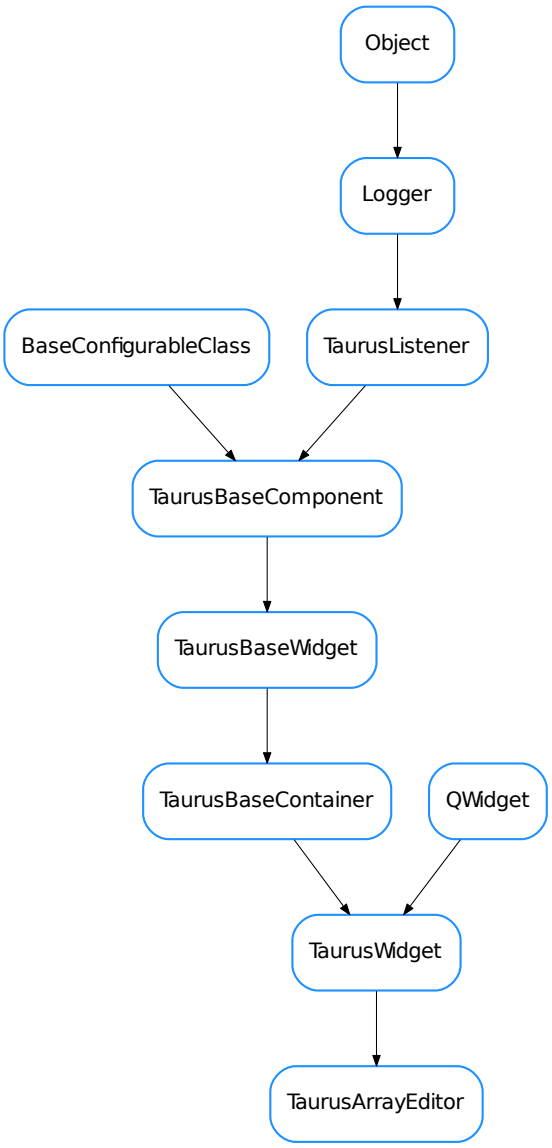
Parameters **enabled** (`bool`) –

setModel(*model*)

setPlotablesFilter(*flt*)

setXDataKey(*key*)

TaurusArrayEditor



class **TaurusArrayEditor** (*parent=None*, *designMode=False*)

Bases: `taurus.qt.qtgui.container.tauruswidget.TaurusWidget`

arrayEditor()

classmethod **getQtDesignerPluginInfo**()

Returns pertinent information in order to be able to build a valid QtDesigner widget plugin

Return type `dict`

Returns a map with pertinent designer information

onFromAttr(*quiet=False*)
reads the Master curve from the attributes set by model.

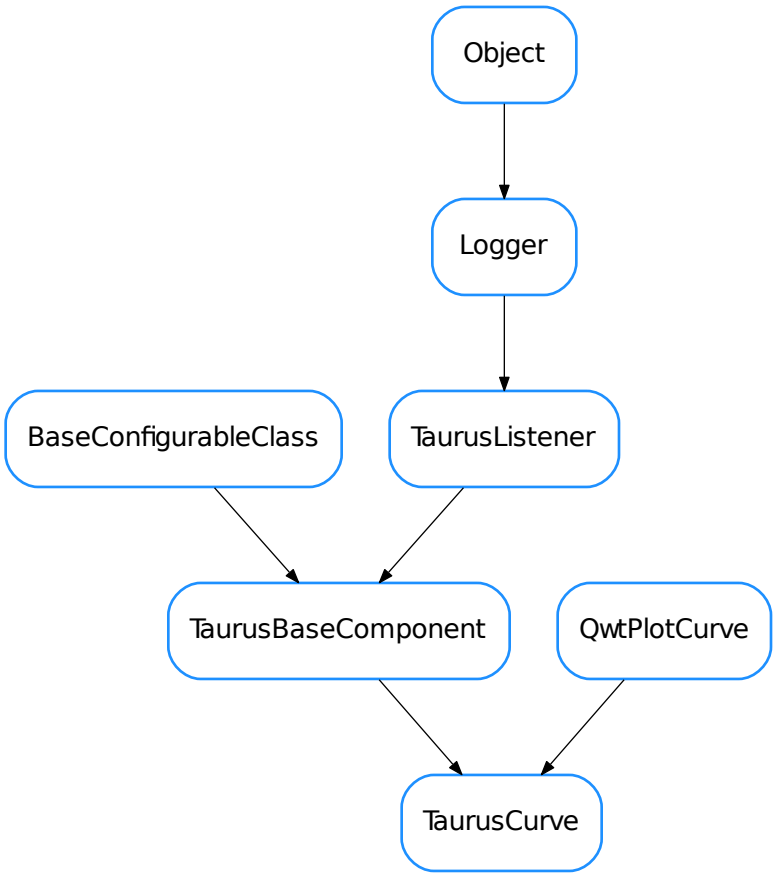
onFromFile(*filename=None*, ***kwargs*)
imports Master curve from a two-column ASCII file. The first colum will be interpreted to be the abcissas. If filename is not given, a dialog for choosing a file is presented kwargs can contain keyword arguments to pass to numpy.loadtxt() when reading each file accepted keywords and their default values are: {dtype=<type 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, usecols=None, unpack=False} see help from numpy.loadtxt for more info on the kwargs

onToAttr(*quiet=False*)
writes the Corrected curve to the attributes set by the model

onToFile()
writes the Corrected curve to an ascii file

setModel(*model*)
returns True if a curve could be set from the attribute. Flase otherwise

TaurusCurve



class TaurusCurve (*name*, *xname=None*, *parent=None*, *rawData=None*, *optimized=False*)

Bases: `PyQt4.Qwt5.QwtPlotCurve`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseComponent`

Taurus-enabled custom version of QwtPlotCurve.

TaurusCurves are attached to *TaurusPlot* objects for displaying 1D data sets.

A TaurusCurve is more complex than simple QwtPlotCurve in that:

- It is taurus-aware (i.e., it is associated to a taurus model (an attribute) and listens to Taurus events to update its data
- They may have an associated *TaurusXValues* object that controls the values for its abscissas.
- It uses a *CurveAppearanceProperties* object to manage how it looks

Important:

The TaurusPlot is in charge of attaching and detaching its TaurusCurves, and keeps information about which TaurusCurves are attached. Therefore the programmer should never attach/detach a TaurusCurve manually.

attachMarkers(*plot*)
attach markers to the plot

Parameters **plot** – the plot (typically, the TaurusPlot instance)

attachMaxMarker(*plot*)
attach marker of max value to the plot

Parameters **plot** – the plot (typically, the TaurusPlot instance)

attachMinMarker(*plot*)
attach markers of min value to the plot

Parameters **plot** – the plot (typically, the TaurusPlot instance)

compileTitleText(*titletext*)
Substitutes the known placeholders by the current equivalent values for a titleText.

Note: Some placeholders may not make sense for certain curves (e.g. <label> for a RawData curve). In these cases, they are left unprocessed (without warning).

Parameters **titletext** (str) – A string which can contain any of the following predefined placeholders:

- <label> the attribute label (default)
- <model> the model name
- <attr_name> attribute name
- <attr_fullname> full attribute name (for backwards compatibility, <attr_full_name> is also accepted)
- <dev_alias> device alias
- <dev_name> device name
- <dev_fullname> full device name (for backwards compatibility, <dev_full_name> is also accepted)
- <current_title> The current title

Return type str

Returns a title string where the placeholders have been substituted by their corresponding values

consecutiveDroppedEventsWarning = 3

dataChanged
Base signal dataChanged

detach()
reimplemented from QwtPlotCurve. In addition to dettaching the curve, it dettaches the associated min/max markers.

detachMarkers()
detaches the min/max markers of this curve

detachMaxMarker()
detaches the max marker of this curve

detachMinMarker()
detaches the min marker of this curve

droppedEventsWarning = -1

getAppearanceProperties()
Returns the appearance properties of the curve (color, symbol, width,...).

Return type *CurveAppearanceProperties*

Returns

getCurveName()
Returns the name of the curve (in the case of non RawDataCurves, it is the same as the model name)

Return type str

Returns

getModelClass()
See TaurusBaseComponent.getModelClass()

getParentTaurusComponent()
Searches the closest ancestor (in the Qt parenting hyerarchy) that is which inherits from TaurusBaseComponent. It returns None if None found.

Return type widget or None

Returns

getRawData()
Returns the rawData

Return type dict or None

Returns a RawData dict or None if the curve is not RawData

See also:

TaurusPlot.attachRawData()

getStats(*limits=None*, *inclusive=(True*, *True)*, *imin=None*, *imax=None*, *ignorenans=True*)
returns a dict containing several descriptive statistics of a region of the curve defined by the limits given by the keyword arguments. It also contains a copy of the data in the considered region. The keys of the returned dictionary correspond to:

-'x' : the abscissas for the considered points (numpy.array) -'y' : the ordinates for the considered points (numpy.array) -'points': number of considered points (int) -'min' : (x,y) pair of the minimum of the curve (float,float) -'max' : (x,y) pair of the maximum of the curve (float,float) -'mean' : arithmetic average of y (float) -'std' : (biased)standard deviation of y (float) -'rms' : root mean square of y (float)

Note that some of the values may be None if that cannot be computed.

Also,

Parameters

- **limits** (None or tuple <float, float>) – tuple containing (min,max) limits. Points of the curve whose abscisa value is outside of these limits are ignored. If None is passed, the limit is not enforced
- **inclusive** (tuple <bool, bool>) – . A tuple consisting of the (lower flag, upper flag). These flags determine whether values exactly equal to the lower or upper limits are included. The default value is (True, True).
- **imin** (int) – lowest index to be considered. If None is given, the limit is not enforced
- **imax** (int) – higest index to be considered. If None is given, the limit is not enforced

- **ignorenans** (bool) – if True (defaul), the points with NaN values are stripped before calculating the stats

Return type dict

Returns A dict containing the stats.

getXValues()
Returns X values using the XValuesBuilder.

Return type sequence

Returns

See also:

setXValuesBuilder()

getYAxisStatus()
returns either None (if the curve is not visible) or its yAxis (if it is visible)

Return type Axis or None

Returns

handleEvent(*src*, *evt_type*, *val*)
Handles Taurus Events for this curve

See: TaurusBaseComponent.handleEvent()

isFilteredWhenLog()
returns True if non-possitive values are being discarded when plotting in log mode.

return: (bool)

See also:

setFilteredWhenLog()

isReadOnly()
see TaurusBaseComponent.isReadOnly()

registerDataChanged(*listener*, *meth*)
registers a listener to the DataChangedSignal of this curve

Parameters

- **listener** (QWidget) – listener object
- **meth** (callable) – callback method

safeSetData()
Calls setData with x= self._xValues and y=self._yValues

See also:

setData()

setAppearanceProperties(*prop*)
Applies the given CurveAppearanceProperties object (prop) to the curve. If a given property is set to None, it is not applied

Parameters **prop** (*CurveAppearanceProperties*) –

setData(*x*, *y*)
Sets the X and Y data for the curve (possibly filtering non-possitive values if in log mode). Reimplemented from Qwt5.QwtPlotCurve.setData.

Parameters

- **x** (sequence) – X values
- **y** (sequence) – Y values

See also:

safeSetData(), *setFilteredWhenLog()*

setFilteredWhenLog(*filtered=True*)
Set whether non-possitive values should be discarded or not when plotting in log mode.

Parameters **filtered** (bool) – if True, filtering is done

setPaused(*paused=True*)
Pauses itself and other listeners depending on it

See also:

TaurusBaseComponent.setPaused()

setTitleText(*titletext*)
Sets the title text for this curve.

Parameters **titletext** (str) – A string which can contain predefined placeholders (which make sense in the case of non-rawdata curves)

See Also : compileTitleText

setXValuesBuilder(*fn=None*)
Sets the callback to be used for creating the 'X' array values for a curve. If None given, the default is that the abscissas are int indexes (from 0 to len(Y)).

Parameters **fn** (callable) – a callable that gets the Y values as a parameter and returns X values

E.g., the default:

```
curve.setXValuesBuilder()
```

is equivalent to:

```
curve.setXValuesBuilder(lambda yVals: numpy.arange(len(yVals)))
```

setXYFromModel(*value*)
sets the X (self._xValues) and Y (self._yValues) values from the given model. This method can be reimplemented by subclasses of Taurusplot that behave differently (e.g. TaurusTrend)

Parameters **value** (TaurusAttrValue) – the value object from the model

setYAxis(*axis*)
changes the Y axis to which the curve is associated

Parameters **axis** (Axis) – the axis to which it should associate

showMaxPeak(*show*)
Specifiefs if we want to show or not the max peak of the curve

Parameters **show** (bool) –

showMinPeak(*show*)
Specifiefs if we want to show or not the min peak of the curve.

Parameters **show** (bool) –

titleText(*compiled=False*)
Returns the titleText string. If compiled == True, the returned string will be processed through compileTitleText

Parameters **compiled** (bool) – Whether to process the return value or not (default is compiled=False)

Return type basestring

Returns the title

See also:

compileTitleText()

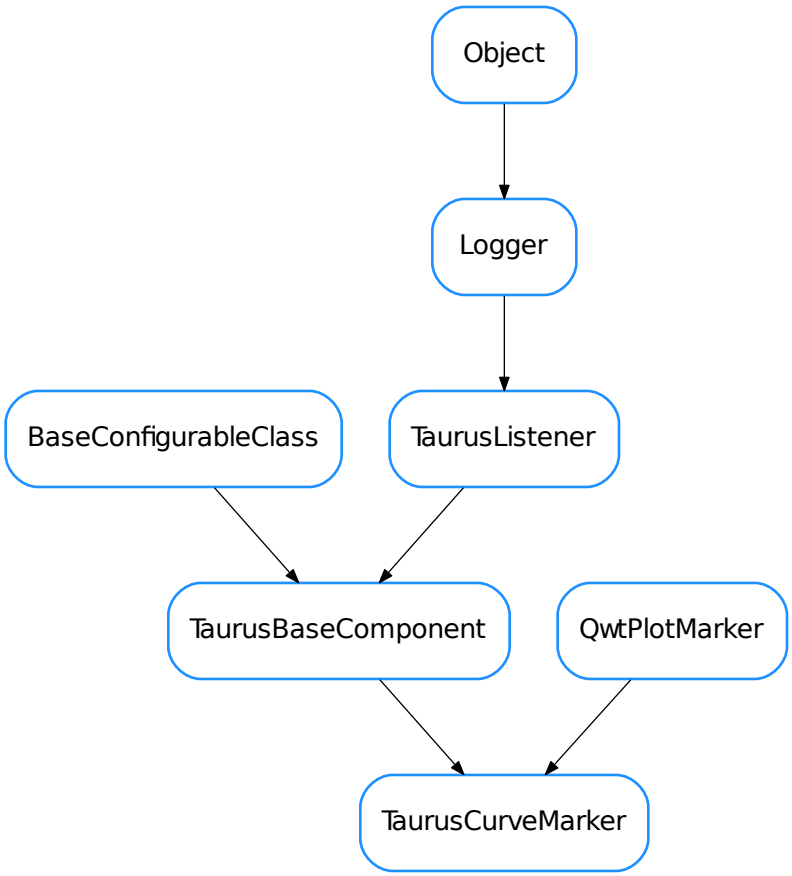
unregisterDataChanged(*listener*, *meth*)
unregisters the given listener and method from the DataChangedSignal of this curve

Parameters

- **listener** (QWidget) – listener object
- **meth** (callable) – callback method

updateTitle()
Updates the title of the curve, according to the titleText property

TaurusCurveMarker



class TaurusCurveMarker (*name*, *parent=None*, *labelOpacity=0.7*)

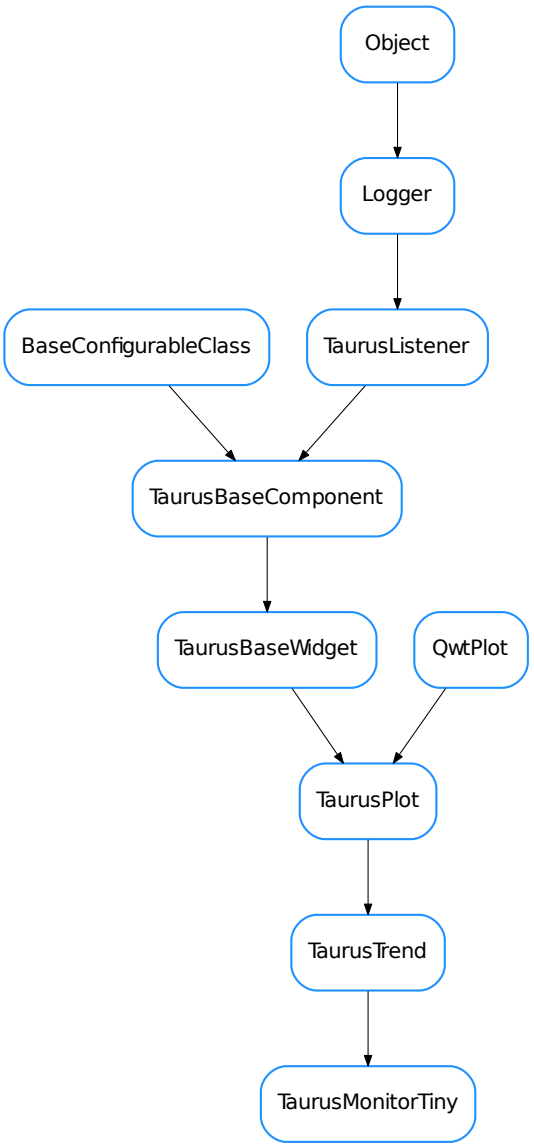
Bases: `PyQt4.Qwt5.QwtPlotMarker`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseComponent`

Taurus-enabled custom version of QwtPlotMarker

alignLabel()

Sets the label alignment in a "smart" way (depending on the current marker's position in the canvas).

TaurusMonitorTiny



class TaurusMonitorTiny (*parent=None*, *designMode=False*)

Bases: `taurus.qt.qtgui.plot.taurustrend.TaurusTrend`

A specialised *TaurusTrend* widget for monitoring scalar values and show their evolution over time. It is designed to be small (e.g. to fit in a toolbar). It is inspired by the SysMon applet in old KDE3.

See also:

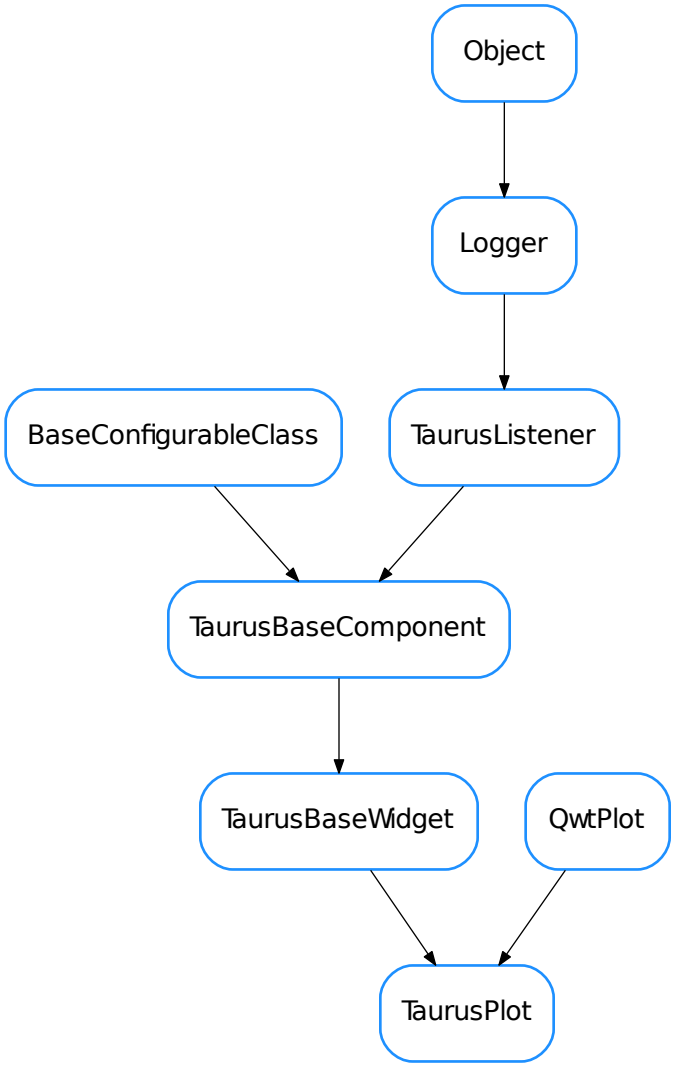
TaurusTrend, *TaurusTrend User's Interface Guide*, *The TaurusTrend coding examples*

DEFAULT_MAX_BUFFER_SIZE = 8192

autoShowYAxes()
reimplemented to avoid auto-enabling of axes

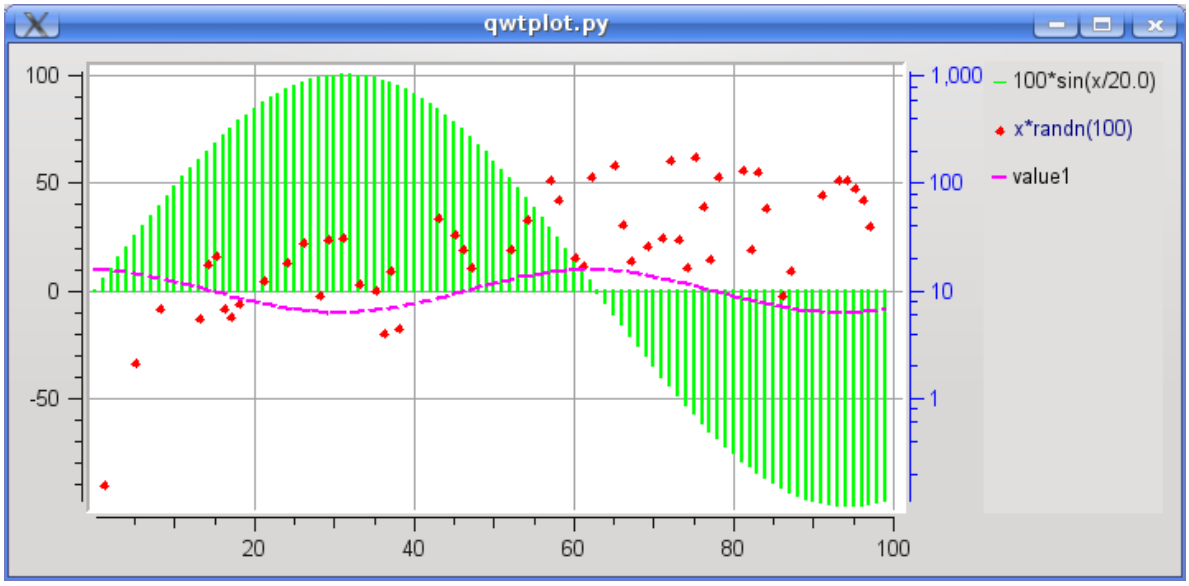
event(*event*)

TaurusPlot



class TaurusPlot(*parent=None*, *designMode=False*)
Bases: `PyQt4.Qwt5.QwtPlot`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget`

TaurusPlot is a general widget for plotting 1D data sets. It is an extended taurus-aware version of `QwtPlot`.



TaurusPlot already incorporates by default many features that can be added to a regular QwtPlot:

- •Zoomming, panning, and magnifier are enabled by default
- •Autoscaling is enabled and associated to the ESC key
- •Methods are available to add new curves which can either be associated to taurus attributes or be "raw data" (i.e., values that are not directly aware of control system events)
- •Context menu offers access to many options
- •A plot configuration dialog, and save/restore configuration facilities
- •Date-time scales and linear/log scales support
- •Methods for importing/exporting curves from/to ASCII data
- •Methods for printing and exporting the plot to PDF
- •Methods for creating curves from arbitrary functions
- •Data inspection facilities
- •...

For an overview of the features from an user point of view, see the *TaurusPlot User's Interface Guide*.

You can also see some code that exemplifies the use of TaurusPlot in *the TaurusPlot coding examples*

Important: although TaurusPlot subclasses QwtPlot and therefore it is possible to use QwtPlot's lower level methods for attaching QwtPlotItems (such as QwtPlotCurves) to the plot, it is *highly* recommended to use the higher-level methods provided by TaurusPlot to interact with the datasets attached to a TaurusPlot (e.g., *addModels()*, *attachRawData()*). This is because TaurusPlot keeps records of the items attached via its own methods.

See also:

`TaurusTrend`, *TaurusPlot User's Interface Guide*, *The TaurusPlot coding examples*

`CurvesYAxisChanged`

`addModels`

`allowZoomers`

applyAxesConfig(*axes*)
sets the axes according to settings stored in the axes dict, which can be generated with _createAxesDict()

Parameters **axes** (dict) – contains axes properties

applyConfig(*configdict*, ***kwargs*)
implemented as in TaurusBaseComponent.applyConfig()

Parameters **configdict** (dict <str, object>) –

See also:

createConfig(), TaurusBaseComponent.applyConfig()

applyMiscConfig(*miscdict*)
sets the configurations according to settings stored in the misc dict, which can be generated with _createMiscDict()

Parameters **miscdict** (dict) – Dictionary of properties

attachRawData(*rawdata*, *properties=None*, *id=None*)
attaches a curve to the plot formed from raw data that comes in a dict

Parameters

- **rawdata** (dict) – A dictionary defining a rawdata curve. It has the following structure (all keys are optional, but either "y" or "f(x)" must be present. Also, the value of x, y and f(x) can be None):

  {"title":<str>, "x":list<float>, "y":list<float>, "f(x)": <str (an expression to evaluate on the x values)>}

- **properties** (*CurveAppearanceProperties*) – appearance properties for the curve
- **id** (str) – This will be the internal name identifier used for the curve. If not given, it defaults to the title or to "rawdata" if no title is given.

Return type QwtPlotCurve

Returns the attached curve

Note: every member of the rawdata dictionary is optional except for the y values (or, alternatively, f(x) AND x)

Note: using "name" in the rawdata dictionary is a still-supported-but-deprecated synonim of "title".

autoScaleAllAxes()
Optimized autoscale of whole plot

autoShowYAxes()
shows/hides Y1 and Y2 depending of whether there are curves associated to them. Also takes care of changing the zoomer if needed

axisScaleDiv(*axis*)
Return the scale division of a specified axis.

Parameters **axis** (Axis) – the axis

Return type QwtScaleDiv

Returns scale division

changeCurvesTitlesDialog(*curveNamesList=None*)
Shows a dialog to set the curves titles (it will change the current curves titles and the default curves titles)

Parameters **curveNamesList** (sequence <str> or iterator <str>) – names of the curves to which the title will be changed (if None given , it will apply to all the curves except the raw data ones and it will also be used as default for newly created ones)

Return type caselessDict <str, str>

Returns dictionary with key=curvename and value=newtitle

See also:

setCurvesTitle(), *setDefaultCurvesTitle()*

checkConfigVersion(*configdict*, *showDialog=False*, *supportedVersions=None*)
Check if the version of configdict is supported.

Parameters

- **configdict** (dict) – configuration dictionary to check
- **showDialog** (bool) – whether to show a QtWarning dialog if check failed (false by default)
- **supportedVersions** (sequence <str> or None) – supported version numbers, if None given, the versions supported by this widget will be used (i.e., those defined in self._supportedConfigVersions)

Return type bool

Returns returns True if the configdict is of the right version

See also:

TaurusBaseComponent.checkConfigVersion()

clearAllRawData()
removes all rawdata curves from the plot.

Return type list <str>

Returns the list of removed curve names

closeEvent(*event*)
See Qwidget.closeEvent()

contextMenuEvent(*event*)
This function is called when there is context menu event. See Qwidget.closeEvent() A pop up menu will be shown with the available options. Different parts of the plot (canvas, axes,...) behave differently

createConfig(*allowUnpickable=False*, *curvenames=None*, ***kwargs*)
Returns a pickable dictionary containing all relevant information about the current plot. Implemented as in TaurusBaseComponent.createConfig() For Tango attributes it stores the attribute name and the curve properties For raw data curves, it stores the data as well.

Hint: The following code allows you to serialize the configuration dictionary as a string (which you can store as a QSetting, or as a Tango Attribute):

```
import pickle
c = pickle.dumps(taurusplot.createConfig())  #c is a string that can be stored
```

Parameters **curvenames** (sequence <str>) – a sequence of curve names for which the configuration will be stored (all by default).

Return type dict

Returns configurations (which can be loaded with applyConfig)

See also:

createConfig(), TaurusBaseComponent.createConfig()

createConfigDict(*allowUnpickable=False*, *curvenames=None*)

curveDataChanged

dataChanged

defaultCurvesTitle

detachRawData(*name*)
dettaches a raw data curve

Parameters **name** (str) – name (identifier) of the curve to dettach

dropEvent(*event*)
reimplemented to support dropping of modelnames in taurusplots

enableMagnifier

enableOptimization

enablePanner

exportAscii(*curves=None*)
Opens a dialog for exporting curves to ASCII files.

Parameters **curves** (sequence <str>) – the curves curves that will be exportable. if None given, all curves are offered for export.

exportPdf(*fileName=None*)
Export the plot to a PDF. slot for the _exportPdfAction.

Parameters **fileName** (str) – The name of the file to which the plot will be exported. If None given, the user will be prompted for a file name.

exportPrint()
Launches a QPrintDialog for printing the plot

getAllowZoomers

getAxisLabelFormat(*axis*)
Returns the label format for the given axis

Parameters **axis** (Axis) – the axis

Return type str or None

Returns

See also:

setAxisLabelFormat()

getAxisName(*axis*)
If set, it returns the axis title text, otherwise returns the default axis name

Parameters **axis** (Axis) –

Return type unicode

Returns

getAxisScale(*axis*)
returns the lower and higher bounds for the given axis, or None,None if the axis is in autoscale mode

Parameters **axis** (Axis) – the axis

Return type float,float

Returns atuple of floats (or None,None)

getAxisTransformationType(*axis*)
Retrieve the transformation type for a given axis (cached)

Parameters **axis** (Axis) – the axis

Return type Type

Returns

Note: this method helps to avoid a memory leak in Qwt (see http://sf.net/p/tauruslib/tickets/171)

---

getCurve(*name*)
gets a curve object by name.

Important: Note that the curve object is not thread safe. Therefore, if you access to the curve object you must do it protected by the TaurusPlot.curves_lock reentrant lock.

Parameters **name** (str) – the curve name

Return type *TaurusCurve*

Returns the curve object corresponding to name

getCurveAppearancePropertiesDict()
Returns the appearance properties of all curves in the plot.

Return type dict <str, *CurveAppearanceProperties*>

Returns a dictionary whose keys are the curve names and whose values are the corresponding CurveAppearanceProperties object

See also:

setCurveAppearanceProperties()

getCurveData(*curvename*, *numpy=False*)
returns the data in the curve as two lists (x,y) of values

Parameters

- **curvename** (str) – the curve name
- **numpy** (bool) – if True, the result is returned as numpy arrays instead of lists

Return type tuple <list, list>

Returns tuple of two lists (x,y) containing the curve data

getCurveNames()
returns the names of all TaurusCurves attached to the plot (in arbitrary order, if you need a sorted list, see *getCurveNamesSorted()*).

Return type list <str>

Returns a copy of self.curves.keys()

See also:

getCurveNamesSorted()

getCurveNamesSorted()
returns the names of the curves in z order (which is the one used in the legend, and in showing the curves).

Return type list <str>

Returns curve names

See also:

getCurveNames()

getCurveStats(*limits=None*, *curveNames=None*)
Shows a dialog containing descriptive statistics on curves

Parameters

- **limits** (None or tuple <float, float>) – tuple containing (min,max) limits. Points of the curve whose abscisa value is outside of these limits are ignored. If None is passed, the limit is not enforced
- **curveNames** (seq <str>) – sequence of curve names for which statistics are requested. If None passed (default), all curves are considered

Return type dict

Returns Returns a dictionary whose keys are the curve names and whose values are the dictionaries returned by *TaurusCurve.getStats()*

getCurveTitle(*curvename*)
return the current title associated to a given curve name

Parameters **curvename** (str) – the name of the curve

Return type str

Returns

getDefaultAxisLabelsAlignment(*axis*, *rotation*)
return a "smart" alignment for the axis labels depending on the axis and the label rotation

Parameters

- **axis** (Axis) – the axis
- **rotation** (float) – The rotation (in degrees, clockwise-positive)

Return type Alignment

Returns an alignment

getDefaultCurvesTitle()
See setDefaultCurvesTitle

getGrid()
returns the grid of the plot

Return type QwtPlotGrid

Returns

getGridColor()
Returns the color of the plot grid

Return type QColor

Returns

getGridWidth()
Returns the width of the grid lines

Return type int

Returns with of the gridlines (in pixels)

getLegend()
Returns the legend object of this plot

Return type QwtLegend

Returns

getLegendPosition()
returns the current legend position

Return type LegendPosition

Returns

getModel()
returns the list of model names.

Return type CaselessList <str>

Returns

See also:

setModel()

getModelObj(*idx*)
See TaurusBaseComponent.getModelObj()

getParentTaurusComponent()
See TaurusBaseComponent.getParentTaurusComponent()

getPickedMarker()
returns the marker for the picked points for this plot

Return type *TaurusCurveMarker*

Returns

getPlot()
deprecated method . Only here for backwards compatibility. It will be removed, eventually. Now you should use the TaurusPlot instance instead of TaurusPlot.getPlot()

classmethod getQtDesignerPluginInfo()
Returns pertinent information in order to be able to build a valid QtDesigner widget plugin

Return type dict

Returns a map with pertinent designer information

getUseParentModel()
See: TaurusBaseComponent.getParentModel()

getXAxisRange(*axis=2*)
same as self.axisScaleDiv(axis).range()

Parameters **axis** (Axis) – the (X) axis. (default=Qwt5.QwtPlot.xBottom)

Return type float

Returns the absolute difference between the higher and lower limits of the axis scale

getXDynScale()
Whether the current X scale is in Dynamic scaling mode

Return type bool

Returns

See also:

setXDynScale(), meth:*isXDynScaleSupported*

getXIsTime()
Returns whether the X axis is in "Time mode"

Return type bool

Returns True means the X axis is in Time mode, False

See also:

setXIsTime()

getZoomers(*axis=None*)
returns a list of the zoomer(s) associated to the given axis. If None is passed, it returns a list containing the current zoomer

Parameters **axis** (Axis) – the axis

gridColor

gridWidth

importAscii(*filenames=None*, *xcol=None*, ***kwargs*)
imports curves from ASCII files. It uses :meth:numpy.loadtxt The data in the file(s) must be formatted in columns, with possibly a header and/or commented lines. Each column in a file will be imported as an independent RawData curve (except for the column whose index is passed in xcol)

Parameters

- **filenames** (sequence <str> or None) – the names of the files to be read. If None passed, the user will be allowed to select them from a dialog. (default=None)
- **xcol** (int or None) – index of the column (starting at 0) containing the abscissas data. If None passed, the abcissa is generated as indexes starting from 0.
- ****kwargs** –

  Other keyword arguments can be passed to this method, which will be passed to numpy.loadtxt() when reading each file. Accepted keywords are:

  – dtype=<type 'float'>
  – comments='#'
  – delimiter=None
  – converters=None
  – skiprows=0
  – usecols=None
  – unpack=False

See also:

`numpy.loadtxt()`

isMagnifierEnabled

isOptimizationEnabled

isPannerEnabled

isPaused()
Returns the pause state

Return type bool

Returns

isXDynScaleSupported()
Whether this widget offers xDynScale-related options. Useful for showing-hiding them in menus and dialogs

Return type bool

Returns

See also:

setXDynScaleSupported(), *getXDynScale()*

legendPosition

loadConfig(*ifile=None*)
Reads a file stored by saveConfig() and applies the settings

Parameters **ifile** (file or string) – file or filename from where to read the configuration

Return type str

Returns file name used

minimumSizeHint()
See `QWidget.minimumSizeHint()`

model

modelChanged
Override the default modelChanged('QString') signal

onCurveAppearanceChanged(*prop*, *names*)
Applies the properties given in prop to all the curves named in names. This functions is called from the config dialog when changes are applied.

Parameters

- **prop** (*CurveAppearanceProperties*) – the properties object
- **names** (sequence <str>) – a sequence of names of curves to which the properties should be applied

onCurveStatsAction()
slot for the curveStatsAction. Allows the user to select a range and then shows curve statistics on that range.

parentModelChanged

pickDataPoint(*pos*, *scope=20*, *showMarker=True*, *targetCurveNames=None*)
Finds the pyxel-wise closest data point to the given position. The valid search space is constrained by the scope and targetCurveNames parameters.

Parameters

- **pos** (QPoint or QPolygon) – the position around which to look for a data point. The position should be passed as a Qt.QPoint (if a Qt.QPolygon is given, the first point of the polygon is used). The position is expected in pixel units, with (0,0) being the top-left corner of the plot canvas.
- **scope** (int) – defines the area around the given position to be considered when searching for data points. A data point is considered within scope if its manhattan distance to position (in pixels) is less than the value of the scope parameter. (default=20)
- **showMarker** (bool) – If True, a marker will be put on the picked data point. (default=True)
- **targetCurveNames** (sequence <str>) – the names of the curves to be searched. If None passed, all curves will be searched

Return type tuple <QPointF, str, int> or tuple <None, None, None>

Returns if a point was picked within the scope, it returns a tuple containing the picked point (as a Qt.QPointF), the curve name and the index of the picked point in the curve data. If no point was found within the scope, it returns None,None,None

readFromFiles(*xcol*, *skiprows*)
helper slot. Calls self.importAscii(xcol=xcol, skiprows=skiprows) See meth:*importAscii*

removeModels

resetAllowZoomers

resetAxisLabelFormat(*axis*)
equivalent to setAxisLabelFormat(axis, None)

Parameters **axis** (Axis) – the axis

See also:setAxisLabelFormat

resetDefaultCurvesTitle()
resets the defaultCurvesTitle property to '<label>'

See also:

setDefaultCurvesTitle()

resetGridColor()
equivalent to self.setGridColor(Qt.Qt.gray)

resetGridWidth()
equivalent to self.setGridWidth(1)

resetLegendPosition()
equivalent to setLegendPosition(Qwt5.QwtPlot.RightLegend)

resetMagnifierEnabled

resetModel()
equivalent to setModel([])

resetOptimizationEnabled

resetPannerEnabled

resetUseParentModel()
equivalent to setUseParentModel(False)

resetXIsTime()
equivalent to setXIsTime(False)

saveConfig(*ofile=None*, *curvenames=None*)
Stores the current curves and their display properties in a file for later retrieval.

Parameters

- **ofile** (file or string) – file or filename to store the configuration. If None passed,
- **curvenames** (list <str>) – a list of curve names for which the configuration will be stored (all by default).

Return type str

Returns file name used

selectXRegion(*axis=2*, *callback=None*)
Changes the input mode to allow the user to select a region of the X axis

Parameters

- **axis** (xBottom or xTop) – on which the region will be defined (Default=Qwt5.QwtPlot.xBottom)
- **callback** (method) – a function that will be called when the user finishes selecting the region. If None passed (default) nothing is done

setAllowZoomers

setAxesLabelFormat(*format=None*, *xformat=None*, *y1format=None*, *y2format=None*)
Convenience method for setting the format of any or all axes if format=None, specific formats for x, y1 and y2 can be explicitly set, e.g:

```
setAxesLabelFormat("%6.2f") #<--sets the "%6.2f" format for all axes
setAxesLabelFormat(xformat=None, y1format="%i") #<--sets the default format␣
↪for x and an integer format fotr y1
```

Parameters

- **format** (str) – format string to be applied to all axes. If None, the default format is used
- **xformat** (str) – format string to be applied to the X axis. If None, the default format is used
- **y1format** (str) – format string to be applied to the Y1 axis. If None, the default format is used
- **y2format** (str) – format string to be applied to the Y2 axis. If None, the default format is used

See also:

setAxisLabelFormat()

setAxisAutoScale(*axis*)
Sets the axis to autoscale and resets the zoomer for that axis if needed

Parameters **axis** (Axis) – the axis

See also:

autoScaleAllAxes()

setAxisCustomLabels(*axis*, *pos_and_labels*, *rotation=0*, *alignment=None*)
By calling this method, the scale vaues can be substituted by custom labels at arbitrary positions. In general, it is a good idea to let the alignment to be autocalculated.

Parameters

- **axis** (Axis) – the axis
- **pos_and_labels** (sequence <tuple>) – a sequence of position(<float>),label(<str>) tuples
- **rotation** (float) – rotation value for the labels (in degrees, clockwise-positive, by default it is 0)
- **alignment** (Alignment) – an alignment for the labels. If None given, it will be autocalculated

setAxisLabelFormat(*axis*, *format=None*)
changes the format of an axis label. format is a python format string (e.g., "%6.2f"), . If format=None, the default behaviour is set (which uses QLocale.system().toString(value))

Parameters

- **axis** (Axis) – the axis
- **format** (str) – format string to be applied to all axes. If None, the default format is used

setAxisScale(*axis*, *min*, *max*)
Rescales the given axis to the range defined by min and max. If min and max are None, autoscales. It also takes care of resetting the affected zoomer(s)

Parameters

- **axis** (Axis) – the axis
- **min** (float or None) – minimum value for the axis
- **max** (float or None) – maximum value for the axis

Example:

```
tt=TaurusTrend()
tt.setAxisScale(tt.yLeft, 0, 10) #this will set the Y1 axis range from 0 to 10
tt.setAxisScale(tt.xBottom, None, None) #This will autoscale the X axis
```

setAxisScaleEngine(*axis*, *scaleEngine*)
reimplemented from Qwt5.QwtPlot.setAxisScaleEngine() to store a cache of the transformation type

setAxisScaleType(*axis*, *scale=None*)
sets the type of scale, (log or linear) for a given axis, If scale is None, the scale type will be toggled

Parameters

- **axis** (Axis) – the axis
- **scale** (Type) – the scale transformation. For convenience, the strings "Linear" and "Logarithmic" can be used as well

setCurveAppearanceProperties(*propDict*)
It gets a dictionary of namecurvenames,properties and applies the properties to the corresponding curves.

Parameters **propDict** (dict <str, *CurveAppearanceProperties*>) – a dictionary whose keys are the curve names and whose values are the corresponding CurveAppearance-Properties object

See also:

getCurveAppearancePropertiesDict()

setCurvesTitle(*titletext*, *curveNamesList=None*)
Changes the titles of current curves.

Parameters

- **titletext** (str) – string to use as title for the curves. It may include placeholders as those defined in TaurusCurve.compileTitleText()
- **curveNamesList** (sequence <str> or iterator <str>) – names of the curves to which the title will be changed (if None given , it will apply to all the curves except the raw data ones)

Return type caselessDict <str, str>

Returns dictionary with key=curvename and value=newtitle

See also:

changeCurvesTitlesDialog(), *setDefaultCurvesTitle()*, *TaurusCurve.setTitleText()*

setCurvesYAxis(*curvesNamesList*, *axis*)
Change the Y axis of the given curves to the given axis.

Parameters

- **curvesNamesList** (list <str>) – the names of the curves whose Y axis is to be changed
- **axis** (Axis) – the axis

setDefaultCurvesTitle(*titletext*)
sets the default title to be used for curves attached to this plot (the title is used, for example in the legend). Note that this does not affect to already existing curves. If you want that, see setCurvesTitle.

Parameters **titletext** (str) – the default text to be used for the titles of curves. It may contain any of the placeholders described in TaurusCurve.setTitleText

See also:

setCurvesTitle(), *TaurusCurve.setTitleText()*

setEventFilters(*filters=None*, *curvenames=None*, *preqt=False*)
propagates a list of taurus filters to the curves given by curvenames. See TaurusBaseComponent.setEventFilters()

setGridColor

setGridWidth

setLegendPosition

setMagnifierEnabled

setModel

setOptimizationEnabled

setPannerEnabled

setPaused(*paused=True*)
delegates the pausing to the curves

Parameters **paused** (bool) – if True, the plot will be paused

setUseParentModel

setXDynScale(*enabled=True*)
it enables/disables the Dynamic scaling feature (also known as Fixed-range X scale, or "auto-scroll mode"). The Dynamic scaling consists in ensuring that:

•the range (=max-min) is always constant

•the latest point plotted is always within range.

Parameters **enabled** (bool) – if True, the Dynamic scaling is enabled for the X axis. Otherwise it is disabled. (Default=True)

See also:

getXDynScale(), *setXDynScaleSupported()*

setXDynScaleSupported(*supported*)
Whether this widget should offer xDynScale-related options in menus and dialogs.

Parameters **supported** (bool) – if True, the options related to xDynScale will be shown

See also:

isXDynScaleSupported(), *getXDynScale()*

setXIsTime(*enable*, *axis=2*)
Specifies whether we the plot is in Time or in normal mode (i.e, whether the abscissas should be interpreted as unix epoch values or not)

Parameters

- **enable** (bool) – if True, the plot will be in Time Mode
- **axis** (xBottom or xTop) – the X axis to which this setting applies. (Default=Qwt5.QwtPlot.xBottom)

See also:

TaurusPlot user manual

showConfigDialog()
Slot for the showConfigMenuAction. Launches the plot configuration dialog.

showCurve(*curve*, *on*)
switch visibility of a curve (as well as any markers associated to it) on/off

Important: This is a non-thread safe method. Do not manipulate curve objects without protecting the access with Taurusplot.curves_lock

Parameters

- **curve** (*TaurusCurve*) – the curve object
- **on** (bool) – if True, the curve will be shown otherwise it will be hidden

showDataImportDlg()
Launches the data import dialog. This dialog lets the user manage which attributes are attached to the plot (using TaurusModelChooser) and also to generate raw data or import it from files

showLegend(*show*, *forever=True*)
whether to show or not the legend.

Parameters

- **show** (bool) – if True, the legend will be shown
- **forever** (bool) – if True, the setting will be permant (e.g., the legend won't be hidden even if only one curve is plotted) (default=True)

showMaxPeaks(*show*)
This function will set the showMaxPeak flag of all the curves in the plot.

Parameters **show** (bool) – if True, the max values of the displayed curve(s) will be shown on the plot. Otherwise, they will be hidden.

showMinPeaks(*show*)
This function will set the showMinPeak flag of all the curves in the plot.

Parameters **show** (bool) – if True, the min values of the displayed curve(s) will be shown on the plot. Otherwise, they will be hidden.

sizeHint()
See QWidget.sizeHint()

sortCurves(*ordered=None*)
Sorts the attached curves in a given z order. This affects both the ordering in the legend and the visibility order when curves overlap in the plotting area. The order is governed by the *ordered* parameter (or alphabetically if no parameter is passed).

Parameters **ordered** (list <str> or None) – A list of curve names in the desired order. If None passed, the items will be ordered alphabetically according to their title.

toggleCurveState(*curve*)
cycles through 3 possible states for a curve:

•invisible

•attached to Y1

•attached to Y2

Parameters **curve** (*TaurusCurve*) – the curve object

toggleDataInspectorMode(*enable=None*)
Enables/Disables the Inspector Mode. When "Inspector Mode" is enabled, the zoomer is disabled and clicking on the canvas triggers a search of a nearby data point using pickDataPoint (the cursor changes to indicate the mode).

Parameters **enable** (bool or None) – If True, it enables the Inspector Mode. If False, it disables it. If None passed, it toggles the mode.

Return type bool

Returns whether the inspector mode has been enabled (True) or disabled (False)

toggleZoomer(*axis=None*)
changes the current zoomer to that associated to the given axis (zoomer1 is attached to Y1 and zoomer2 to Y2). If no axis is passed, the zoomers are toggled.

Parameters **axis** (Axis or None) – axis to activate for zooming. If None passed, the zoomers are toggled.

Return type Axis

Returns the Y axis of the enabled zoomer

updateCurves(*names*)

Updates the TaurusCurves being plotted. It adds a new curve for each new curve model passed and removes curves if they are not in the names.

Parameters **names** (sequence <str>) – a sequence of curve models. One curve will be created for each element of names. Each curve model can consist of a single attribute name (which will be used for the Y values) or by two attribute names separated by a '|' (in which case, the left-hand attribute is used for the X values and the right hand value for the Y values)

updateLegend(*force=False*)

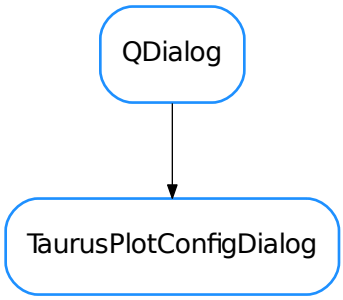
Updates the legend object of the plot (if it does not exist, it may create a fresh one)

Parameters **force** (bool) – if True, the legend will be updated even if it is not being shown. (default=False)

useParentModel

xIsTime

TaurusPlotConfigDialog



class **TaurusPlotConfigDialog**(*parent=None*, *flags=<PyQt4.QtCore._Mock object>*)

Bases: PyQt4.QtGui.QDialog

This class is used to build and manage the plot configuration dialog. It has been designed using the qt designer application, and then loaded to this widget. Hence, if you need to modify the dialog, you can use the TaurusPlotConfigDialog.ui file (under ui directory) to make it easier.

apply()

This will apply the values set in the dialog. Note that some of them are not necessary to be set, since they're already are set when changing the item selected

changeBackgroundColor()

Launches a dialog for choosing the parent's canvas background color

deltatime2str(*dt*, *fuzzy=False*)
converts a time diff in secs to a string. If fuzzy=True it returns an approx time diff in s, min, hours or days

loadUi(*filename=None*, *path=None*)

modeComboChanged(*itemSelected*)
This will catch the combo box selection change and will set the corresponding axis scale to the value passed as parameter

onChangeTitles()
Calls The parent's changeCurvesTitlesDialog method, with the selected curves list as the parameter

onCurveTitleEdited(*name*, *newTitle*)
slot used when a curve title is edited

Parameters

- **name** (`QString`) – curve name
- **name** – new title

peaksComboChanged(*itemSelected*)
This will catch the combo box selection change and will set the corresponding axis to show peaks

setCurvesYAxis(*curvesNamesList=None*, *axis=None*)
calls the parent's setCurvesYAxis method but it automatically determines the parameters if not given

setXDynScale(*checked*)

showCalendar(*target*)

str2deltatime(*strtime*)
Translates a time string to seconds examples of valid relative times are: "now", "NOW", "Now", "-1d", "3w", "- 3.6e3 s",... examples of non-valid relative times:, "now + 2h", "-5", "3H" (unit names are case-sensitive)

strtime2epoch(*strtime*)
Translates a str into an epoch value. It accepts "absolute" time notation as well as "relative to current time notation" (by expliciting a "+" or "-" prefix) (see str2deltatime for relative time notation).

examples of valid absolute times: "2008-3-25 14:21:59", "25/03/08 14:21", "03-25-2008",...

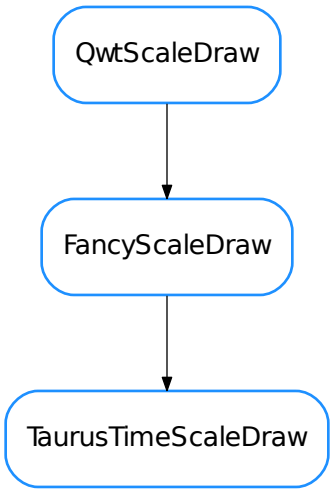
It returns None if strtime couldn't be interpreted

toggledAutoScale(*toggled*)
This will catch the group boxes check/uncheck event, and will enable autoscale in case the event has been unchecking 'disable autoscale'

validate()
validates the inputs in the dialog. If validation is ok, it returns a tuple containing min/max values for each axis (None if they are autoscaled) If validation failed, it returns False.

Note: the values of the max/min boxes are already validated thanks to their attached QDoubleValidators (except for xMin/xMax in time Mode, but this case is handled by strtime2epoch)

TaurusTimeScaleDraw



class **TaurusTimeScaleDraw**(**args*)

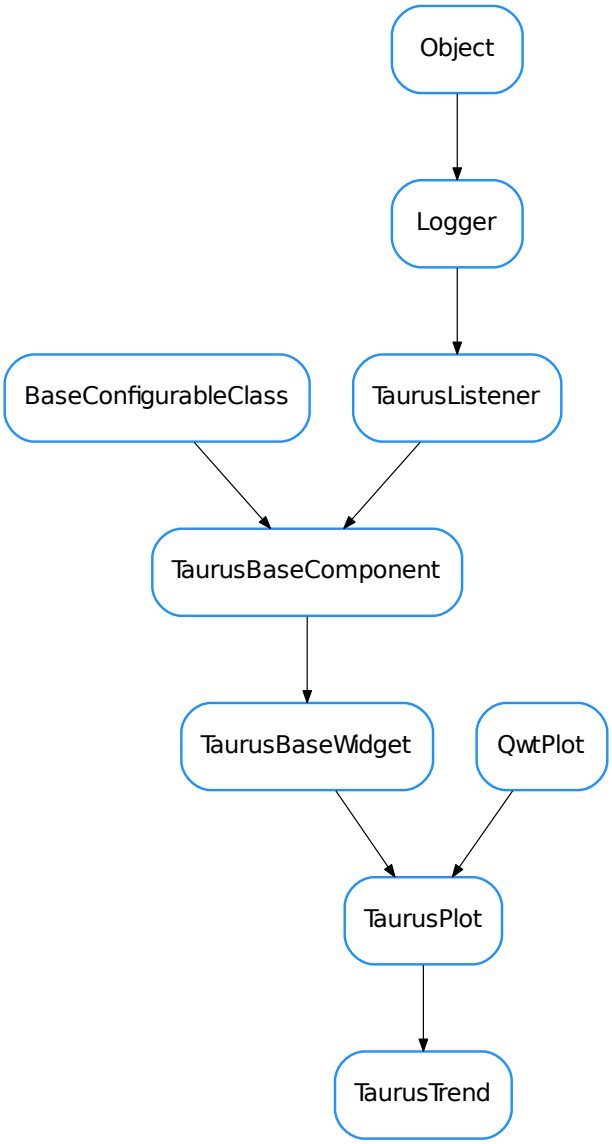
Bases: `taurus.qt.qtgui.plot.scales.FancyScaleDraw`

datetimeLabelFormat()

label(*val*)

setDatetimeLabelFormat(*format*)

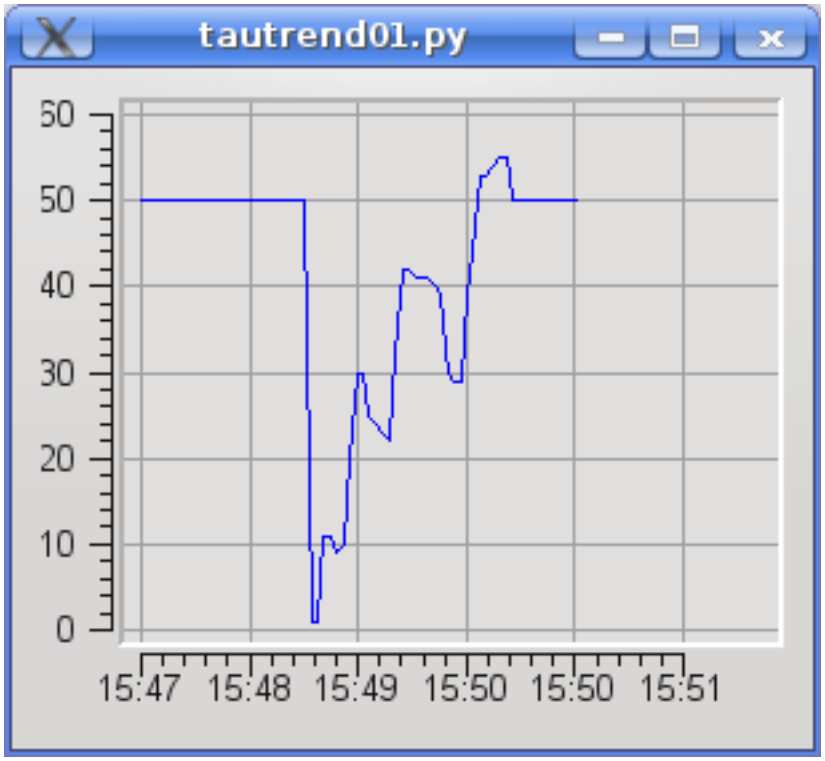
TaurusTrend



class TaurusTrend (*parent=None*, *designMode=False*)

Bases: `taurus.qt.qtgui.plot.taurusplot.TaurusPlot`

A *TaurusPlot* -derived widget specialised in plotting trends (i.e., evolution of parameters).



TaurusTrend inherits all the features from TaurusPlot (zooming, exporting/importing, data inspection,...) and also provides some specific features (e.g. fixed-range X scale mode and Archiving support).

For an overview of the features from an user point of view, see the *TaurusTrend User's Interface Guide*.

You can also see some code that exemplifies the use of TaurusTrend in *the TaurusTrend coding examples*

Note: if you pass a model that is a 1D attribute (instead of a scalar), TaurusTrend will interpret it as a collection of scalar values and will plot a separate trend line for each.

Note 2: As an special case, you can pass a model of the type scan://doorname. This will link the TaurusTrend to the given Taurus door and will listen to it for scan record events, which will be plotted.

See also:

TaurusPlot, *TaurusTrend User's Interface Guide*, *The TaurusTrend coding examples*

DEFAULT_MAX_BUFFER_SIZE = 65536

applyConfig(*configdict*, ***kwargs*)
applies the settings stored in a configdict to the current plot.

Parameters **configdict** (dict) –

See also:

createConfig()

changeCurvesTitlesDialog(*curveNamesList=None*)
Shows a dialog to set the curves titles (it will change the current curves titles and the default curves titles)

Parameters **curveNamesList** (string_sequence or string_iterator) – names of the curves to which the title will be changed (if None given , it will apply to all the TrendsSets and it will also be used as default for newly created ones)

Return type caselessDict <str, QString> or None

Returns The return value will be *None* if *curveNamesList* is None. Otherwise it will be a dictionary with key=curvename and value=newtitle.

See also:

`setCurvesTitle()`, `setDefaultCurvesTitle()`

clearBuffers()
clears the buffers of existing trend sets (note that this does not remove the models, it simply removes all stored data)

clearScan(*scanname*)
resets the curves associated to the given scan

Parameters **scanname** (`str`) – the scan model name (e.g. "scan://a/b/c")

createConfig(*tsnames=None*, ***kwargs*)
Returns a pickable dictionary containing all relevant information about the current plot. For Taurus attributes it stores the attribute name and the curve properties For raw data curves, it stores the data as well.

Hint: The following code allows you to serialize the configuration dictionary as a string (which you can store as a QSetting, or as a Taurus Attribute):

```
import pickle
c = pickle.dumps(taurusplot.createConfig())  #c is a string that can be stored
```

Parameters **names** (`sequence <str>`) – a sequence of TrendSet names for which the configuration will be stored (all by default).

Return type `dict`

Returns configurations (which can be loaded with applyConfig)

curveDataChanged

dataChanged

doReplot()
calls `replot()` only if there is new data to be plotted

forcedReadingPeriod

getCurveTitle(*name*, *index=None*)
reimplemented from *TaurusPlot*. Returns the title of a curve from a trendset

Parameters

- **name** (`str`) – The name of the trendset. If the name is not a known trendset name and index is None, we will try with tsetname and index obtained from parsing the given name (assuming the format '<tsetname>[<index>]').
- **index** (`int` or `None`) – the index of the curve in the trend set. If None is passed, it returns the base title of the trendset

Return type `str`

Returns the title

getForcedReadingPeriod(*tsetname=None*)
returns the forced reading period for the given trend (or the general period if None is given)

Parameters **tsetname** (`str` or `None`) – name of the trend set for which the forced reading should be returned. If None passed, the default period for all curves is returned

getMaxDataBufferSize()
returns the maximum number of events that can be plotted in the trend

Return type int

Returns

classmethod getQtDesignerPluginInfo()
Returns pertinent information in order to be able to build a valid QtDesigner widget plugin

Return type dict

Returns a map with pertinent designer information

getScansAutoClear()

getScrollStep()
returns the value of the scroll step

Return type float

Returns

getTrendSet(*name*)
gets a trend set object by name.

Important: Note that the TrendSet object is not thread safe. Therefore, if you access it you must do it protected by the TaurusTrend.curves_lock reentrant lock.

Parameters **name** (str) – the trend set name

Return type TaurusTrendSet

Returns the trend set object corresponding to name

getTrendSetNames()
returns the names of all TrendSets attached to this TaurusTrend.

Return type list <str>

Returns a copy of self.trendSets.keys()

getUseArchiving()
whether TaurusTrend is looking for data in the archiver when needed

Return type bool

Returns

See also:

setUseArchiving()

getUsePollingBuffer()
whether TaurusTrend is looking for data in the PollingBuffer

Return type bool

Returns

See also:

setUsePollingBuffer()

hideEvent(*event*)
reimplemented from TaurusPlot.showEvent() so that the replot timer is active only when needed

isTimerNeeded(*checkMinimized=True*)
checks if it makes sense to activate the replot timer. The following conditions must be met:

•the replot timer must exist

•the area of the plot must be non-zero

•at least one trendset must be attached

•the plot should be visible

•the plot should not be minimized (unless checkMinimized=False)

Parameters **checkMinimized** (bool) – whether to include the check of minimized (True by default)

Return type bool

Returns

maxDataBufferSize

onChangeXDataKeyAction()

onScanPlotablesFilterChanged(*flt*, *scanname=None*)

rescheduleReplot(*axis=2*, *width=1080*)
calculates the replotting frequency based on the time axis range. It assumes that it is unnecessary to replot with a period less than the time per pixel.

Parameters

- **axis** (Axis) – the axis to which it should associate
- **width** (int) – the approx canvas width (in pixels). The exact value could be obtained from the widget, but an order of magnitude approximation is usually ok (and cheaper). The default value is 1080 (HD ready!)

resetForcedReadingPeriod()
Equivalent to setForcedReadingPeriod(msec=-1, tsetnames=None)

resetMaxDataBufferSize()
Same as setMaxDataBufferSize(self.DEFAULT_MAX_BUFFER_SIZE)

resetScrollStep()
equivalent to setScrollStep(0.2)

resetUseArchiving()
Same as setUseArchiving(True)

resetUsePollingBuffer()
Same as setUsePollingBuffer(True)

resizeEvent(*event*)
reimplemented from TaurusPlot.resizeEvent() so that the replot timer is active only when needed

scrollstep

setEventFilters(*filters=None*, *tsetnames=None*, *preqt=False*)
propagates a list of taurus filters to the trendsets given by tsetnames. See TaurusBaseComponent.setEventFilters()

setForcedReadingPeriod(*msec=None*, *tsetnames=None*)
Sets the forced reading period for the trend sets given by tsetnames.

Parameters

- **msec** (int or None) – period in milliseconds. If None passed, the user will be prompted
- **tsetnames** (seq <str> or None) – names of the curves for which the forced reading is set. If None passed, this will be set for all present *and future* curves added to this trend

setMaxDataBufferSize(*maxSize=None*)
sets the maximum number of events that can be plotted in the trends

Parameters **maxSize** (int or None) – the maximum limit. If None is passed, the user is prompted for a value.

See also:

TaurusTrendSet.setMaxDataBufferSize()

setPaused(*paused=True*)
Pauses itself and other listeners (e.g. the trendsets) depending on it

See also:

TaurusBaseComponent.setPaused()

setScanDoor(*qdoorname*)
sets the door to which TaurusTrend will listen for scans. This removes any previous scan set using this method, but respects scans set with setModel

setScansAutoClear(*enable*)
sets whether the trend sets associated to scans should be reset every time a data_desc packet is received from the door.

Parameters **enable** (bool) –

See also:

setScanDoor() and *ScanTrendsSet*

setScansUsePointNumber(*enable*)

Note: This method is deprecated. Please use *setScansXDataKey()* instead

---

sets whether the trend sets associated to scans should use the point number from the data record for the abscissas (default).

Parameters **enable** (bool) –

setScansXDataKey(*key*, *scanname=None*)
selects the source for the data to be used as abscissas in the scan plot.

Parameters

- **key** (str) – a string corresponding to a data label for data present in the scan. Alternatively, "__SCAN_TREND_INDEX__" can be used for an internal integer count of scan records
- **scanname** (str or None) – name of the model for the scan. If None, the default scan is selected

See also:

the constructor of *ScanTrendsSet*

setScrollStep(*scrollStep*)
Sets the scroll step when in Dynamic X mode. This is used to avoid excessive replotting, which may be a problem when plotting a lot of points.

Parameters **scrollStep** (float) – portion of the current range that will be added when scrolling. For example, 0.1 means that 10% of the current range will be added when scrolling. A value of 0 means that no extra space will be added (thus the scroll is not in "steps"). Large scroll steps mean rough scrolls, but also less CPU usage.

See also:

setXDynScale()

setTrendSetsTitles(*basetitle*, *setNames=None*)
Calls setTitleText(basetitle) for each Trend Set set in setNames

Parameters

- **basetitle** (str) – the base title
- **setNames** (sequence <str> or iterator <str>) – names of the sets to be changed

See: TaurusTrendsSet.setTitleText

setUseArchiving(*enable*)
enables/disables looking up in the archiver for data stored before the Trend was started

Parameters **enable** (bool) – if True, archiving values will be used if available

setUsePollingBuffer(*enable*)
enables/disables looking up in the PollingBuffer for data

Parameters **enable** (bool) – if True, PollingBuffer values will be used if available

setXIsTime(*enable*, *axis=2*)
Reimplemented from *TaurusPlot.setXIsTime()*

showArchivingWarning()
shows a dialog warning of the potential isuues with archiving performance. It offers the user to disable archiving retrieval

showEvent(*event*)
reimplemented from TaurusPlot.showEvent() so that the replot timer is active only when needed

updateCurves(*names*)
Defines the curves that need to be plotted. For a TaurusTrend, the models can refer to:

•scalar data: they are to be plotted in a trend

•on-dimensional data: each element of the spectrum is considered independently

Note that passing an attribute for X values makes no sense in this case

Internally, every curve is grouped in a TaurusTrendSet. For each SPECTRUM attribute, a TrendSet is created, containing as many curves as the lenght of the spectrum For eacha SCALAR attribute, a TrendSet containing just one curve is created.

Parameters **names** (sequence <str>) – a sequence of model names

---

Note: Adding/removing a model will add/remove a whole set. No sub-set adding/removing is allowed. Still, each curve will be independent regarding its properties, and can be hidden/shown independently.

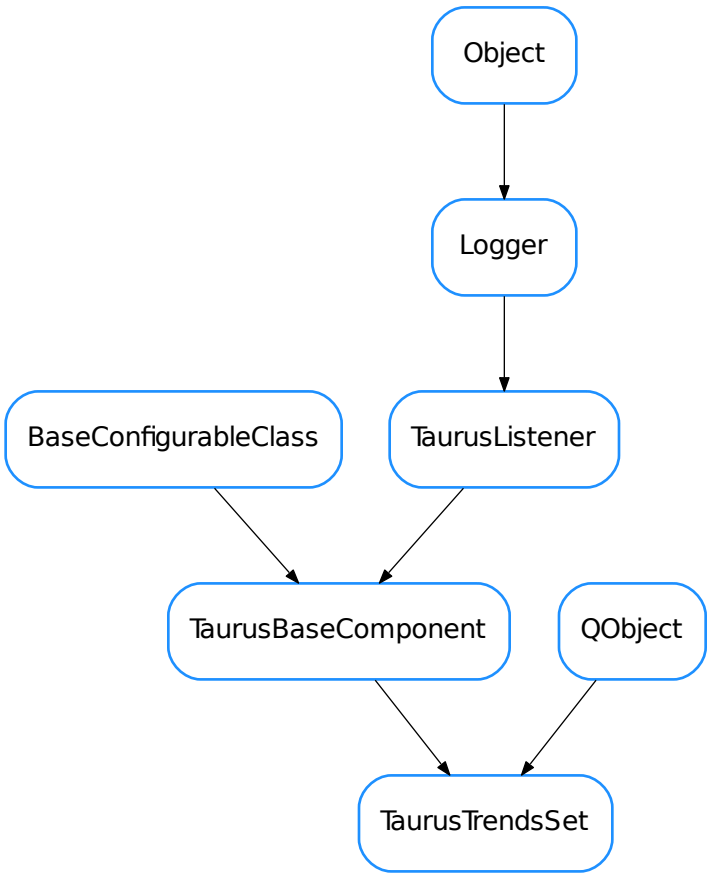
See also:

TaurusPlot.updateCurves()

useArchiving

usePollingBuffer

TaurusTrendsSet



class TaurusTrendsSet(*name*, *parent=None*, *curves=None*)

Bases: PyQt4.QtCore.QObject, taurus.qt.qtgui.base.taurusbase.TaurusBaseComponent

A collection of TaurusCurves generated from a Taurus Attribute.

If the attribute is a scalar, The Trend Set consists of only one curve representing the evolution of the value of the attribute. If the attribute is a SPECTRUM, as many curves as the length of the spectrum are created, each representing the evolution of the value of a component of the array.

When an event is received, all curves belonging to a TaurusTrendSet are updated.

TaurusTrendSet objects can be considered as containers of (sorted) curves. As such, the curves contained by them can be accessed with item notation, as in the following example:

```
attrname = 'a/b/c/d'        #consider this attribute is a SPECTRUM of 3 elements
ts=TaurusTrendSet(attrname)
...                         # wait for a Taurus Event arriving so that the curves␣
↪are created
ncurves = len(ts)           #ncurves will be 3 (assuming the event already arrived)
curve0 = ts[0]              #you can access the curve by index
curve1 = ts['a/b/c/d[1]']   #and also by name
```

Note that internally each curve is treated as a RawData curve (i.e., it is not aware of events by itself, but it relies on the TaurusTrendSet object to update its values)

addCurve(*name*, *curve*)
add a curve (with the given name) to the internal curves dictionary of this TaurusTrendSet

Parameters

- **name** (str) – the name of the curve
- **curve** (*TaurusCurve*) – the curve object to be added

clearTrends(*replot=True*)
clears all stored data (buffers and copies of the curves data)

Parameters **replot** (bool) – do a replot after clearing

compileBaseTitle(*basetitle*)
Return a base tile for a trend in whichs substitution of known placeholders has been performed.

Parameters **basetitle** (str) –

String on which the substitutions will be performed. The following placeholders are supported:

- <label> the attribute label (default)
- <model> the model name
- <attr_name> attribute name
- <attr_fullname> full attribute name (for backwards compatibility, <attr_full_name> is also accepted)
- <dev_alias> device alias
- <dev_name> device name
- <dev_fullname> full device name (for backwards compatibility, <dev_full_name> is also accepted)
- <current_title> The current title
- <[trend_index]> Same as: *"[<trend_index>]" if Ntrends>1 else ""*

Note that <trend_index> itself is not substituted!

Return type str

Returns the compiled base title.

See also:

compileTitles()

compileTitles(*basetitle*)
Return a list of titles. Each title corresponds to a trend of the trendset (ordered). Substitution of known placeholders is performed.

Parameters **basetitle** (str) –

A string to be used as base title. It may contain any of the following placeholders (which will be substituted by their corresponding value):

- <label> the attribute label (default)
- <model> the model name
- <attr_name> attribute name
- <attr_fullname> full attribute name (for backwards compatibility, <attr_full_name> is also accepted)
- <dev_alias> device alias
- <dev_name> device name
- <dev_fullname> full device name (for backwards compatibility, <dev_full_name> is also accepted)
- <current_title> The current title
- <trend_index> The index of the trend in the trendset
- <[trend_index]> Same as: *"[<trend_index>]" if Ntrends>1 else ""*

Return type string_list

Returns a list of title strings that correspond to the list of trends in the set.

See also:

compileBaseTitle()

consecutiveDroppedEventsWarning = 3

dataChanged

droppedEventsWarning = -1

forceReading(*cache=False*)
Forces a read of the attribute and generates a fake event with it. By default it ignores the cache

Parameters **cache** (bool) – set to True to do cache'd reading (by default is False)

getCurveNames()
returns a list of the names of the curves associated to this TaurusTrendSet. The curve names will always be returned in the order they were added to the set

Return type list <str>

Returns the names of the curves

getCurves()
returns an iterator of (curveName,curveObject) tuples associated to this TaurusTrendSet. The curves will always be returned in the order they were added to the set

Return type iterator <str, *TaurusCurve*>

Returns

getForcedReadingPeriod()

getModelClass()
see TaurusBaseComponent.getModelClass()

handleEvent(*evt_src*, *evt_type*, *evt_value*)
processes Change (and Periodic) Taurus Events: updates the data of all curves in the set according to the value of the attribute.

For documentation about the parameters of this method, see TaurusBaseComponent.handleEvent()

index(*curveName*)
Returns the index in the trend for the given curve name. It gives an exception if the curve is not in the set.

Parameters **curveName** (str) – the curvename to find

Return type int

Returns The index associated to the given curve in the TrendSet

isReadOnly()

maxDataBufferSize()

registerDataChanged(*listener*, *meth*)
see TaurusBaseComponent.registerDataChanged()

setForcedReadingPeriod(*msec*)
Forces periodic reading of the subscribed attribute in order to show get new points even if no events are received. It will create fake events as needed with the read value. Note that setting a period may yield unwanted results when the x axis is set to show event numbers (xIsTime==False)since there is no way of distinguishing the real from the fake events.

Parameters **msec** (int) – period in milliseconds. Use msec<0 to stop the forced periodic reading

setMaxDataBufferSize(*maxSize*)
sets the maximum number of events that are stored in the internal buffers of the trend. Note that this sets the maximum amount of memory used by the data in this trend set to:

~(1+ntrends)*2*8*maxSize bytes

(the data is stored as float64, and two copies of it are kept: one at the x and y buffers and another at the QwtPlotCurve.data)

Parameters **maxSize** (int) – the maximum limit

setTitleText(*basetitle*)
Sets the title text of the trends this trendset. The name will be constructed by appending "[%i]" to the basetitle, where %i is the index position of the trend in the trendset. As a particular case, nothing is appended if the trendset consists of only one trend.

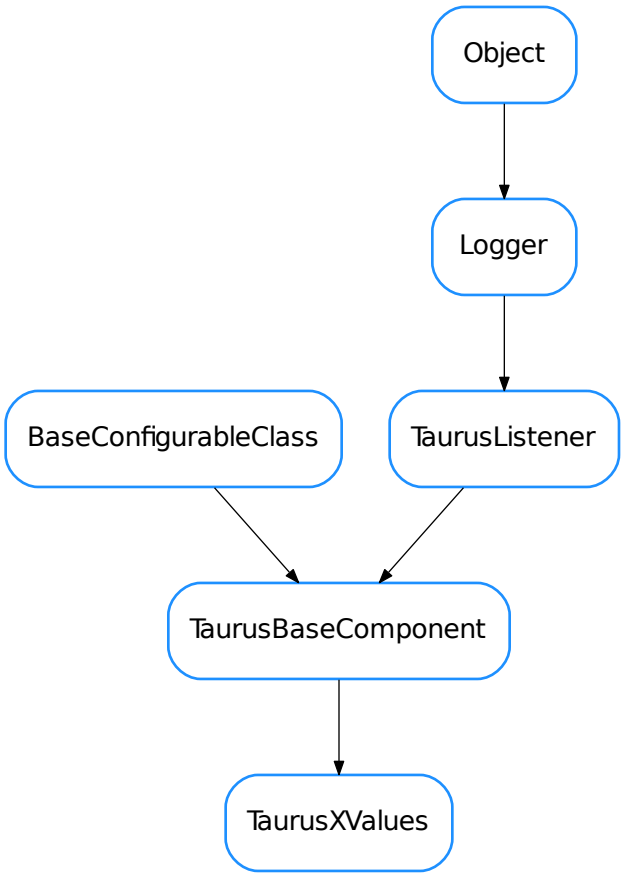
Parameters **basetitle** (str) – The title text to use as a base for constructing the title of each trend belonging to this trendset. It may contain placeholders as those used in *TaurusCurve.setTitleText()*

See also:

TaurusCurve.setTitleText()

unregisterDataChanged(*listener*, *meth*)
see TaurusBaseComponent.unregisterDataChanged()

TaurusXValues



class **TaurusXValues**(*name*, *parent=None*)

Bases: `taurus.qt.qtgui.base.taurusbase.TaurusBaseComponent`

Class for managing abscissas values in a TaurusCurve

eventHandle(*src*, *evt_type*, *val*)

see `TaurusBaseComponent.eventHandle()`

getModelClass()

see `TaurusBaseComponent.getModelClass()`

getValues()

returns the X values.

Return type `array`

Returns

isReadOnly()

see `TaurusBaseComponent.isReadOnly()`

registerDataChanged(*listener*)
see `TaurusBaseComponent.registerDataChanged()`

unregisterDataChanged(*listener*)
see `TaurusBaseComponent.unregisterDataChanged()`

- *ArrayEditor*
- *CurveAppearanceProperties*
- *CurvePropertiesView*
- *CurveStatsDialog*
- *CurvesAppearanceChooser*
- *DateTimeScaleEngine*
- *DeltaTimeScaleDraw*
- *DeltaTimeScaleEngine*
- *FancyScaleDraw*
- *FixedLabelsScaleDraw*
- *FixedLabelsScaleEngine*
- *ScanTrendsSet*
- *TaurusArrayEditor*
- *TaurusCurve*
- *TaurusCurveMarker*
- *TaurusMonitorTiny*
- *TaurusPlot*
- *TaurusPlotConfigDialog*
- *TaurusTimeScaleDraw*
- *TaurusTrend*
- *TaurusTrendsSet*
- *TaurusXValues*

Functions

isodatestr2float(*s*, *sep*='_')
converts a date string in iso format to a timestamp (seconds since epoch) with microseconds precision

taurus.qt.qtgui.style

This module provides Qt styles

Functions

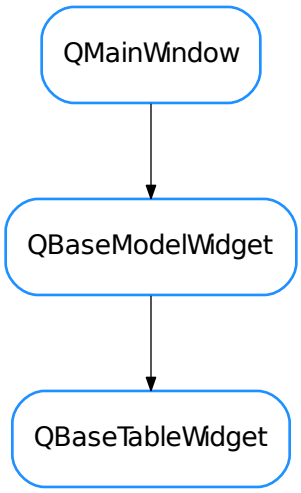
setTaurusStyle(*newStyle*)

taurus.qt.qtgui.table

This package provides taurus Qt table widgets

Classes

QBaseTableWidget



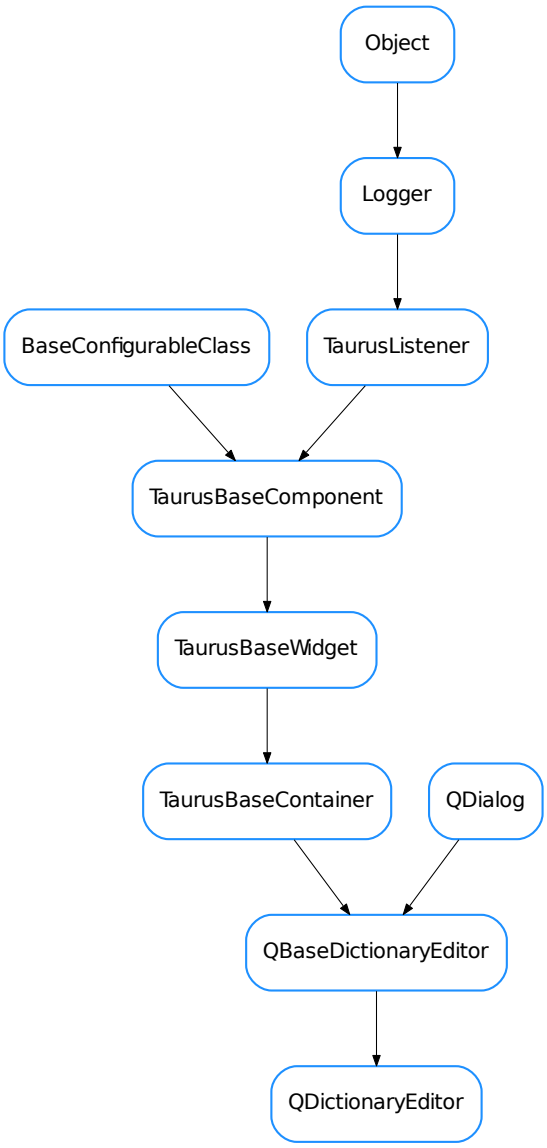
class QBaseTableWidget(*parent=None*, *designMode=False*, *with_filter_widget=True*, *with_selection_widget=True*, *with_refresh_widget=True*, *perspective=None*, *proxy=None*)

Bases: `taurus.qt.qtgui.model.qbasemodel.QBaseModelWidget`

createViewWidget(*klass=None*)

tableView()

QDictionaryEditor



class **QDictionaryEditor** (*parent=None*, *designMode=None*, *title=None*)

Bases: taurus.qt.qtgui.table.qdictionary.QBaseDictionaryEditor

getModelClass()

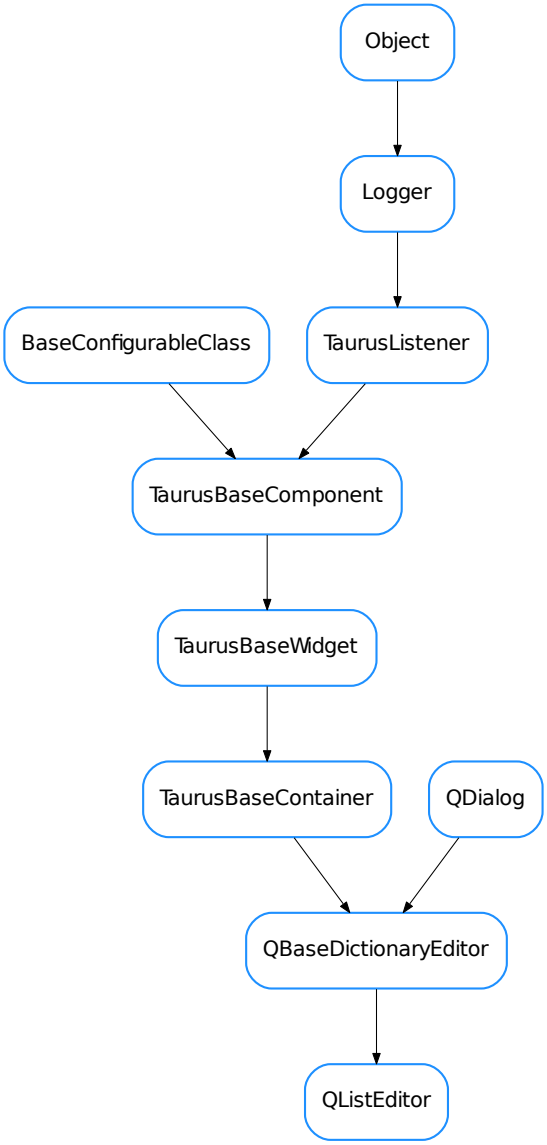
getValues()

save()

setModel(*model*)

updateStyle()

QListEditor



class QListEditor(*parent=None*, *designMode=None*, *title=None*)

Bases: taurus.qt.qtgui.table.qdictionary.QBaseDictionaryEditor

defineStyle()

getModelClass()

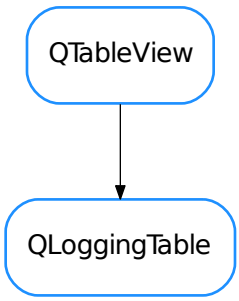
getValues()

save()

setModel(*model*)

updateStyle()

QLoggingTable



class QLoggingTable(**a*, ***kw*)

Bases: PyQt4.QtGui.QTableView

A Qt table that displays the taurus logging messages

getScrollLock()

Returns wheater or not the scrollLock is active

resetScrollLock()

rowsInserted(*index*, *start*, *end*)

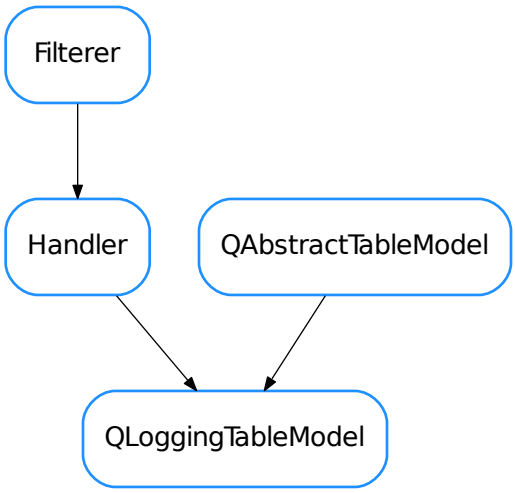
Overwrite of slot rows inserted to do proper resize and scroll to bottom if desired

scrollLock = False

setScrollLock(*scrollLock*)

Sets the state for scrollLock

QLoggingTableModel



class **QLoggingTableModel** (*capacity=500000*, *freq=0.25*)

Bases: `PyQt4.QtCore.QAbstractTableModel`, `logging.Handler`

DftColSize = (<PyQt4.QtCore.QSize object>, <PyQt4.QtCore.QSize object>, <PyQt4.QtCore.QSize object>, <PyQt4.Q

DftFont

close ()

columnCount (*index=<PyQt4.QtCore.QModelIndex object>*)

data (*index*, *role=0*)

emit (*record*)

flush ()

getRecord (*index*)

headerData (*section*, *orientation*, *role=0*)

insertRows (*position*, *rows=1*, *index=<PyQt4.QtCore.QModelIndex object>*)

removeRows (*position*, *rows=1*, *index=<PyQt4.QtCore.QModelIndex object>*)

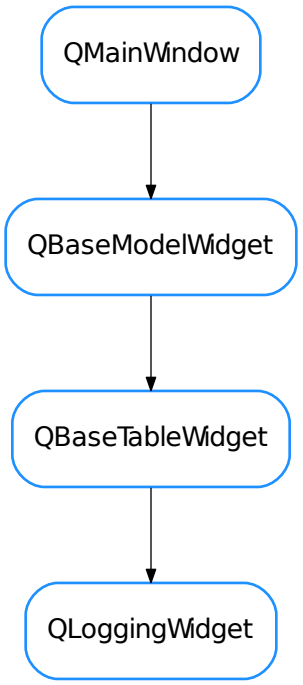
rowCount (*index=<PyQt4.QtCore.QModelIndex object>*)

sort (*column*, *order=0*)

timerEvent (*evt*)

updatePendingRecords ()

QLoggingWidget



class **QLoggingWidget**(*parent=None*, *designMode=False*, *with_filter_widget=<class 'taurus.qt.qtgui.table.qlogtable.LoggingToolBar'>*, *with_selection_widget=True*, *with_refresh_widget=True*, *perspective=None*, *proxy=None*)

Bases: `taurus.qt.qtgui.table.qtable.QBaseTableWidget`

DftPerspective = 'Standard'

KnownPerspectives = {'Remote': {'model': [<class 'taurus.qt.qtgui.table.qlogtable.QLoggingFilterProxyModel'>, <cl

createToolArea()

createViewWidget(*klass=None*)

destroy(*destroyWindow=True*, *destroySubWindows=True*)

classmethod **getQtDesignerPluginInfo**()

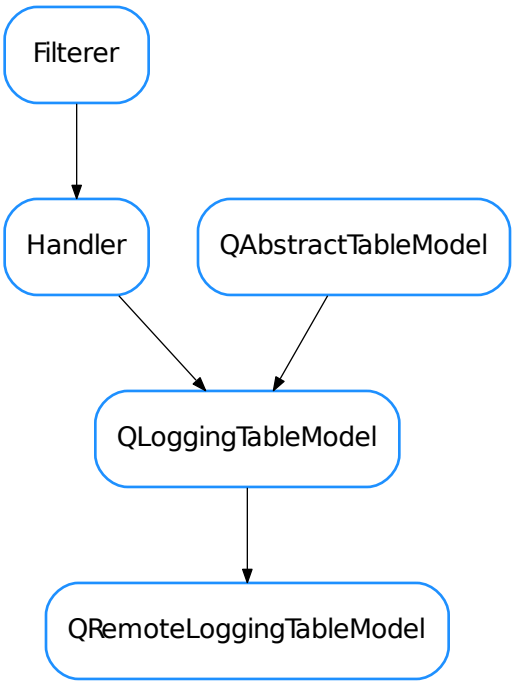
onFilterChanged(*filter*)

onScrollLockToggled(*yesno*)

onSwitchPerspective(*perspective*)

stop_logging()

QRemoteLoggingTableModel



class **QRemoteLoggingTableModel**(*capacity=500000*, *freq=0.25*)

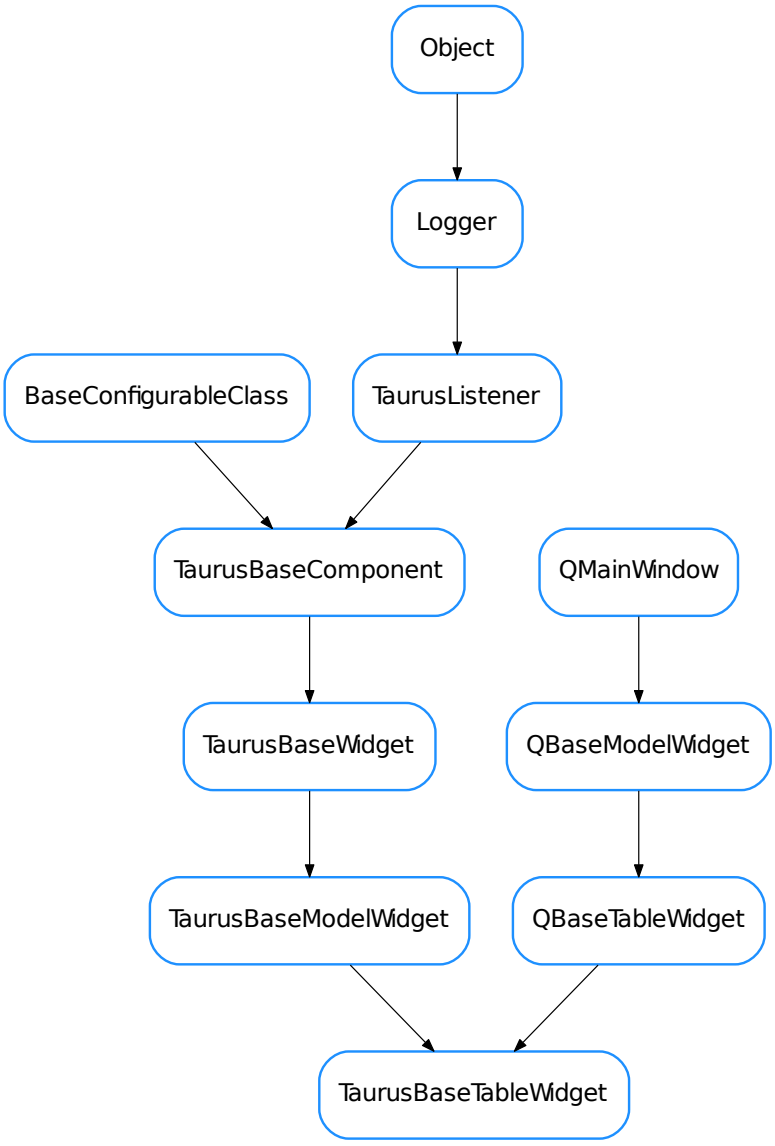
Bases: `taurus.qt.qtgui.table.qlogtable.QLoggingTableModel`

A remote Qt table that displays the taurus logging messages

connect_logging(*host='localhost'*, *port=9020*, *handler=<class taurus.qt.qtgui.table.qlogtable._LogRecordStreamHandler>*)

disconnect_logging()

TaurusBaseTableWidget



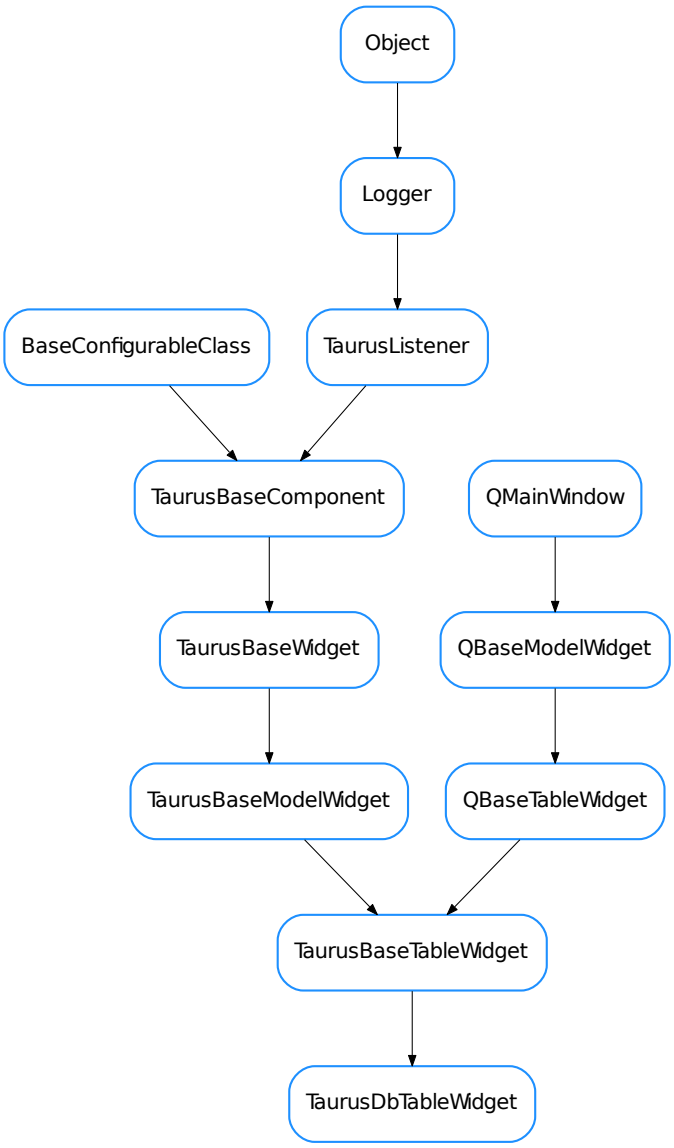
class **TaurusBaseTableWidget**(*parent=None*, *designMode=False*, *with_filter_widget=True*, *perspective=None*, *proxy=None*)

Bases: `taurus.qt.qtgui.table.qtable.QBaseTableWidget`, `taurus.qt.qtgui.model.qbasemodel.TaurusBaseModelWidget`

A class:*taurus.qt.qtgui.tree.QBaseTableWidget* that connects to a taurus model.

Filters can be inserted into this widget to restrict the items that are seen.

TaurusDbTableWidget



class TaurusDbTableWidget(*parent=None*, *designMode=False*, *with_filter_widget=True*, *perspective=None*, *proxy=None*)

Bases: `taurus.qt.qtgui.table.taurustable.TaurusBaseTableWidget`

A class:*taurus.qt.qtgui.tree.TaurusBaseTableWidget* that connects to a `taurus.core.taurusauthority.TaurusAuthority` model. It can show the list of database elements in two different perspectives:

- device : a device list based perspective

- server : a server list based perspective

Filters can be inserted into this widget to restrict the items that are seen.

DftPerspective = 3

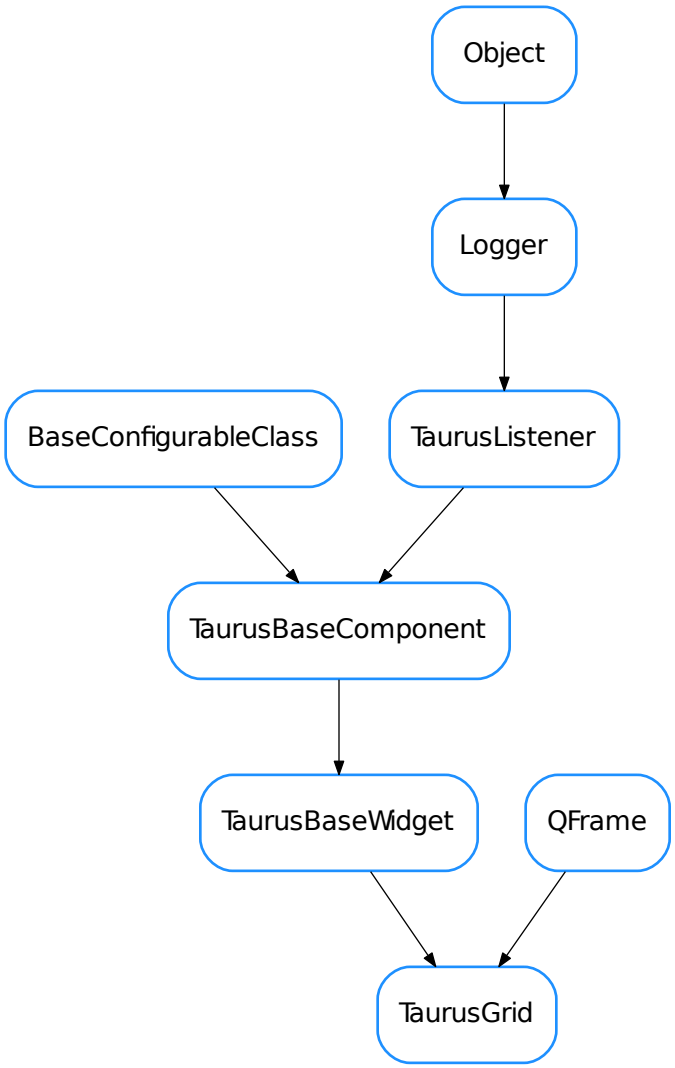
KnownPerspectives = {8: {'model': [<class 'taurus.qt.qtcore.model.taurusdatabasemodel.TaurusDbServerProxyMod

getModelClass()

classmethod getQtDesignerPluginInfo()

sizeHint()

TaurusGrid



class **TaurusGrid**(*parent=None*, *designMode=False*)

Bases: `PyQt4.QtGui.QFrame`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget`

TaurusGrid is a Taurus widget designed to represent a set of attributes distributed in columns and rows. The Model will be a list with attributes or device names (for devices the State attribute will be shown). Each setModel(*) execution will be able to modify the attribute list. An example of execution:<pre> /usr/bin/python taurusgrid.py "model=lt.*/VC.*/.*/((C*)|(P*)|(I*))" cols=IP,CCG,PNV rows=LT01,LT02 </pre> @author originally developed by gcuni, extended by srubio and sblanch @todo Future releases should allow a list of filters as argument @todo names/widgets should be accessible as a caselessdict dictionary (e.g. for adding custom context menus) @todo refactoring to have methods that add/remove new widgets one by one, not only the whole dictionary @todo _TAGS property should allow to change row/columns meaning and also add new Custom tags based on regexp

attach()
Attaches the widget to the model

build_table(*values*)
This is a builder. For all the elements in widgets matrix, just set the corresponding cells of the QTableWidget.

build_widgets(*values*, *show_labels=False*, *width=240*, *height=20*, *value_width=120*)

columnlabels

create_frame_with_gridlayout()
Just a 'macro' to create the layouts that seem to fit better.

create_widgets_dict(*models*)

create_widgets_table(*models*)

defineStyle()
Defines the initial style for the widget

detach()
Detaches the widget from the model

getColumnLabels()

getItemByModel(*model*, *index=0*)

getModel()

getModelClass()

classmethod **getQtDesignerPluginInfo**()

getRowLabels()

itemClicked(*item_name*)

itemSelected

load(*filename*, *delayed=False*)

minimumSizeHint()

model

modelsThread

parse_labels(*text*)

resetColumnLabels()

resetModel()

resetRowLabels()

rowlabels

save(*filename*)

setColumnLabels(*columns*)
The model can be initialized as a list of devices or hosts or ...

setItemSelected(*item_name=''*, *selected=True*)
it adds a blue frame around a clicked item.

setModel(*model*, *devsInRows=False*, *delayed=False*, *append=False*, *load=True*)
The model can be initialized as a list of devices or hosts or dictionary or ...

setRowLabels(*rows*)
The model can be initialized as a list of devices or hosts or ...

setTitle(*title*)

showAttributeLabels(*boolean*)

showAttributeUnits(*boolean*)

showColumnFrame(*boolean*)

showOthers(*boolean*)

showRowFrame(*boolean*)

show_hide_columns()
This needs refactoring to be together with the show_hide_rows method

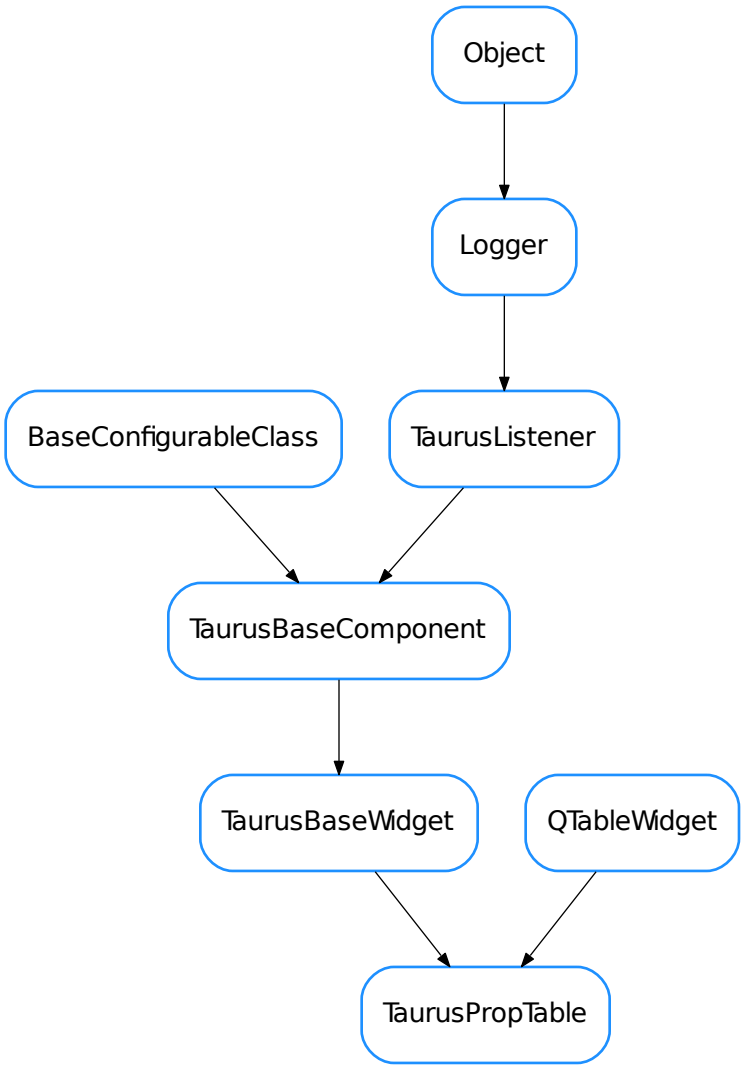
show_hide_rows()
This needs refactoring to be together with the show_hide_columns method

sizeHint()

updateStyle()

useParentModel

TaurusPropTable



class TaurusPropTable(*parent=None*, *designMode=False*)

Bases: `PyQt4.QtGui.QTableWidget`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget`

This widget will show a list of properties of device and the list of values. @todo add a frame for Add, Delete and Refresh buttons!

addProperty()

contextMenuEvent(*event*)

This function is called when right clicking on qwt plot area. A pop up menu will be shown with the available options.

defineStyle()
Defines the initial style for the widget

deleteProperty()

editProperty()

getModelClass()

classmethod getQtDesignerPluginInfo()

get_device_property_names(*dev_name*, *wildcard='*'*)

minimumSizeHint()

model

put_device_property(*dev_name*, *dict*)

setModel(*model*)

setNewPropertyValue(*new_text*)

setPropertyValue(*value*, *i*, *j*)
This method inserts a new table widget inside the cell @deprecated ... use setText() and editProperty() event call instead!!!

setTable

setText(*value*, *i=None*, *j=None*)

sizeHint()

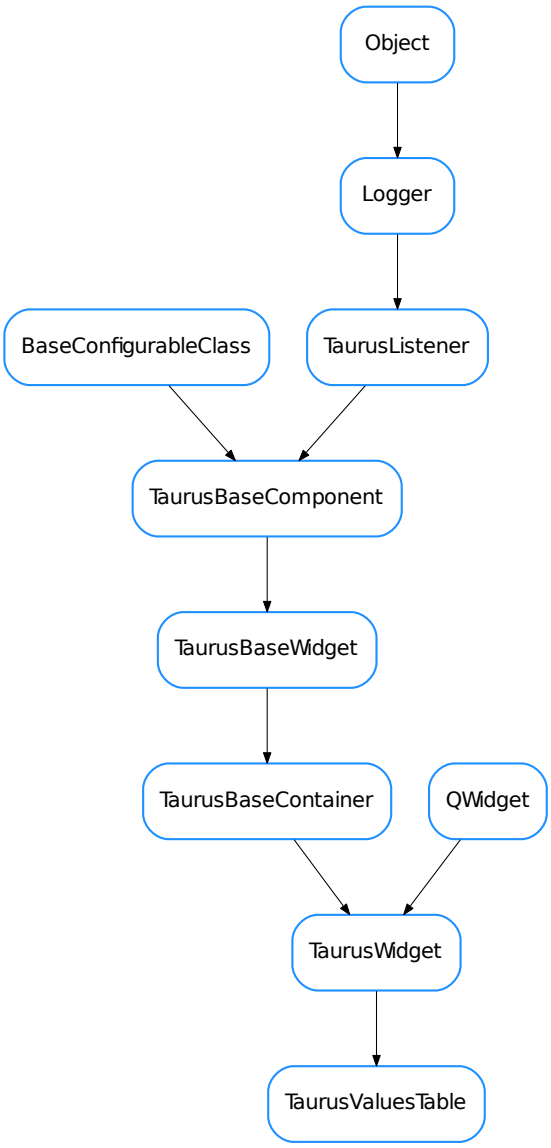
updateStyle()

useParentModel

valueChanged()
@deprecated valueChanged THIS DOES NOTHING!

valueDoubleClicked(*x*, *y*)

TaurusValuesTable



class **TaurusValuesTable** (*parent=None*, *designMode=False*, *defaultWriteMode=None*)

Bases: `taurus.qt.qtgui.container.tauruswidget.TaurusWidget`

A table for displaying and/or editing 1D/2D Taurus attributes

applyChanges()

Writes table modifications to the device server.

askCancel()

Shows a QMessageBox, asking if user wants to cancel all changes. Triggered when user clicks Cancel

button.

cancelClicked()
This is a SLOT that is being triggered when CANCEL button is clicked.

Note: This SLOT is called, when user does not want to apply table modifications. When no cell was modified it will not be called.

---

chooseModel()
shows a model chooser

contextMenuEvent(*event*)
Reimplemented from `QWidget.contextMenuEvent()`

getModelClass()
see `TaurusWidget.getModelClass()`

classmethod getQtDesignerPluginInfo()
Reimplemented from `TaurusWidget.getQtDesignerPluginInfo()`

getWriteMode()
whether the widget is showing the read or write values

Return type `bool`

Returns

handleEvent(*evt_src*, *evt_type*, *evt_value*)
see `TaurusWidget.handleEvent()`

isReadOnly()
Reimplemented from `TaurusWidget.isReadOnly()`

model

okClicked()
This is a SLOT that is being triggered when ACCEPT button is clicked.

Note: This SLOT is called, when user wants to apply table modifications. When no cell was modified it will not be called. When modifications have been done, they will be writen to w_value of an attribute.

---

resetWriteMode()
equivalent to self.setWriteMode(self.defaultWriteMode)

setModel(*model*)
Reimplemented from `TaurusWidget.setModel()`

setModifiableByUser(*modifiable*)
Reimplemented from `TaurusWidget.setModifiableByUser()`

setWriteMode(*isWrite*)
Triggered when the read mode is changed to write mode.

Parameters **isWrite** (`bool`) –

writeMode

- *QBaseTableWidget*
- *QDictionaryEditor*
- *QListEditor*

- *QLoggingTable*
- *QLoggingTableModel*
- *QLoggingWidget*
- *QRemoteLoggingTableModel*
- *TaurusBaseTableWidget*
- *TaurusDbTableWidget*
- *TaurusGrid*
- *TaurusPropTable*
- *TaurusValuesTable*

taurus.qt.qtgui.taurusgui

This package provides TaurusGui, a generic framework for creating GUIs without actual coding (just configuration files).

See the examples provided in the conf subdirectory directory as well as the documentation of the *TaurusGui* class.

The "new GUI wizard" and XML configuration files

Note that the configuration files can either be written by hand or by launching the "new GUI" wizard with *taurusgui –new-gui*, which will create a new directory containing configuration, resource and launcher files.

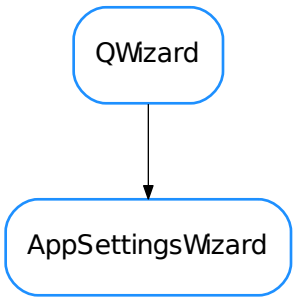
The new GUI wizard stores all the options in xml format in a file called *config.xml* and creates a simple *config.py* file containing the following line:

```
XML_CONFIG = 'config.xml'
```

This line indicates that *config.xml* should also be used as a source of configuration options (in case of conflict, the options set in *config.py* prevail).

Classes

AppSettingsWizard



class AppSettingsWizard(*parent=None*, *jdrawCommand='jdraw'*, *configFilePrefix='config'*)
Bases: PyQt4.QtGui.QWizard

This Wizard provide functionality for creating from scratch a configuration directory for a TaurusGUI based application.

The files in the configuration dir determine the default, permanent, pre-defined contents of the GUI. While the user may add/remove more elements at run time and those customizations will also be stored, this file defines what a user will find when launching the GUI for the first time.

Pages = Enumeration('Pages', ['IntroPage', 'ProjectPage', 'GeneralSettings', 'CustomLogoPage', 'SynopticPage', 'Mac

SARDANA_INSTALLED = False

addPage(*page*)

generateXml()
returns the xml code corresponding to the options selected in the wizard and a dictionary representing the paths that have been substituted.

Return type str, dict <str, str>

Returns The return value is a tuple whose first element is the xml code and the second element is a dict where the keys are the destination files and the values are the original paths.

static getArrayFromNode(*rootNode*, *nodeName*, *default=None*)
returns an array contained by given Node :type rootNode: Element :param rootNode: root node :param nodeName: the name of node to find :param default: returned value if node is None or contains empty string

getConfigFilePrefix()

getPages()

getProjectWarnings()

static getValueFromNode(*rootNode*, *nodeName*, *default=None*)
returns a value from given Node :type rootNode: Element :param rootNode: root node :param nodeName: the name of node to find :param default: returned value if node is None or contains empty string

getXml()

getXmlConfigFileName()

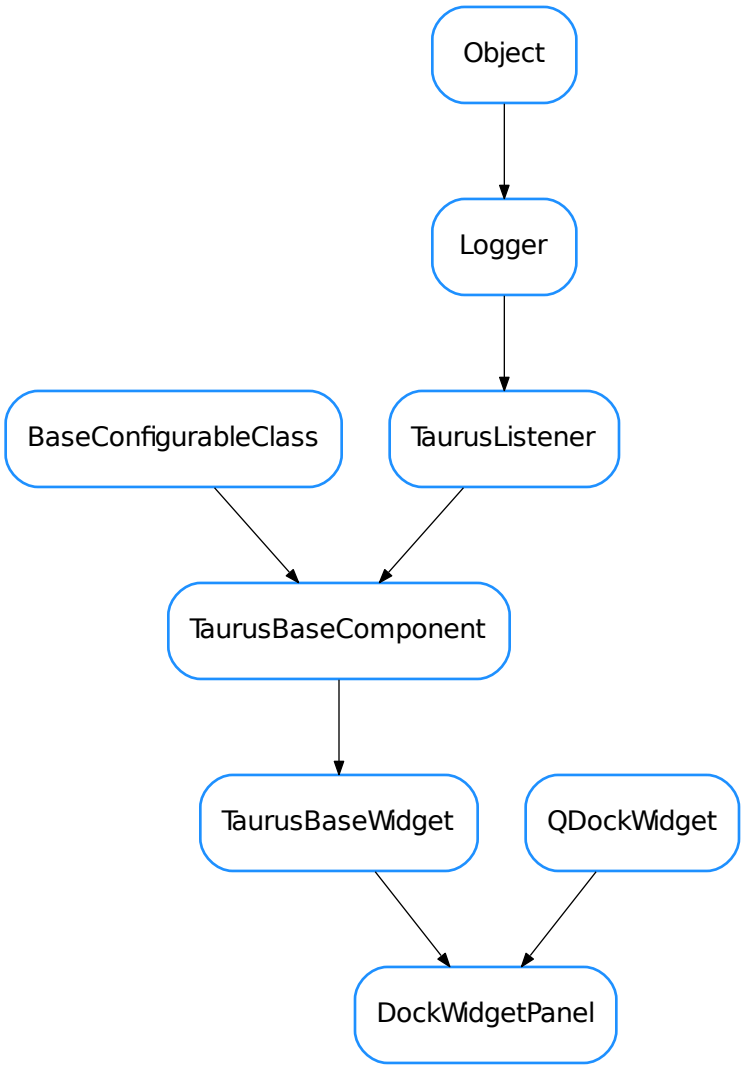
loadXml(*fname*)
parses xml code and sets all pages according to its contents. It raises an exception if something could not be processed

Parameters **fname** (unicode) – path to file containing xml code

setPage(*id*, *page*)

substitutionName(*src*, *mod_dir*)

DockWidgetPanel



class DockWidgetPanel(*parent*, *widget*, *name*, *mainwindow*)

Bases: `PyQt4.QtGui.QDockWidget`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget`

This is an extended QDockWidget which provides some methods for being used as a "panel" of a TaurusGui application. Widgets of TaurusGui are inserted in the application by adding them to a DockWidgetPanel.

applyConfig(*configdict*, *depth=-1*)

closeEvent(*event*)

createConfig(**args*, ***kwargs*)

getWidgetClassName()

getWidgetModuleName()

isCustom()

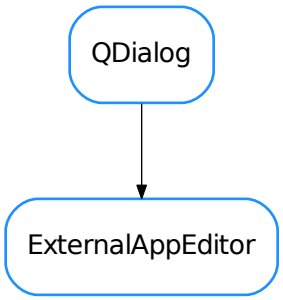
isPermanent()

setCustom(*custom*)

setPermanent(*permanent*)

setWidgetFromClassName(*classname*, *modulename=None*)

ExternalAppEditor



class ExternalAppEditor(*parent=None*)

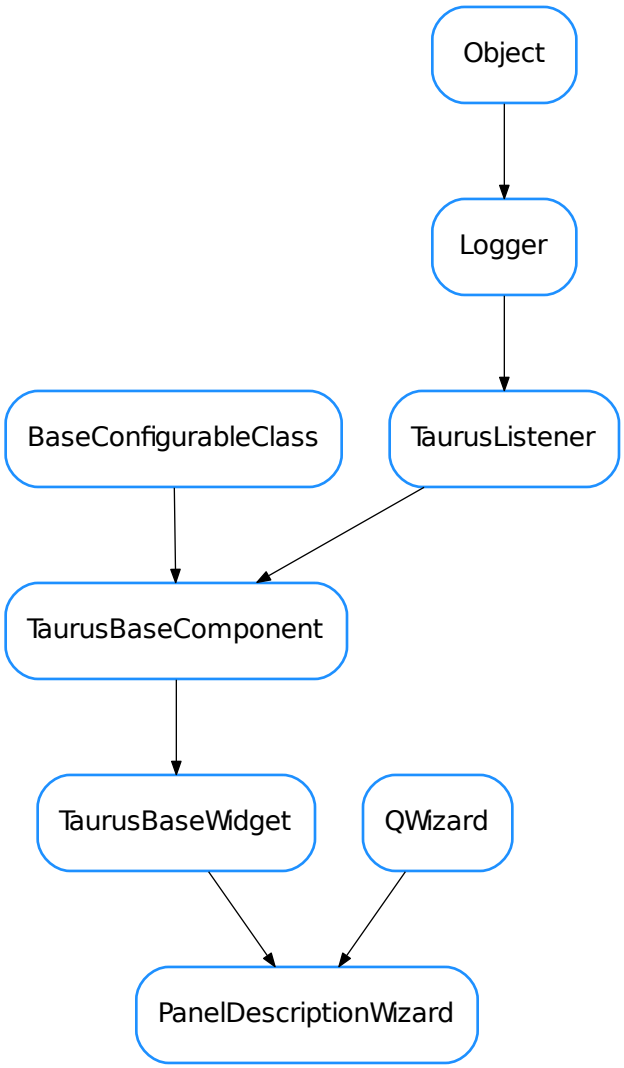
Bases: `PyQt4.QtGui.QDialog`

A dialog for configuring an external appaction for a TaurusMainWindow.

checkData()

static getDialog()

PanelDescriptionWizard



class **PanelDescriptionWizard**(*parent=None*, *designMode=False*, *gui=None*, *extraWidgets=None*)

Bases: `PyQt4.QtGui.QWizard`, `taurus.qt.qtgui.base.taurusbase.TaurusBaseWidget`

A wizard-style dialog for configuring a new TaurusGui panel. Use *`getDialog()`* for launching it

static **getDialog**(*parent*, *extraWidgets=None*)

Static method for launching a new Dialog.

Parameters **parent** – parent widget for the new dialog

Return type `tuple` <`PanelDescription`, `bool`>

Returns tuple of a description object and a state flag. The state is True if the dialog was accepted

and False otherwise

getGui()
returns a reference to the GUI to which the dialog is associated

getPanelDescription()
Returns the panel description with the choices made so far

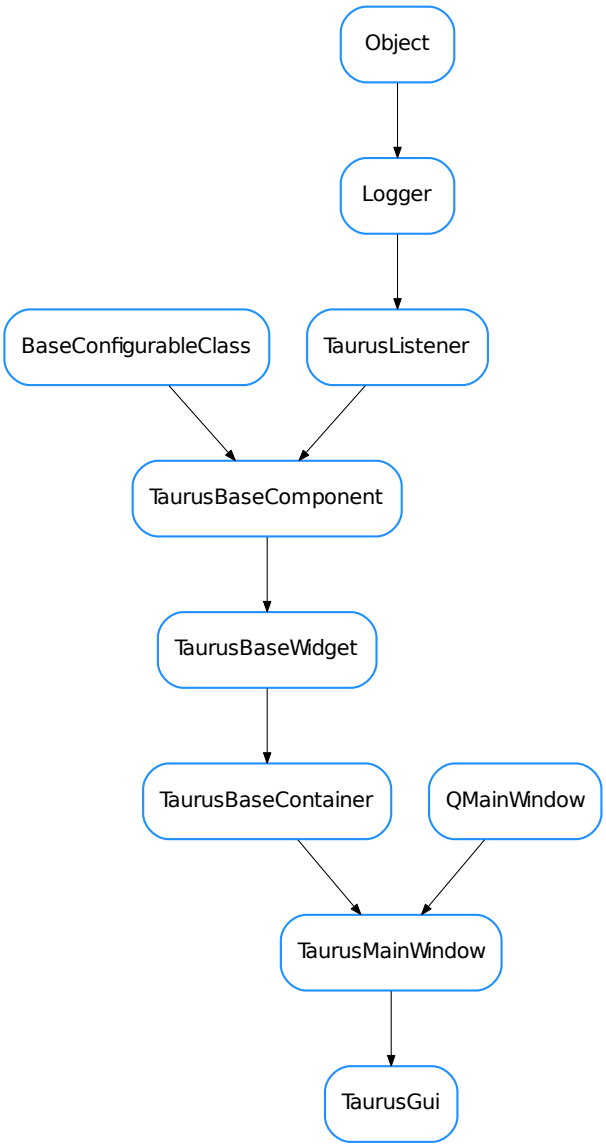
Return type PanelDescription

Returns the panel description

setPanelDescription(*desc*)
Sets the Panel description

Parameters **desc**(PanelDescription) –

TaurusGui



class TaurusGui (*parent=None*, *confname=None*, *configRecursionDepth=None*)

Bases: `taurus.qt.qtgui.container.taurusmainwindow.TaurusMainWindow`

This is main class for constructing the dynamic GUIs. TaurusGui is a specialised TaurusMainWindow which is able to handle "panels" and load configuration files. There are several ways of using TaurusGui. In the following we will give 3 examples on how to create a simple GUI called "MyGui" which contains one panel called "Foo" and consisting of a *QWidget*:

Example 1: use declarative configuration files.

You can create a purely declarative configuration file to be interpreted by the standard *taurusgui* script:

```
from taurus.qt.qtgui.taurusgui.utils import PanelDescription

GUI_NAME = 'MyGui'
panel = PanelDescription('Foo',
                         classname='taurus.external.qt.Qt.QWidget')
```

Note that this just a very simple example. For a much richer one, see the `taurus.qt.qtgui.taurusgui.conf.tgconf_example01`

Example 2: do everything programmatically.

A stand-alone python script that launches the gui when executed. No configuration file is used here. Panels and other components are added programatically:

```
if __name__ == '__main__':
    from taurus.qt.qtgui.application import TaurusApplication
    from taurus.qt.qtgui.taurusgui import TaurusGui
    from taurus.external.qt import Qt
    app = TaurusApplication(app_name='MyGui')
    gui = TaurusGui()
    panel = Qt.QWidget()
    gui.createPanel(panel, 'Foo')
    gui.show()
    app.exec_()
```

Example 3: mixing declarative and programmatic ways

It is also possible to create a stand-alone python script which loads itself as a configuration file. In this way you can add things programmatically and at the same time use the declarative way:

```
GUI_NAME = 'MyGui' # <-- declarative!
if __name__ == '__main__':
    from taurus.qt.qtgui.application import TaurusApplication
    from taurus.qt.qtgui.taurusgui import TaurusGui
    from taurus.external.qt import Qt
    app = TaurusApplication()
    gui = TaurusGui(confname=__file__)
    panel = Qt.QWidget()
    gui.createPanel(panel, 'Foo')  # <-- programmatic!
    gui.show()
    app.exec_()
```

IMPLICIT_ASSOCIATION = '__[IMPLICIT]__'

SelectedInstrument

closeEvent(*event*)

createConfig(**args*, ***kwargs*)
reimplemented from TaurusMainWindow.createConfig

createConsole(*kernels*)

createCustomPanel(*paneldesc=None*)
Creates a panel from a Panel Description and sets it as "custom panel".

Parameters **paneldesc** (`PanelDescription`) – description of the panel to be created

See also:

createPanel()

createExternalApp()
Add a new external application on execution time

createInstrumentsFromPool(*macroservername*)
Creates a list of instrument panel descriptions by gathering the info from the Pool. Each panel is a TaurusForm grouping together all those elements that belong to the same instrument according to the Pool info

Return type list <PanelDescription>

Returns

createMainSynoptic(*synopticname*)
Creates a synoptic panel and registers it as "SelectedInstrument" reader and writer (allowing selecting instruments from synoptic

createPanel(*widget*, *name*, *floating=False*, *registerconfig=True*, *custom=False*, *permanent=False*, *icon=None*, *instrumentkey=None*)
Creates a panel containing the given widget.

Parameters

- **wiget** (QWidget) – the widget to be contained in the panel
- **name** (str) – the name of the panel. It will be used in tabs as well as for configuration
- **floating** (bool) – whether the panel should be docked or floating. (see note below)
- **registerconfig** (bool) – if True, the panel will be registered as a delegate for configuration
- **custom** (bool) – if True the panel is to be considered a "custom panel"
- **permanent** (bool) – set this to True for panels that need to be recreated when restoring the app
- **icon** (QIcon) – icon for the panel
- **instrumentkey** (str) – name of an instrument to which this panel is to be associated

Return type *DockWidgetPanel*

Returns the created panel

Note: On a previous version, there was a mandatory parameter called *area* (which accepted a Qt.DockWidgetArea or None as values) this parameter has now been substituted by the keyword argument *floating*. In order to provide backwards compatibility, the "floating" keyword argument stays at the same position as the old *area* argument and if a Qt.DockWidgetArea value is given, it will be interpreted as floating=True (while if *None* is passed, it will be interpreted as floating=False.

---

doorNameChanged

findPanelsInArea(*area*)
returns all panels in the given area

Parameters

- **area** (DockWidgetArea or str) – . If area=='FLOATING', the dockwidgets that are floating will be returned.
- **area** –

> **Warning:** This method is deprecated

getAllInstrumentAssociations()
Returns the dictionary of instrument-panel associations

Return type dict <str, str>

Returns a dict whose keys are the instruments known to the gui and whose values are the corresponding associated panels (or None).

getCustomWidgetMap()
Returns the default map used to create custom widgets by the TaurusForms belonging to this GUI

Return type dict <str, QWidget>

Returns a dictionary whose keys are device type strings (i.e. see PyTango.DeviceInfo) and whose values are widgets to be used

See also:

setCustomWidgetMap()

getInstrumentAssociation(*instrumentname*)
Returns the panel name associated to an instrument name

Parameters **instrumentname** (str or None) – The name of the instrument whose associated panel is wanted

Return type str or None

Returns the associated panel name (or None).

getPanel(*name*)
get a panel object by name

Return type *DockWidgetPanel*

Returns

getPanelNames()
returns the names of existing panels

Return type list <str>

Returns

classmethod getQtDesignerPluginInfo()
TaurusGui is not to be in designer

hideAllPanels()
hides all current panels

loadConfiguration(*confname*)
Reads a configuration file

Parameters **confname** (str or None) – the name of module located in the PYTHONPATH or in the conf subdirectory of the directory in which taurusgui.py file is installed. This method will try to import <confname>. If that fails, it will try to import *tgconf_*<*confname*>. Alternatively, *confname* can be the path to the configuration module (not necessarily in the PYTHONPATH). *confname* can also be None, in which case a dummy empty module will be used.

macroserverNameChanged

newShortMessage

onExportCurrentPanelConfiguration(*fname=None*)

onSelectedInstrument(*instrumentname*)
Slot to be called when the selected instrument has changed (e.g. by user clicking in the synoptic)

Parameters **instrumentname** (str) – The name that identifies the instrument.

onShortMessage(*msg*)
Slot to be called when there is a new short message. Currently, the only action taken when there is a new message is to display it in the main window status bar.

Parameters **msg** (str) – the short descriptive message to be handled

onShowAssociationDialog()
launches the instrument-panel association dialog (modal)

onShowManual(*anchor=None*)
reimplemented from TaurusMainWindow to show the manual in a panel (not just a dockwidget)

removeExternalApp(*name=None*)
Remove the given external application from the GUI.

Parameters **name** (str or None) – the name of the external application to be removed If None given, the user will be prompted

removePanel(*name=None*)
remove the given panel from the GUI.

Note: The panel; is actually removed from the current perspective. If the panel is saved in other perspectives, it should be removed from them as well.

Parameters **name** (str or None) – the name of the panel to be removed If None given, the user will be prompted

setAllInstrumentAssociations(*associationsdict*, *clearExisting=False*)
Sets the dictionary of instrument-panel associations. By default, it keeps any existing association not present in the associationsdict.

Parameters

- **associationsdict** (dict <str, str>) – a dict whose keys are the instruments names and whose values are the corresponding associated panels (or None)
- **clearExisting** (bool) – if True, the the existing asociations are cleared. If False (default) existing associations are updated with those in associationsdict

setCustomWidgetMap(*map*)
Sets the widget map that is used application-wide. This widget map will be used by default in all TaurusForm Panels belonging to this gui.

Parameters **map** (dict <str, QWidget>) – a dictionary whose keys are device type strings (e.g. see PyTango.DeviceInfo) and whose values are widgets to be used

See also:

TaurusForm.setCustomWidgetMap(), *getCustomWidgetMap()*

setFocusToPanel(*panelname*)
Method that sets a focus for panel passed via an argument

Parameters **panelname** (str) – The name that identifies the panel. This name must be unique within the panels in the GUI.

setInstrumentAssociation(*instrumentname*, *panelname*)
Sets the panel name associated to an instrument

Parameters

- **instrumentname** (str) – The name of the instrument
- **panelname** (str or None) – The name of the associated panel or None to remove the association for this instrument.

setLockView(*locked*)

setModifiableByUser(*modifiable*)

showAllPanels()
shows all current panels

showSDMInfo()
pops up a dialog showing the current information from the Shared Data Manager

tabifyArea(*area*)
tabifies all panels in a given area.

Parameters **area** (DockWidgetArea) –

> **Warning:** This method is deprecated

updatePermanentCustomPanels(*showAlways=True*)
Shows a dialog for selecting which custom panels should be permanently stored in the configuration.

Parameters **showAlways** (bool) – forces showing the dialog even if there are no new custom Panels

updatePermanentExternalApplications(*showAlways=True*)
Shows a dialog for selecting which new externals applications should be permanently stored in the configuration.

Parameters **showAlways** (bool) – forces showing the dialog

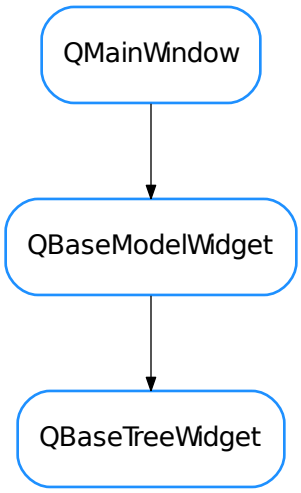
- *AppSettingsWizard*
- *DockWidgetPanel*
- *ExternalAppEditor*
- *PanelDescriptionWizard*
- *TaurusGui*

taurus.qt.qtgui.tree

This package provides taurus Qt tree widgets

Classes

QBaseTreeWidget



class QBaseTreeWidget(*parent=None*, *designMode=False*, *with_navigation_bar=True*, *with_filter_widget=True*, *with_selection_widget=True*, *with_refresh_widget=True*, *perspective=None*, *proxy=None*)

Bases: `taurus.qt.qtgui.model.qbasemodel.QBaseModelWidget`

A pure Qt tree widget implementing a tree with a navigation toolbar

collapseAllTree()

collapseSelectionTree()

createToolArea()

createViewWidget(*klass=None*)

expandAllTree()

expandSelectionTree()

goIntoAction()

goIntoTree()

goTopAction()

goTopTree()

goUpAction()

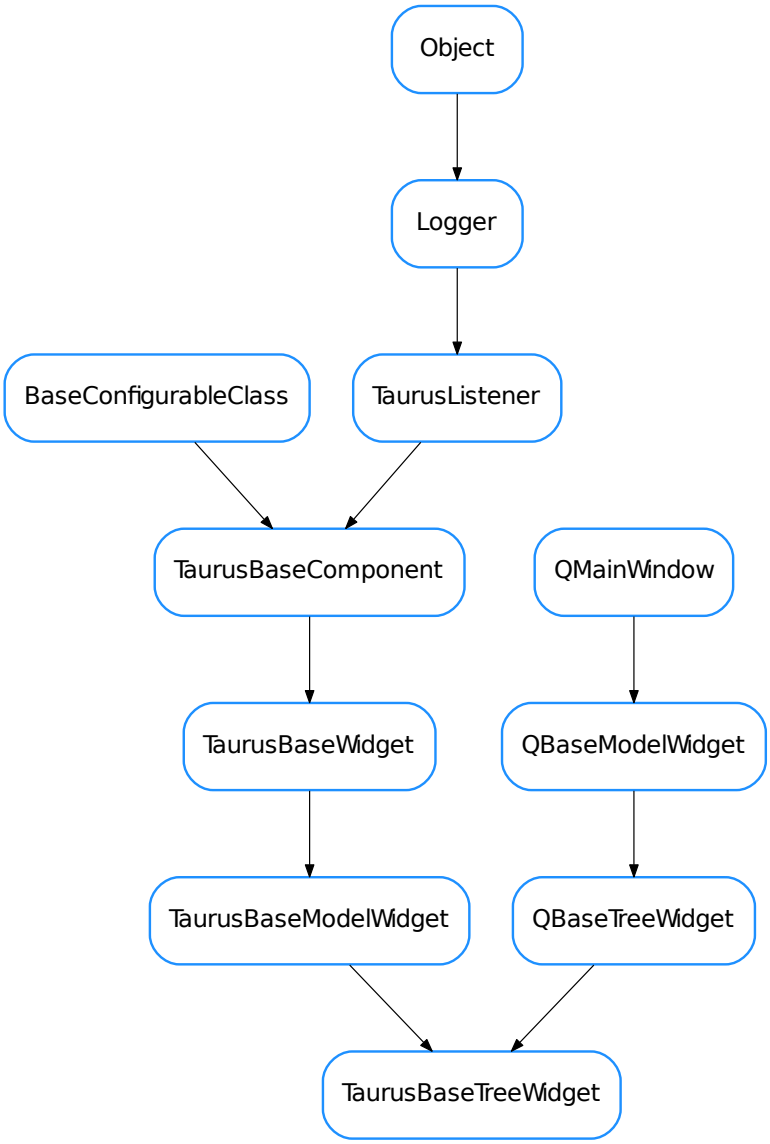
goUpTree()

onExpanded()

resizeColumns()

treeView()

TaurusBaseTreeWidget



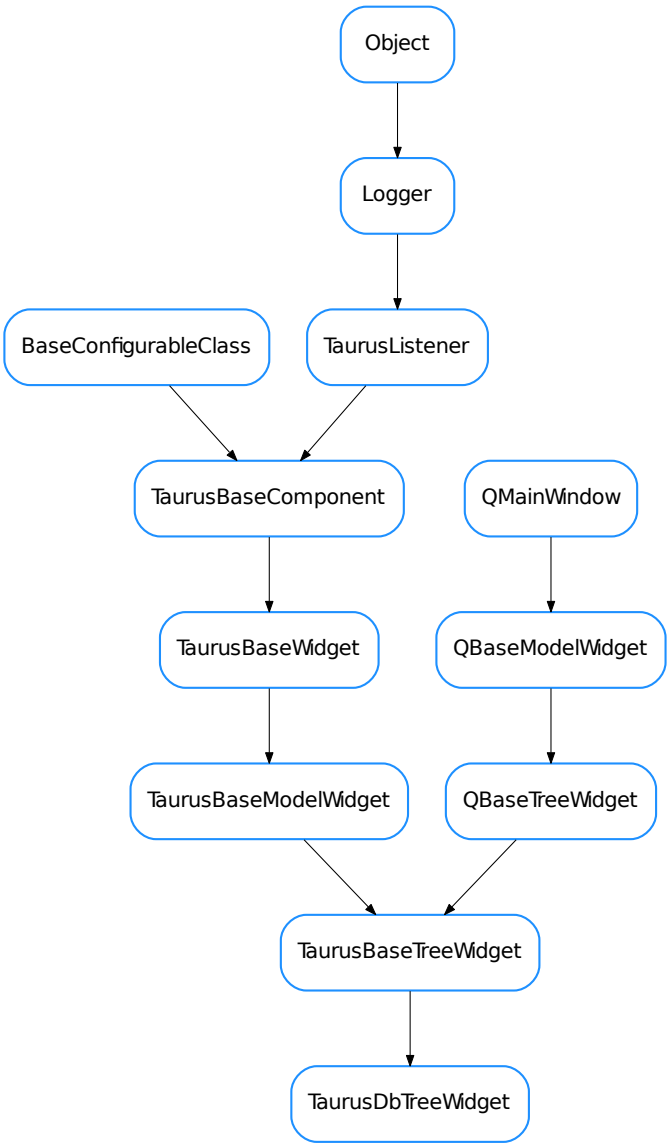
class TaurusBaseTreeWidget(*parent=None*, *designMode=False*, *with_navigation_bar=True*, *with_filter_widget=True*, *perspective=None*, *proxy=None*)

Bases: `taurus.qt.qtgui.tree.qtree.QBaseTreeWidget`, `taurus.qt.qtgui.model.qbasemodel.TaurusBaseModelWidget`

updateStyle()

overwritten from class:*taurus.qt.qtgui.base.TaurusBaseWidget*. It is called when the taurus model changes.

TaurusDbTreeWidget



class TaurusDbTreeWidget (*parent=None*, *designMode=False*, *with_navigation_bar=True*, *with_filter_widget=True*, *perspective=None*, *proxy=None*)

Bases: `taurus.qt.qtgui.tree.taurustree.TaurusBaseTreeWidget`

A class:*taurus.qt.qtgui.tree.TaurusBaseTreeWidget* that connects to a `taurus.core.taurusauthority.TaurusAuthority` model. It can show the list of database elements in four different perspectives:

- device : a three level hierarchy of devices (domain/family/name)
- server : a server based perspective

•class : a class based perspective

Filters can be inserted into this widget to restrict the tree nodes that are seen.

DftPerspective = 3

KnownPerspectives = {8: {'model': [<class 'taurus.qt.qtcore.model.taurusdatabasemodel.TaurusDbServerProxyMod

getModelClass()

classmethod getQtDesignerPluginInfo()

sizeHint()

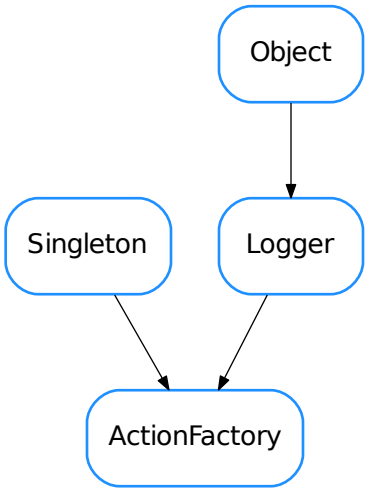
- *QBaseTreeWidget*
- *TaurusBaseTreeWidget*
- *TaurusDbTreeWidget*

taurus.qt.qtgui.util

This package provides a set of taurus wiget utilities like color management, configuration, actions.

Classes

ActionFactory



class ActionFactory

Bases: taurus.core.util.singleton.Singleton, taurus.core.util.log.Logger

A Singleton class designed to provide Action related objects.

buildAction(*widget*, *a_node*)

buildMenu(*widget*, *m_node*)

createAction(*parent*, *text*, *shortcut=None*, *icon=None*, *tip=None*, *toggled=None*, *triggered=None*, *data=None*, *context=1*)
Create a QAction

getActions()

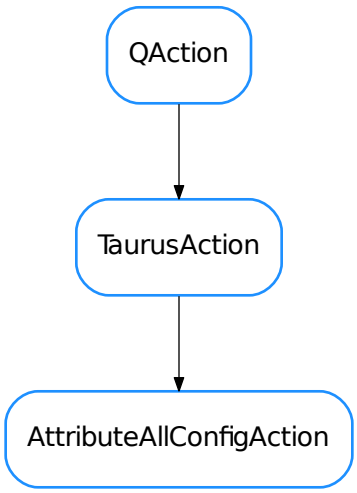
getMenus()

getNewAction(*widget*, *id*)

getNewMenu(*widget*, *data*)

init(**args*)
Singleton instance initialization.

AttributeAllConfigAction

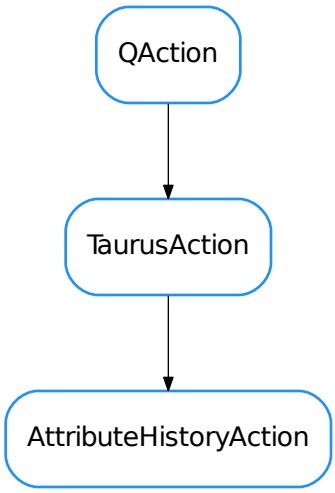


class AttributeAllConfigAction(*parent=None*)
Bases: taurus.qt.qtgui.util.taurusaction.TaurusAction

actionTriggered()

menuID = 'AttrConfig'

AttributeHistoryAction



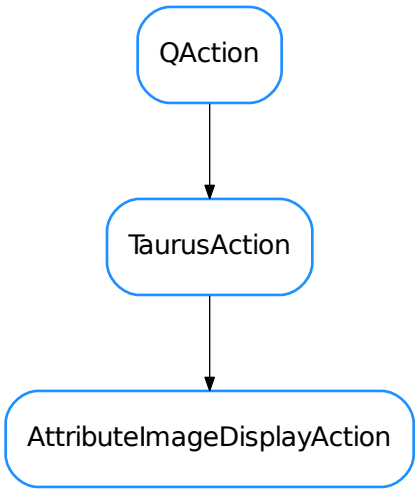
class **AttributeHistoryAction**(*parent=None*)

Bases: taurus.qt.qtgui.util.taurusaction.TaurusAction

actionTriggered()

menuID = 'AttrHistory'

AttributeImageDisplayAction



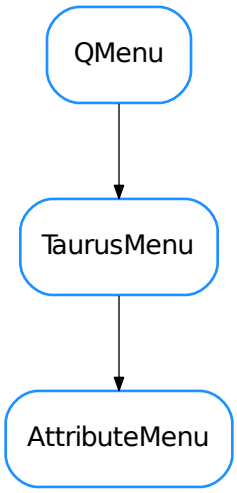
class **AttributeImageDisplayAction** (*parent=None*)

Bases: taurus.qt.qtgui.util.taurusaction.TaurusAction

actionTriggered()

menuID = 'ImageDisplay'

AttributeMenu



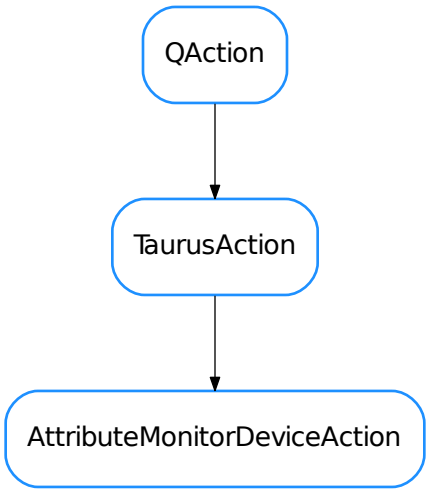
class AttributeMenu (*parent*)

Bases: taurus.qt.qtgui.util.taurusaction.TaurusMenu

menuData = "<Menu label='Attribute'><MenuItem class='AttrHistory'/><MenuItem class='_Separator_'/><Menu clas

menuID = 'AttrMenu'

AttributeMonitorDeviceAction



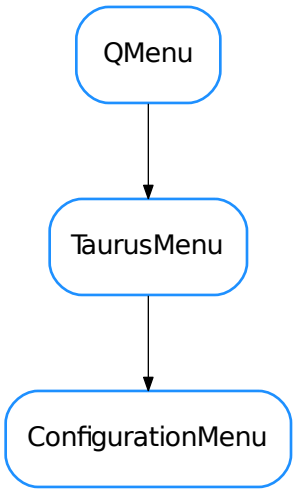
class `AttributeMonitorDeviceAction`(*parent=None*)

Bases: `taurus.qt.qtgui.util.taurusaction.TaurusAction`

`actionTriggered`()

`menuID` = 'MonitorDevice'

ConfigurationMenu



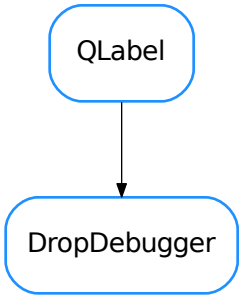
class ConfigurationMenu (*parent*)

Bases: taurus.qt.qtgui.util.taurusaction.TaurusMenu

menuData = "<Menu label='Configuration'><MenuItem class='AttrConfig'/></Menu>"

menuID = 'AttrConfigMenu'

DropDebugger



class DropDebugger (*parent=None*)

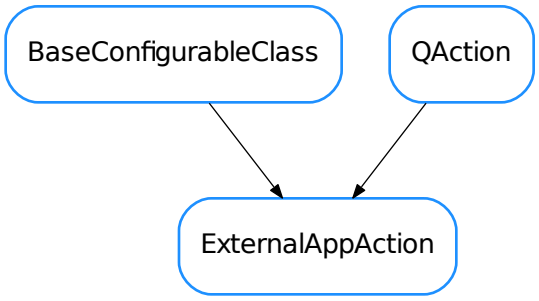
Bases: PyQt4.QtGui.QLabel

A simple utility for debugging drag&drop. This widget will accept drops and show a pop-up with the contents of the MIME data passed in the drag&drop

dragEnterEvent(*event*)

dropEvent(*event*)
reimplemented to support drag&drop of models. See `QWidget`

ExternalAppAction



class ExternalAppAction(*cmdargs*, *text=None*, *icon=None*, *parent=None*, *interactive=True*)
Bases: `PyQt4.QtGui.QAction`, `taurus.qt.qtcore.configuration.configuration.BaseConfigurableClass`

An specialized QAction for launching external applications

Signals: apart of those from QAction, it emits a "cmdArgsChanged" signal with the current cmdArgs list as its argument.

DEFAULT_ICON_NAME = 'application-x-executable'

actionTriggered

check()
Returns True if the application is available for executing

Return type `bool`

Returns

cmdArgs()

cmdArgsChanged

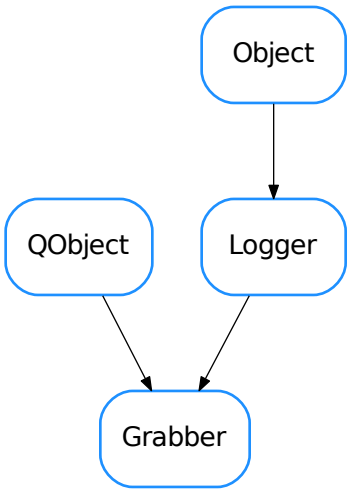
kill()

setCmdArgs(*cmdargs*, *emitsignal=True*)
Sets the command args for executing this external application.

It emits the "cmdArgsChanged" signal with the new cmdArgs list

Parameters **cmdargs** (`list` <`str`> or `str`) – A list of strings to be passed to `subprocess.Popen()` for launching the external application. It can also be a string containing a command, which will be automatically converted to a list

Grabber



class **Grabber**(*widget*, *fileName*)

Bases: PyQt4.QtCore.QObject, taurus.core.util.log.Logger

grab

grabTrigger()

static **grabWidget**(*widget*, *fileName*, *period=None*)

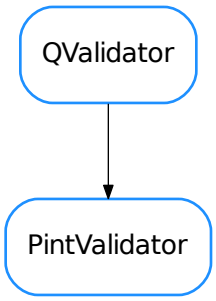
Grabs the given widget into the given image filename. If period is not given (or given with None) means grab immediately once and return. If period is given and >0 means grab the image every period seconds

Parameters

- **widget** (QWidget) – the qt widget to be grabbed
- **fileName** (str) – the name of the image file
- **period** (float) – period (seconds)

onGrab()

PintValidator



class **PintValidator**(**a*, ***kw*)

Bases: PyQt4.QtGui.QValidator

A QValidator for pint Quantities

bottom

Return type Quantity or None

Returns minimum accepted or None if it is not enforced

setBottom(*bottom*)

Set minimum limit

Parameters **bottom** (Quantity or None) – minimum acceptable value or None if it is not to be enforced

setTop(*top*)

Set maximum limit

Parameters **top** (Quantity or None) – maximum acceptable value or None if it is not to be enforced

setUnits(*units*)

Set implicit units. They will be assumed when the text does not explicit the unit. They will also be used for dimensionality coherence checks.

Parameters **units** (Unit or None) – . The implicit unit. If None, implicit dimension is "unitless" and no dimensionality checks will be performed (other than those inherent to range enforcement)

top

Return type Quantity or None

Returns maximum accepted or None if it is not enforced

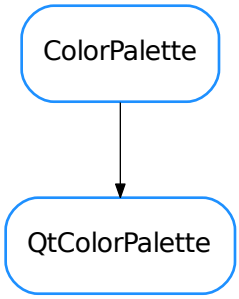
units

Return type Unit or None

Returns base units or None if it should not be enforced

validate(*input*, *pos*)
Reimplemented from `QValidator` to validate if the input string is a representation of a quantity within the set bottom and top limits

QtColorPalette



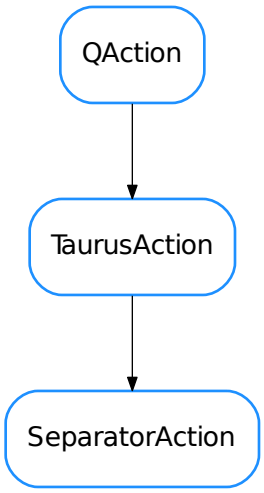
class QtColorPalette(*dat*, *int_decoder_dict*)
Bases: `taurus.core.util.colors.ColorPalette`

qbrush(*stoq*)
Returns the brush for the specified state or quality

qcolor(*stoq*)
Returns the color for the specified state or quality

qvariant(*stoq*)
Returns the color for the specified state or quality

SeparatorAction

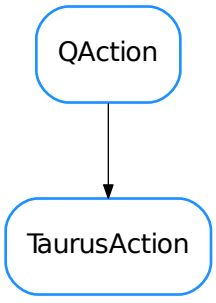


class SeparatorAction (*parent=None*)

Bases: taurus.qt.qtgui.util.taurusaction.TaurusAction

menuID = '_Separator_'

TaurusAction



class TaurusAction (*parent*)

Bases: PyQt4.QtGui.QAction

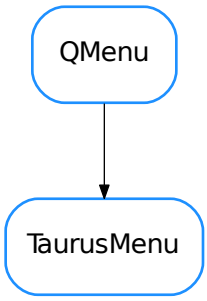
Base class for Taurus Actions

actionTriggered

modelChanged

update()

TaurusMenu



class TaurusMenu(*parent*)

Bases: PyQt4.QtGui.QMenu

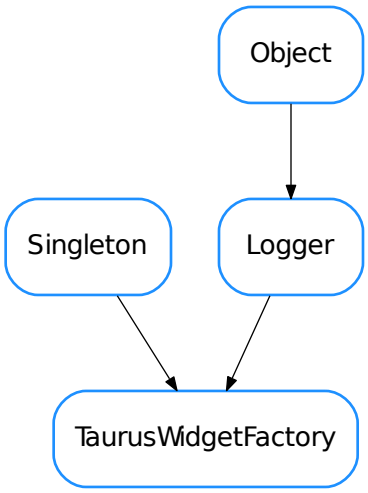
Base class for Taurus Menus

build(*data*)

buildFromXML(*m_node*)

getActionFactory()

TaurusWidgetFactory



class TaurusWidgetFactory

Bases: taurus.core.util.singleton.Singleton, taurus.core.util.log.Logger

The TaurusWidgetFactory is a utility class that provides information about all Qt widgets (Taurus and non Taurus) that are found in the current taurus distribution. TaurusWidgetFactory is a singleton class so you can freely create new instances since they will all reference the same object. Usage:

```
from taurus.qt.qtgui.util import TaurusWidgetFactory

wf = TaurusWidgetFactory()
print wf.getTaurusWidgetClassNames()
```

getTaurusWidgetClass(*name*)

getTaurusWidgetClassNames()

getTaurusWidgetClasses()

getTaurusWidgets()

getWidgetClass(*name*)

getWidgetClassNames()

getWidgetClasses()

getWidgets()

init(**args*)

Singleton instance initialization.

skip_modules = ('widget', 'util', 'qtdesigner', 'uic', 'resource')

- *ActionFactory*
- *AttributeAllConfigAction*

- *AttributeHistoryAction*
- *AttributeImageDisplayAction*
- *AttributeMenu*
- *AttributeMonitorDeviceAction*
- *ConfigurationMenu*
- *DropDebugger*
- *ExternalAppAction*
- *Grabber*
- *PintValidator*
- *QtColorPalette*
- *SeparatorAction*
- *TaurusAction*
- *TaurusMenu*
- *TaurusWidgetFactory*

Functions

UILoadable (*klass=None*, *with_ui=None*)

A class decorator intended to be used in a Qt.QWidget to make its UI loadable from a predefined QtDesigner UI file. This decorator will add a *loadUi()* method to the decorated class and optionaly a property with a name given by *with_ui* parameter.

The folowing example assumes the existence of the ui file `<my_widget_dir>/ui/MyWidget.ui` which is a QWidget panel with *at least* a QPushButton with objectName *my_button*

```
from taurus.external.qt import Qt
from taurus.qt.qtgui.util.ui import UILoadable

@UILoadable
class MyWidget(Qt.QWidget):

    def __init__(self, parent=None):
        Qt.QWidget.__init__(self, parent)
        self.loadUi()
        self.my_button.setText("This is MY button")
```

Another example using a `superUI.ui` file in the same directory as the widget. The widget UI components can be accessed through the widget member *_ui*

```
import os.path

from taurus.external.qt import Qt
from taurus.qt.qtgui.util.ui import UILoadable

@UILoadable(with_ui="_ui")
class MyWidget(Qt.QWidget):

    def __init__(self, parent=None):
        Qt.QWidget.__init__(self, parent)
        self.loadUi(filename="superUI.ui", path=os.path.dirname(__file__))
        self._ui.my_button.setText("This is MY button")
```

Parameters **with_ui** (*str*) – assigns a member to the decorated class from which you can access all UI components [default: None, meaning no member is created]

Warning: the current implementation (Jul14) doesn't prevent Qt from overloading any members you might have defined previously by the widget object names from the UI file. This happens even if *with_ui* parameter is given. For example, if the UI contains a QPushButton with objectName *my_button*:

```
@UILoadable(with_ui="_ui")
class MyWidget(Qt.QWidget):

    def __init__(self, parent=None):
        Qt.QWidget.__init__(self, parent)
        self.my_button = "hello"
        self.loadUi()
widget = MyWidget()
print widget.my_button
<PyQt4.QtGui.QPushButton object at 0x159e2f8>
```

This little problem should be solved in the next taurus version.

getWidgetsOfType(*widget*, *class_or_type_or_tuple*)
Returns all widgets in a hierarchy of a certain type

Parameters

- **widget** (*Qt.QWidget*) – the widget to be inspected
- **class-or-type-or-tuple** (*type class or a tuple of type classes*) – type to be checked

Returns a sequence containning all widgets in the hierarchy that match the given type

Return type seq<Qt.QWidget>

grabWidget(*widget*, *fileName*, *period=None*)
Grabs the given widget into the given image filename. If period is not given (or given with None) means grab immediately once and return. If period is given and >0 means grab the image every period seconds

Parameters

- **widget** (QWidget) – the qt widget to be grabbed
- **fileName** (str) – the name of the image file
- **period** (float) – period (seconds)

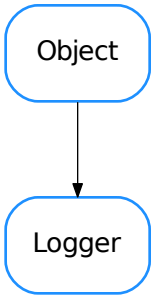
loadUi(*obj*, *filename=None*, *path=None*, *with_ui=None*)
Loads a QtDesigner .ui file into the given widget. If no filename is given, it tries to load from a file name which is the widget class name plus the extension ".ui" (example: if your widget class is called MyWidget it tries to find a MyWidget.ui). If path is not given it uses the directory where the python file which defines the widget is located plus a *ui* directory (example: if your widget is defined in a file /home/homer/workspace/taurusgui/my_widget.py then it uses the path /home/homer/workspace/taurusgui/ui)

Parameters

- **filename** (*str*) – the QtDesigner .ui file name [default: None, meaning calculate file name with the algorithm explained before]
- **path** (*str*) – directory where the QtDesigner .ui file is located [default: None, meaning calculate path with algorithm explained before]
- **with_ui** (*bool*) – if True, the objects defined in the ui file will be accessible as submembers of an ui member of the widget. If False, such objects will directly be members of the widget.

Classes

Logger



class Logger (*name=''*, *parent=None*, *format=None*)

Bases: taurus.core.util.object.Object

The taurus logger class. All taurus pertinent classes should inherit directly or indirectly from this class if they need taurus logging facilities.

Critical = 50
Critical message level (constant)

Debug = 10
Debug message level (constant)

DftLogFormat = <logging.Formatter object>
Default log format (constant)

DftLogLevel = 20
Default log level (constant)

DftLogMessageFormat = '%(threadName)-14s %(levelname)-8s %(asctime)s %(name)s: %(message)s'
Default log message format (constant)

Error = 40
Error message level (constant)

Fatal = 50
Fatal message level (constant)

Info = 20
Info message level (constant)

Trace = 5
Trace message level (constant)

Warning = 30
Warning message level (constant)

addChild (*child*)
Adds a new logging child

Parameters **child** (Logger) – the new child

classmethod addLevelName(*level_no*, *level_name*)
Registers a new log level

Parameters

- **level_no** (int) – the level number
- **level_name** (str) – the corresponding name

addLogHandler(*handler*)
Registers a new handler in this object's logger

Parameters **handler** (Handler) – the new handler to be added

classmethod addRootLogHandler(*h*)
Adds a new handler to the root logger

Parameters **h** (Handler) – the new log handler

changeLogName(*name*)
Change the log name for this object.

Parameters **name** (str) – the new log name

cleanUp()
The cleanUp. Default implementation does nothing Overwrite when necessary

copyLogHandlers(*other*)
Copies the log handlers of other object to this object

Parameters **other** (object) – object which contains 'log_handlers'

critical(*msg*, **args*, ***kw*)
Record a critical message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.critical().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

debug(*msg*, **args*, ***kw*)
Record a debug message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.debug().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

deprecated(*msg=None*, *dep=None*, *alt=None*, *rel=None*, *dbg_msg=None*, *_callerinfo=None*, ***kw*)
Record a deprecated warning message in this object's logger. If message is not passed, a estandard deprecation message is constructued using dep, alt, rel arguments. Also, an extra debug message can be recorded, followed by traceback info.

Parameters

- **msg** (str) – the message to be recorded (if None passed, it will be constructed using dep (and, optionally, alt and rel)

- **dep** (str) – name of deprecated feature (in case msg is None)
- **alt** (str) – name of alternative feature (in case msg is None)
- **rel** (str) – name of release from which the feature was deprecated (in case msg is None)
- **dbg_msg** (str) – msg for debug (or None to log only the warning)
- **_callerinfo** – for internal use only. Do not use this argument.
- **kw** – any additional keyword arguments, are passed to logging.Logger.warning()

classmethod disableLogOutput()
Disables the logging.StreamHandler which dumps log records, by default, to the stderr.

classmethod enableLogOutput()
Enables the logging.StreamHandler which dumps log records, by default, to the stderr.

error(*msg*, **args*, ***kw*)
Record an error message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.error().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

exception(*msg*, **args*)
Log a message with severity 'ERROR' on the root logger, with exception information.. Accepted *args* are the same as logging.Logger.exception().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments

fatal(*msg*, **args*, ***kw*)
Record a fatal message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.fatal().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

flushOutput()
Flushes the log output

getChildren()
Returns the log children for this object

Return type Logger

Returns the list of log children

classmethod getLogFormat()
Retuns the current log message format (the root log format)

Return type str

Returns the log message format

getLogFullName()
Gets the full log name for this object

Return type str

Returns the full log name

classmethod getLogLevel()
Retuns the current log level (the root log level)

Return type int

Returns a number representing the log level

getLogName()
Gets the log name for this object

Return type str

Returns the log name

getLogObj()
Returns the log object for this object

Return type Logger

Returns the log object

classmethod getLogger(*name=None*)

getParent()
Returns the log parent for this object or None if no parent exists

Return type Logger or None

Returns the log parent for this object

classmethod getRootLog()
Retuns the root logger

Return type Logger

Returns the root logger

info(*msg*, **args*, ***kw*)
Record an info message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.info().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

classmethod initRoot()
Class method to initialize the root logger. Do **NOT** call this method directly in your code

log(*level*, *msg*, **args*, ***kw*)
Record a log message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.log().

Parameters

- **level** (int) – the record level
- **msg** (str) – the message to be recorded

- **args** – list of arguments
- **kw** – list of keyword arguments

log_format = <logging.Formatter object>
Default log message format

log_level = 20
Current global log level

classmethod removeRootLogHandler(*h*)
Removes the given handler from the root logger

Parameters **h** (Handler) – the handler to be removed

classmethod resetLogFormat()
Resets the log message format (the root log format)

classmethod resetLogLevel()
Resets the log level (the root log level)

root_init_lock = <thread.lock object>
Internal usage

root_inited = True
Internal usage

classmethod setLogFormat(*format*)
sets the new log message format

Parameters **level** (str) – the new log message format

classmethod setLogLevel(*level*)
sets the new log level (the root log level)

Parameters **level** (int) – the new log level

stack(*target=5*)
Log the usual stack information, followed by a listing of all the local variables in each frame.

Parameters **target** (int) – the log level assigned to the record

Return type str

Returns The stack string representation

stream_handler = <logging.StreamHandler object>
the main stream handler

syncLog()
Synchronises the log output

trace(*msg*, **args*, ***kw*)
Record a trace message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.log().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

traceback(*level=5*, *extended=True*)
Log the usual traceback information, followed by a listing of all the local variables in each frame.

Parameters

- **level** (int) – the log level assigned to the traceback record
- **extended** (bool) – if True, the log record message will have multiple lines

Return type str

Returns The traceback string representation

warning(*msg*, **args*, ***kw*)

Record a warning message in this object's logger. Accepted *args* and *kwargs* are the same as logging.Logger.warning().

Parameters

- **msg** (str) – the message to be recorded
- **args** – list of arguments
- **kw** – list of keyword arguments

- *Logger*

Functions

Attribute(*dev_or_attr_name*, *attr_name=None*)

Returns the taurus attribute for either the pair (device name, attribute name) or full attribute name

•Attribute(full_attribute_name)

•Attribute(device_name, attribute_name)

It is a shortcut to:

```
import taurus.core.taurusmanager
manager = taurus.core.taurusmanager.TaurusManager()
factory = manager.getFactory()
attribute  = factory.getAttribute(full_attribute_name)
```

or:

```
import taurus.core.taurusmanager
manager = taurus.core.taurusmanager.TaurusManager()
factory = manager.getFactory()
device  = factory.getDevice(device_name)
attribute = device.getAttribute(attribute_name)
```

Parameters

- **dev_or_attr_name** (*str or* TaurusDevice) – the device name or full attribute name
- **attr_name** (*str*) – attribute name

Returns a taurus attribute

Return type taurus.core.taurusattribute.TaurusAttribute

Authority(*name=None*)

Returns a taurus authority

It is a shortcut to:

```
import taurus.core.taurusmanager
manager = taurus.core.taurusmanager.TaurusManager()
factory = manager.getFactory()
db  = factory.getAuthority(dname)
```

Parameters **name** (*str or None*) – authority name. If None (default) it will return the default authority of the default scheme. For example, if the default scheme is tango, it will return the default TANGO_HOST database

Returns a taurus authority

Return type taurus.core.taurusauthority.TaurusAuthority

Configuration(**args*, ***kwargs*)

Returns the taurus configuration for either the pair (attribute name, conf name) or full conf name

- Configuration(full_conf_name)
- Configuration(attribute_name, conf_name)

It is a shortcut to:

```
import taurus.core.taurusmanager
manager = taurus.core.taurusmanager.TaurusManager()
factory = manager.getFactory()
conf  = factory.getConfiguration(attr_or_conf_name)
```

or:

```
import taurus.core.taurusmanager
manager = taurus.core.taurusmanager.TaurusManager()
factory = manager.getFactory()
attribute  = factory.getAttribute(attribute_name)
conf = attribute.getConfig(conf_name)
```

param attr_or_conf_name the full attribute name or full conf name

type attr_or_conf_name str

param conf_name conf name

type conf_name str or None

return a taurus configuration

rtype taurus.core.taurusconfiguration.TaurusConfiguration

Deprecated since version 4.0: Use Attribute instead

Database(**args*, ***kwargs*)

Deprecated since version 4.0: Use Authority instead

Device(*device_name*)

Returns the taurus device for the given device name

It is a shortcut to:

```
import taurus.core.taurusmanager
manager = taurus.core.taurusmanager.TaurusManager()
```

```
factory = manager.getFactory()
device  = factory.getDevice(device_name)
```

Parameters **device_name** (*str*) – the device name

Returns a taurus device

Return type taurus.core.taurusdevice.TaurusDevice

Factory(*scheme=None*)

Returns the one and only Factory for the given scheme

It is a shortcut to:

```
import taurus.core.taurusmanager
manager = taurus.core.taurusmanager.TaurusManager()
factory = manager.getFactory(scheme)
```

Parameters **scheme** (*str*) – a string representing the scheme. Default value is None meaning tango scheme

Returns a taurus factory

Return type taurus.core.taurusfactory.TaurusFactory

Manager()

Returns the one and only TaurusManager

It is a shortcut to:

```
import taurus.core
manager = taurus.core.taurusmanager.TaurusManager()
```

Returns the TaurusManager

Return type taurus.core.taurusmanager.TaurusManager

See also:

taurus.core.taurusmanager.TaurusManager

Object(**args*)

Returns an taurus object of given class for the given name

Can be called as:

•Object(name)

•Object(cls, name)

Where:

•*name* is a model name (str)

•*cls* is a class derived from TaurusModel

If *cls* is not given, Object() will try to guess it from *name*.

Returns a taurus object

Return type taurus.core.taurusmodel.TaurusModel

changeDefaultPollingPeriod(*period*)

check_dependencies()
Prints a check-list of requirements and marks those that are fulfilled

critical(*msg*, **args*, ***kw*)

debug(*msg*, **args*, ***kw*)

deprecated(**args*, ***kw*)

error(*msg*, **args*, ***kw*)

fatal(*msg*, **args*, ***kw*)

getSchemeFromName(*name*, *implicit=True*)
Return the scheme from a taurus name.

Parameters

- **name** (str) – taurus model name URI.
- **implicit** (bool) – controls whether to return the default scheme (if implicit is True -default-) or None (if implicit is False) in case *model* does not contain the scheme name explicitly. The default scheme may be defined in *Taurus custom settings* ('tango' is assumed if not defined)

getValidTypesForName(*name*, *strict=None*)
Returns a list of all Taurus element types for which *name* is a valid model name (while in many cases a name may only be valid for one element type, this is not necessarily true in general)

Parameters

- **name** (str) – taurus model name
- **strict** (bool) – If True, names that are not RFC3986-compliant but which would be accepted for backwards compatibility are considered valid.

Return type list <element>

Returns where element can be one of: *Attribute*, *Device* or *Authority*

info(*msg*, **args*, ***kw*)

isValidName(*name*, *etypes=None*, *strict=None*)
Returns True is the given name is a valid Taurus model name. If *etypes* is passed, it returns True only if name is valid for at least one of the given the element types. Otherwise it returns False. For example:

```
isValidName('tango:foo')--> True
isValidName('tango:a/b/c', [TaurusElementType.Attribute]) --> False
```

Parameters

- **name** (str) – the string to be checked for validity
- **etypes** (seq <TaurusElementType>) – if given, names will only be considered valid if they represent one of the given element types. Supported element types are: *Attribute*, *Device* and *Authority*
- **strict** (bool) – If True, names that are not RFC3986-compliant but which would be accepted for backwards compatibility are considered valid.

Return type bool

Returns

log(*level*, *msg*, **args*, ***kw*)

log_dependencies()
deprecated since '4.0.4'

makeSchemeExplicit(*name*, *default=None*)
return the name guaranteeing that the scheme is present. If name already contains the scheme, it is returned unchanged.

Parameters

- **name** (str) – taurus model name URI.
- **default** (str) – The default scheme to use. If no default is passed, the one defined in tauruscustomsettings.DEFAULT_SCHEME is used.

Returns the name with the explicit scheme.

trace(*msg*, **args*, ***kw*)

warning(*msg*, **args*, ***kw*)

All Classes for taurus

- *AboutDialog*
- *ActionFactory*
- *AppSettingsWizard*
- *ArrayBuffer*
- *ArrayEditor*
- *AttrQuality*
- *AttributeAllConfigAction*
- *AttributeEventIterator*
- *AttributeEventWait*
- *AttributeHistoryAction*
- *AttributeImageDisplayAction*
- *AttributeMenu*
- *AttributeMonitorDeviceAction*
- *BZ2Codec*
- *BaseConfigurableClass*
- *BaseToolBar*
- *BoundMethodWeakref*
- *CCDPVCAM*
- *CaselessDefaultDict*
- *CaselessDict*
- *CaselessList*
- *CaselessWeakValueDict*
- *CircBuf*
- *Codec*
- *CodecFactory*
- *CodecPipeline*
- *ColorPalette*
- *ConfigEventGenerator*
- *ConfigurationMenu*
- *CriticalIt*
- *CurveAppearanceProperties*
- *CurvePropertiesView*
- *CurveStatsDialog*

- *CurvesAppearanceChooser*
- *DataModel*
- *DateTimeScaleEngine*
- *DebugIt*
- *DefaultLabelWidget*
- *DefaultTaurusValueCheckBox*
- *DefaultThreadDict*
- *DefaultUnitsWidget*
- *DeltaTimeScaleDraw*
- *DeltaTimeScaleEngine*
- *DevState*
- *DockWidgetPanel*
- *DoubleRegistration*
- *DropDebugger*
- *EditorToolBar*
- *EnumException*
- *Enumeration*
- *EpicsFactory*
- *ErrorIt*
- *EvaluationAttribute*
- *EvaluationAuthority*
- *EvaluationDevice*
- *EvaluationFactory*
- *EventGenerator*
- *EventListener*
- *ExternalAppAction*
- *ExternalAppEditor*
- *Falcon*
- *FancyScaleDraw*
- *FilterToolBar*
- *FixedLabelsScaleDraw*
- *FixedLabelsScaleEngine*
- *FunctionCodec*
- *Grabber*
- *GraphicalChoiceDlg*
- *GraphicalChoiceWidget*
- *HelpPanel*
- *ImageCounterDevice*
- *ImageDevice*
- *ImgBeamAnalyzer*
- *ImgGrabber*
- *InfoIt*
- *JSONCodec*
- *LIFO*
- *LimaCCDs*
- *ListEventGenerator*
- *LogExceptHook*
- *LogFilter*
- *LogIt*
- *Logger*
- *Logger*
- *Logger*
- *Logger*
- *LoopList*

- *MacroServerMessageErrorHandler*
- *MemoryLogHandler*
- *NullCodec*
- *Object*
- *PanelDescriptionWizard*
- *PerspectiveToolBar*
- *PintValidator*
- *PlotCodec*
- *ProtectTaurusMessageBox*
- *PyImageViewer*
- *Q7SegDigit*
- *QBaseModelWidget*
- *QBaseTableWidget*
- *QBaseTreeWidget*
- *QButtonBox*
- *QConfigEditor*
- *QDataExportDialog*
- *QDictionaryEditor*
- *QDoubleListDlg*
- *QEmitter*
- *QFallBackWidget*
- *QGraphicsTextBoxing*
- *QGroupWidget*
- *QIconCatalog*
- *QLed*
- *QLedOld*
- *QListEditor*
- *QLoggingTable*
- *QLoggingTableModel*
- *QLoggingWidget*
- *QLogo*
- *QPixmapWidget*
- *QRawDataWidget*
- *QRemoteLoggingTableModel*
- *QSpline*
- *QWheelEdit*
- *QtColorPalette*
- *RefreshToolBar*
- *ResourcesFactory*
- *SafeEvaluator*
- *ScanTrendsSet*
- *SelectionToolBar*
- *SeparatorAction*
- *SharedDataManager*
- *Singleton*
- *Singleton*
- *TangoAttrInfo*
- *TangoAttrValue*
- *TangoAttribute*
- *TangoAttributeEventListener*
- *TangoAuthority*
- *TangoConfigLineEdit*
- *TangoConfiguration*
- *TangoDatabase*

- *TangoDatabaseCache*
- *TangoDevClassInfo*
- *TangoDevInfo*
- *TangoDevice*
- *TangoFactory*
- *TangoInfo*
- *TangoMessageErrorHandler*
- *TangoServInfo*
- *TaurusAction*
- *TaurusApplication*
- *TaurusArrayEditor*
- *TaurusArrayEditorButton*
- *TaurusAttrForm*
- *TaurusAttrListComboBox*
- *TaurusAttrValue*
- *TaurusAttribute*
- *TaurusAttributeControllerHelper*
- *TaurusAttributeNameValidator*
- *TaurusAuthority*
- *TaurusAuthorityNameValidator*
- *TaurusBaseComponent*
- *TaurusBaseContainer*
- *TaurusBaseController*
- *TaurusBaseEditor*
- *TaurusBaseGraphicsFactory*
- *TaurusBaseModel*
- *TaurusBaseModelWidget*
- *TaurusBaseProxyModel*
- *TaurusBaseTableWidget*
- *TaurusBaseTreeItem*
- *TaurusBaseTreeWidget*
- *TaurusBaseWidget*
- *TaurusBaseWritableWidget*
- *TaurusBoolRW*
- *TaurusCommandButton*
- *TaurusCommandsForm*
- *TaurusConfigLineEdit*
- *TaurusConfigValue*
- *TaurusConfiguration*
- *TaurusConfigurationControllerHelper*
- *TaurusConfigurationPanel*
- *TaurusConfigurationProxy*
- *TaurusConsole*
- *TaurusCurve*
- *TaurusCurveDialog*
- *TaurusCurveMarker*
- *TaurusDbBaseModel*
- *TaurusDbBaseProxyModel*
- *TaurusDbDeviceClassModel*
- *TaurusDbDeviceClassProxyModel*
- *TaurusDbDeviceModel*
- *TaurusDbDeviceProxyModel*
- *TaurusDbPlainDeviceModel*
- *TaurusDbPlainServerModel*

- *TaurusDbServerModel*
- *TaurusDbServerProxyModel*
- *TaurusDbSimpleDeviceAliasModel*
- *TaurusDbSimpleDeviceModel*
- *TaurusDbTableWidget*
- *TaurusDbTreeWidget*
- *TaurusDevButton*
- *TaurusDevPanel*
- *TaurusDevState*
- *TaurusDevice*
- *TaurusDeviceNameValidator*
- *TaurusDevicePanel*
- *TaurusEllipseStateItem*
- *TaurusExceptHookMessageBox*
- *TaurusException*
- *TaurusException*
- *TaurusExceptionListener*
- *TaurusFactory*
- *TaurusFactory*
- *TaurusFallBackWidget*
- *TaurusFallBackWidget*
- *TaurusForm*
- *TaurusFrame*
- *TaurusGraphicsAttributeItem*
- *TaurusGraphicsItem*
- *TaurusGraphicsScene*
- *TaurusGraphicsStateItem*
- *TaurusGraphicsUpdateThread*
- *TaurusGraphicsView*
- *TaurusGrid*
- *TaurusGroupBox*
- *TaurusGroupItem*
- *TaurusGroupStateItem*
- *TaurusGroupWidget*
- *TaurusGui*
- *TaurusImageDialog*
- *TaurusInputDialog*
- *TaurusInputPanel*
- *TaurusJDrawGraphicsFactory*
- *TaurusJDrawGraphicsFactory*
- *TaurusJDrawSynopticsView*
- *TaurusJDrawSynopticsView*
- *TaurusLCD*
- *TaurusLabel*
- *TaurusLabelEditRW*
- *TaurusLauncherButton*
- *TaurusLed*
- *TaurusLineStateItem*
- *TaurusListener*
- *TaurusLockButton*
- *TaurusLockInfo*
- *TaurusMainWindow*
- *TaurusManager*
- *TaurusMenu*

- *TaurusMessageBox*
- *TaurusMessageErrorHandler*
- *TaurusMessagePanel*
- *TaurusMessageReportHandler*
- *TaurusModel*
- *TaurusModelChooser*
- *TaurusModelItem*
- *TaurusModelList*
- *TaurusModelModel*
- *TaurusModelSelectorTree*
- *TaurusModelValue*
- *TaurusMonitorTiny*
- *TaurusNeXusBrowser*
- *TaurusOperation*
- *TaurusPlot*
- *TaurusPlotButton*
- *TaurusPlotConfigDialog*
- *TaurusPollingTimer*
- *TaurusPolygonStateItem*
- *TaurusPropTable*
- *TaurusReadWriteSwitcher*
- *TaurusRectStateItem*
- *TaurusRoundRectItem*
- *TaurusRoundRectStateItem*
- *TaurusScalarAttributeControllerHelper*
- *TaurusScrollArea*
- *TaurusSplineStateItem*
- *TaurusTextAttributeItem*
- *TaurusTextStateItem*
- *TaurusTimeScaleDraw*
- *TaurusTimeVal*
- *TaurusTreeAttributeItem*
- *TaurusTreeDeviceClassItem*
- *TaurusTreeDeviceDomainItem*
- *TaurusTreeDeviceFamilyItem*
- *TaurusTreeDeviceItem*
- *TaurusTreeDeviceMemberItem*
- *TaurusTreeDevicePartItem*
- *TaurusTreeServerItem*
- *TaurusTreeServerNameItem*
- *TaurusTreeSimpleDeviceItem*
- *TaurusTrend*
- *TaurusTrend2DDialog*
- *TaurusTrendDialog*
- *TaurusTrendsSet*
- *TaurusValue*
- *TaurusValueCheckBox*
- *TaurusValueComboBox*
- *TaurusValueLineEdit*
- *TaurusValueSpinBox*
- *TaurusValueSpinBoxEx*
- *TaurusValuesFrame*
- *TaurusValuesTable*
- *TaurusValuesTableButton*

- *TaurusValuesTableButton_W*
- *TaurusWheelEdit*
- *TaurusWidget*
- *TaurusWidgetFactory*
- *TaurusWidgetPlugin*
- *TaurusXValues*
- *ThreadDict*
- *ThreadPool*
- *TimedQueue*
- *Timer*
- *TraceIt*
- *WarnIt*
- *Worker*
- *WriteAttrOperation*
- *ZIPCodec*
- *configurableProperty*
- *defaultdict*
- *defaultdict_fromkey*

Taurus Enhancement Proposals

- genindex
- modindex
- search

Last Update Aug 17, 2017

Release 4.1.2-alpha

Python Module Index

t

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z