TwinPy

Robert Roos

May 31, 2022

CONTENTS:

TwinPy is a package containing tools to easily create your own Python GUIs for TwinCAT based on Simulink models. The GUIs are based on PyQt5 and the TwinCAT interface is done through pyads.

CHAPTER

ONE

INSTALLATION

The modules are best installed using `pip`. This is explained in the ReadMe of the repository.

CHAPTER

TWO

USING TWINPY WITH A REMOTE TARGET

You can also use TwinPy (and other pyads applications) remotely. In this scenario a client PC runs the GUI, interacting with a running TwinCAT instance on a target PC.

Using TwinPy remotely involves two steps:

- Getting access to the compiled Simulink model's XML file for the model structure
- Making a remote ADS connection

Tip: Often in a target and client situation, the client is also the development PC (which compiles the TwinCAT solution for the target). When this is the case, giving access to the XML files for TwinPy is trivial since they already exist on the client computer.

2.1 Access to the XML file

When the client is not the computer that compiles the Simulink model, it won't automatically have access to the most recent model XML. There are a few approaches:

- **Compile the model also on the client PC**
  - This will require all the compile dependencies on the client PC, and each compilation has to be done twice.
- **Copy the XML from the compiled model to the client PC**
  - No extra compilation is needed, but copying the XML each time can be time consuming.
- **Set up a file share to give direct access to the compiled XMLs**
  - By sharing the XMLs directly a client has access to the model structure without extra steps

2.1.1 Sharing the XML files directly

On the development PC (could be the same as the target PC), create a network share for the TwinCAT modules directory. By default this is `C:\TwinCAT\3.1\CustomConfig\Modules`. Simply right-click on the `Modules` directory in Windows explorer and click 'Give access to…'. Complete the wizard by adding the client PC. Read-only permissions should be all that's needed.

On the client PC you should now be able to find the PC and view the modules. Note the absolute path, which might be something like `\\<ip>\Modules`.

Search for info on Windows file sharing in case you run into problems.

2.2 Remote ADS connection

See the pyads documentation on routing for more information: https://pyads.readthedocs.io/en/latest/documentation/routing.html

2.2.1 Adding routes on Windows

Use the TwinCAT UI and add the remote. Right-click on the TwinCAT icon in the taskbar and click 'Router…' > 'Edit Routes'.

This gif exemplifies the procedure:

The route must *not* be set to 'unidirectional', which seems to be the default.

Note that you might need to add allow-rules in the firewall for both inbound and outbound traffic on TCP ports 48898 and 8016, and UPD port 48899.

2.2.2 Adding routes on Linux

The `pyads.connection.Connection` will create a route from the client to the target. You can then use `pyads.ads.add_route_to_plc()` to create a route back to the client. Or you can use the TwinCAT UI on the remote target to create the route back.

2.3 TwinPy Remote

In your application script, set the IP address, AMS net id and port correctly for the remote target when instantiating *`TwincatConnection`*.

In case no local XML file is available, specify an absolute path to the XML file when creating a *`SimulinkModel`*.

CHAPTER

THREE

API

3.1 TwinCAT and Simulink Interfacing

3.1.1 TwinCAT Connection

class twinpy.twincat.connection.**TwincatConnection**(*ams_net_id: str = '127.0.0.1.1.1'*, *ams_net_port: int = 350*, *ip_address: Optional[str] = None*)

Bases: Connection

Extend default Connection object (typically named *plc*).

ADS connection with custom features.

Note that this version will connect on object creation, throwing an exception when it fails. *pyads.Connection* waits for *.open()* and will fail quietly.

Parameters

- **ams_net_id** – TwinCAT AMS address (default is localhost)
- **ams_net_port** – ADS Port (default is 350)
- **ip_address** – Target IP (automatically deduced from AMS address)

Raises

pyads.ADSError – When connection failed

get_module_info(*module_name: str*) → dict

Get information about live module.

get_parameter(*name: Optional[str] = None*, *index_group: Optional[int] = None*, *index_offset: Optional[int] = None*, *symbol_type: Optional[Union[str, Type]] = None*) → *Parameter*

Get Parameter instance.

See Parameter.

get_signal(*name: Optional[str] = None*, *index_group: Optional[int] = None*, *index_offset: Optional[int] = None*, *symbol_type: Optional[Union[str, Type]] = None*) → *Signal*

Get Signal instance.

See Signal.

read_list_of_symbols(*symbols: List[AdsSymbol]*, *ads_sub_commands: int = 500*) → Dict[AdsSymbol, Any]

Read a list of symbols in a single request.

Same principe as *read_list_by_name*. See read_list_by_name() for more info.

This version doesn't work for structs.

The *_value* property for each symbol will be updated. A dictionary will also be returned of the symbol names and their new values.

write_list_of_symbols(*symbols_and_values: Dict[AdsSymbol, Any]*, *ads_sub_commands: int = 500*) → Dict[AdsSymbol, str]

Write new values to a list of symbols.

Same principe as *write_list_by_name*. See `write_list_by_name()` for more info.

For example:

```
# Using dict
new_data = {symbol1: 3.14, symbol2: False}
plc.write_list_of_symbols(new_data)
```

Parameters

- **symbols_and_values** – Symbols to write to
- **ads_sub_commands** – Max. number of symbols per call (see *write_list_by_name*)

3.1.2 Simulink

Model to wrap around a Simulink model.

An object for a Simulink model is created first before a TwinCAT connection is made. We cannot get the original model structure from TwinCAT alone.

twinpy.twincat.simulink.**sanitize_name**(*name: str*) → str

Reduce a string to characters which are allowed in a Python variable name.

This is needed because Simulink blocks can contain more characters than this. Python variables can only contain a-z, A-Z, 0-9 and '_'. Additionally, a variable cannot start with a digit, nor can it start with an underscore to prevent conflicts with semi-private properties.

SimulinkModel

class twinpy.twincat.**SimulinkModel**(*object_id: int*, *object_name: str*, *type_name: Optional[str] = None*)

Bases: *SimulinkBlock*

Wrapper for a compiled Simulink model in TwinCAT.

The model is built using the XML file, created when the model is compiled. Therefore the model can be loaded without TwinCAT running.

This model object is actually an extension of a SimulinkBlock. The complete model is basically just the root block.

By default the *TWINCAT3DIR* environment variable is used to locate the TwinCAT installation and look for the installed compiled XML files.

To work around this, you can pass either of the following to *type_name*:

- A single name (*TWINCAT3DIR* will be used)
- A path to a directory (default XML file name will be searched)
- A path to the XML file, typically named like **_ModuleInfo.xml* (no searching will be done)

Parameters

- **object_id** – ID of the TcCOM object in TwinCAt (the symbol group index)
- **object_name** – Object Name (as shown in TwinCAT)
- **type_name** – Type name (as shown in TwinCAT) (defaults to be the same as object_name).

connect_to_twincat(*connection:* TwincatConnection)

Connect model the one running in TwinCAT.

This will link all the symbols in the model to actual ADS symbols. And the remote model is compared to the local .xml file through the model checksum.

Parameters
connection – Connection object to connect through

get_index_group() → int

Return the group index (owned by model).

static get_module_info(*xmltree: Element*) → dict

Get dictionary of module info fields.

The *DefaultValues* section is a list of names and values, this method creates a regular dict from it.

get_plc() → Optional[*TwincatConnection*]

Return Connection (owned by model).

static get_xml_data(*type_name: str*) → Element

Find and parse model XML file.

The block diagram is returned.

SimulinkBlock

class twinpy.twincat.**SimulinkBlock**(*xmltree: Element*, *model:* SimulinkModel)

Bases: object

A single Simulink Block (anything, e.g. constant, gain, a sub-system)

A SimulinkBlock can contain children, which are also SimulinkBlock objects. Using __getattr__ those subblocks (and their symbols) can be addressed directly:

model = ... # Subblocks can be addressed smoothly: print(model.MySubsystem.MyConstant.Value)

Blocks contain parameters (*Value*) in the example above. When only a single parameter or signal is present, you can directly call it from the block itself:

print(model.MySubsystem.MyConstant.get()) # Short print(model.MySubsystem.MyConstant.Value.get()) # Same but longer

print(model.MySubsystem.MySineWave.Phase.get()) # Multiple parameters print(model.MySubsystem.MySineWave.Amplitude.get())

Create this block based on an XML tree

Sub-blocks are created too based on the remaining tree structure. This means the creation of blocks works recursively.

Parameters

- **xmltree** – A branch of a model XML tree (or the entire tree)

- **model** – A reference back to the original model (the root of the structure)

get()

Get value of the first symbol.

get_index_group() → int

Return the group index (owned by model).

get_plc() → Optional[*TwincatConnection*]

Return Connection (owned by model).

get_symbols_recursive() → List[*Symbol*]

Recursively navigate subblocks and collect all parameters and signals.

make_parameters(*xmltree: Element*) → dict

Find and create Parameters in the current block.

make_signals(*xmltree: Element*) → dict

Find and create Signals in the current block.

make_subblocks(*xmltree: Element*) → Dict[str, *SimulinkBlock*]

Build sub-blocks (this makes the SimulinkBlocks recursive).

print_structure(*max_depth: Optional[int] = 3*, *depth: int = 0*)

Recursively print the child signals and parameters of this block.

Use this to test your model from the command line.

Parameters

- **max_depth** – Max recursion depth (set to None for infinite)
- **depth** – Current depth (do not use this argument, it's used internally)

set(*val*)

Set value of the first symbol.

3.1.3 ADS Variables

Module with classes that wrap around TwinCAT symbols.

With 'symbol' we mean ADS variable.

Symbol

class twinpy.twincat.**Symbol**(*block: Optional[SimulinkBlock] = None*, *plc: pyads.Connection = None*, *name: Optional[str] = None*, *index_group: Optional[int] = None*, *index_offset: Optional[int] = None*, *symbol_type: Optional[Union[str, Type]] = None*)

Bases: AdsSymbol, ABC

Base (abstract) class for a TwinCAT symbol.

Extends pyads.AdsSymbol - Introduced in pyads 3.3.1

A symbol (or a Symbol sub-class) is typically owned by a block in a Simulink model. Each symbol contains a reference back to the block that owns it, which can be used to trace back to the model that owns that block. The symbol needs a reference to the connection object directly.

Symbols can be created from a block or manually (either based on name or by providing all information).

Variables
value – The **buffered** value, *not* necessarily the latest value. The buffer is updated on each read, write and notification callback. It can be useful when the value needs to be applied multiple times, to avoid storing the value in your own variable.

See `pyads.Symbol`. If a block was passed, index_group and plc are automatically extracted from it and do not need to passed too.

Additional arguments:

Parameters
block – Block that owns this symbol (default: None)

Raises
ValueError –

add_device_notification(*callback: Callable[[Any], None]*, *attr: Optional[NotificationAttrib] = None*, *user_handle: Optional[int] = None*) → Optional[Tuple[int, int]]

Add on-change callback to symbol.

Superclass method is used, this version adds a wrapper for the callback to set the variable type. The user-defined callback will be called with the new symbol value as an argument.

del_device_notification(*handles: Tuple[int, int]*)

Remove a single device notification by handles.

get()

Get the symbol value from TwinCAT.

Simply an alias for *read()*.

get_value_from_string(*text: str*) → Any

Parse a string to the right data type.

read() → Any

Read the current value of this symbol.

The new read value is also saved in the buffer. Overridden from AdsSymbol, to work without an open Connection.

set(*val*)

Write the symbol in TwinCAT.

Simply an alias for *write()*.

set_connection(*connection: Optional[Connection]*)

Update the connection reference.

write(*new_value: Optional[Any] = None*) → None

Write a new value or the buffered value to the symbol.

When a new value was written, the buffer is updated. Overridden from AdsSymbol, to work without an open Connection

:param new_value Value to be written to symbol (if None,
the buffered value is send instead)

Parameter

class twinpy.twincat.**Parameter**(*block: Optional[SimulinkBlock] = None*, *plc: pyads.Connection = None*, *name: Optional[str] = None*, *index_group: Optional[int] = None*, *index_offset: Optional[int] = None*, *symbol_type: Optional[Union[str, Type]] = None*)

Bases: *Symbol*

A TwinCAT parameter.

A constant setting, e.g. a gain block value, constant block value. For read/write access. Needs no changes, can use the default.

See *Symbol*.

See `pyads.Symbol`. If a block was passed, index_group and plc are automatically extracted from it and do not need to passed too.

Additional arguments:

Parameters
block – Block that owns this symbol (default: None)

Raises
ValueError –

Signal

class twinpy.twincat.**Signal**(*block: Optional[SimulinkBlock] = None*, *plc: pyads.Connection = None*, *name: Optional[str] = None*, *index_group: Optional[int] = None*, *index_offset: Optional[int] = None*, *symbol_type: Optional[Union[str, Type]] = None*)

Bases: *Symbol*

A TwinCAT signal.

Typically a port, e.g. a subsystem input or output

See `pyads.Symbol`. If a block was passed, index_group and plc are automatically extracted from it and do not need to passed too.

Additional arguments:

Parameters
block – Block that owns this symbol (default: None)

Raises
ValueError –

set(*val*)

Write the symbol in TwinCAT.

Simply an alias for `write()`.

3.2 GUI

3.2.1 TwinCAT UI Elements

Here we define all the non-implemented TwinCAT stuff, without any platform.

The *TcElement* class should be extended by a class of another platform, ideally through multiple inheritance.

TcElement

class twinpy.element.**TcElement**(**args*, ***kwargs*)

Bases: ABC

Abstract class for TwinCAT element.

Must be inherited into a new class for something meaningful.

There are different event types, which determine how and when new remote values are retrieved:

- *EVENT_NOTIFICATION*: An ADS notification is created, resulting in a callback on a remote value change. Suitable for rarely changing values or when a very quick response is needed. ADS notifications have some overhead. No more than 200 symbol notifications should exist at the same time.
- *EVENT_TIMER*: New values are read at a fixed interval. Useful when remote values change often but no instant response is needed. This method has very little overhead.
- *EVENT_NONE*: No attempts are made to update according to remote values.

You can override the defaults for your entire project by changing the `TcWidget.DEFAULT_EVENT_TYPE` and `TcWidget.DEFAULT_UPDATE_FREQ` class properties. Make sure to update these before any widgets are created and they will have new standard values.

Variables

- **DEFAULT_EVENT_TYPE** – (default: EVENT_NOTIFICATION)
- **DEFAULT_UPDATE_FREQ** – (default: 10.0 Hz)

Note: these methods do not affect when or how a value is _written_ to the ADS pool.

Parameters

- **args** –
- **kwargs** – See list below - kwargs are passed along to *connect_symbol* too

Kwargs

- ***symbol*: Symbol to link to**
  (i.e. to read from and/or write to)
- ***format*: Formatter symbol, e.g. '%.1f' or '%d' or callable**
  ('%.3f' by default, ignored when not relevant) Callable must have a single argument
- ***event_type*: Possible values are EVENT_* constants**
  (default: DEFAULT_EVENT_TYPE)
- ***update_freq*: Frequency (Hz) for timed update**
  (for EVENT_TIMER only, default: DEFAULT_UPDATE_FREQ)
- ***greyed*: When true, the widget is visibly disabled**
  When false, the widget is shown normally even when disconnected (default: *true*)

DEFAULT_EVENT_TYPE = 'notification'

DEFAULT_UPDATE_FREQ = 10.0

EVENT_NONE = 'none'

EVENT_NOTIFICATION = 'notification'

EVENT_TIMER = 'timer'

connect_symbol(*new_symbol: Optional[Symbol] = None*, ***kwargs*) → bool

Connect a symbol (copy of symbol is left as property).

By default a device callback is created with an on-change event from TwinCAT. Old callbacks are deleted first. Pass None to only clear callbacks. The notification handles are stored locally. Extend (= override but call the parent first) this method to configure more of the widget, useful if e.g. widget callbacks depend on the symbol.

Parameters

- **new_symbol** – Symbol to link to (set None to only clear the previous)
- **kwargs** – See list below - Keyword arguments are passed along as device notification settings too

Returns

True if new symbol was connected

Kwargs

- *event_type*: See *TcElement*
- *update_freq*: See *TcElement*

format(*value: Any*) → str

"Use the stored formatting to created a formatted text.

In case the format specifier is a string and the new value is a list, element-wise string formatting will be concatenated automatically.

static make_timer(*plc*, *interval*)

Create timer instance - to be extended by a different implementation

on_mass_timeout()

Callback for the event timer.

This assumes the remote read was already performed!

abstract twincat_receive(*value*)

Callback attached to the TwinCAT symbol.

Note: changing a state of a widget (e.g. checkbox being checked through *setChecked(True)*) will typically fire the on-change events again. So be careful to prevent an event loop when updating a widget based on a remote change: a change could result in a state change, which could result in a remote change, etc.

Parameters

value – New remote value

twincat_receive_wrapper(*value*)

Intermediate twincat_receive callback to prevent event loops.

twincat_send(*value: Any*)

Set value in symbol (and send to TwinCAT).

Method is safe: if symbol is not connected, nothing will happen.

3.2.2 Qt Widgets

TwinCAT widgets are Qt elements that are easily linked to an ADS symbol.

E.g. a label that shows an output value or an input box which changes a parameter.

The *@pyqtSlot()* is Qt decorator. In many cases it is not essential, but it's good practice to add it anyway.

TcWidget

class twinpy.ui.**TcWidget**(**args*, ***kwargs*)

Bases: *TcElement*

Abstract class, to be multi-inherited together with a Qt item.

It is important to call this init() as late as possible from a subclass! The order should be:

1. Subclass specific stuff (e.g. number formatting)
2. Call to super().__init__(...) - Now the QWidget stuff has been made ready
3. QWidget related stuff

Parameters

- **args** –
- **kwargs** – See `TcElement.__init__()`

static close_with_error(*err: Exception*)

Show an error popup and close the active application.

This uses *QApplication.instance()* to find the current application and won't work perfectly.

connect_symbol(*new_symbol: Optional[Symbol] = None*, ***kwargs*) → bool

Connect a symbol (copy is left as property).

Parameters

- **new_symbol** – Symbol to link to (set None to only clear the previous)
- **kwargs** – See `TcElement.connect_symbol()`

static make_timer(*plc*, *interval*) → TcTimer

Use QTimer based timer instead.

twincat_send(*value: Any*)

Set value in symbol (and send to TwinCAT).

Method is safe: if symbol is not connected, nothing will happen.

value_format: Union[str, Callable[[Any], str]]

TcLabel

class twinpy.ui.**TcLabel**(**args*, ***kwargs*)

Bases: QLabel, *TcWidget*

Label that shows a value.

Parameters

- **args** –
- **kwargs** – See *TcWidget*

twincat_receive(*value*)

Callback attached to the TwinCAT symbol.

Note: changing a state of a widget (e.g. checkbox being checked through *setChecked(True)*) will typically fire the on-change events again. So be careful to prevent an event loop when updating a widget based on a remote change: a change could result in a state change, which could result in a remote change, etc.

Parameters
value – New remote value

value_format: Union[str, Callable[[Any], str]]

TcLineEdit

class twinpy.ui.**TcLineEdit**(**args*, ***kwargs*)

Bases: QLineEdit, *TcWidget*

Readable and writable input box.

Parameters

- **args** –
- **kwargs** – See *TcWidget*

on_editing_finished()

Called when [Enter] is pressed or box loses focus.

on_text_edited(**_value*)

Callback when text was modified (i.e. on key press).

twincat_receive(*value*) → Any

Callback attached to the TwinCAT symbol.

Note: changing a state of a widget (e.g. checkbox being checked through *setChecked(True)*) will typically fire the on-change events again. So be careful to prevent an event loop when updating a widget based on a remote change: a change could result in a state change, which could result in a remote change, etc.

Parameters
value – New remote value

value_format: Union[str, Callable[[Any], str]]

TcPushButton

class twinpy.ui.**TcPushButton**(**args*, ***kwargs*)

Bases: QPushButton, *TcWidget*

Button that sends value when button is held pressed.

Parameters

- **args** –
- **kwargs** –

Kwargs

- *value_pressed*: Value on press (default: 1), None for no action
- *value_released*: Value on release (default: 0), None for no action
- See *TcWidget*

on_pressed()

Callback on pressing button.

on_released()

Callback on releasing button.

twincat_receive(*value*)

Do nothing, method requires definition anyway.

value_format: Union[str, Callable[[Any], str]]

TcRadioButton

class twinpy.ui.**TcRadioButton**(**args*, ***kwargs*)

Bases: QRadioButton, *TcWidget*

Radiobutton that updates the symbol when it is selected.

The radiobutton will _not_ update the symbol when another selection is made. Instead a write could be performed if that other radio is also a TcWidget.

Use *TcRadioButtonGroupBox* instead to create a set of radio buttons together that all update the same ADS symbol.

When connecting to a boolean symbol, use 0 and 1 as values for the best result instead of *True* and *False*.

Radios need to be in a QButtonGroup together to link together.

Parameters

- **label** (*str*) – Label of this radio button
- **args** –
- **kwargs** –

Kwargs

- *value_checked*: Value when radio becomes checked (default: 1)
- See *TcWidget*

on_toggled()

Callback when radio state is togged (either checked or unchecked).

twincat_receive(*value*)

Set checked state if the new value is equal to the is-checked value.

value_format: Union[str, Callable[[Any], str]]

TcRadioButtonGroupBox

class twinpy.ui.**TcRadioButtonGroupBox**(**args*, ***kwargs*)

Bases: QGroupBox, *TcWidget*

An instance on this class forms a group of radio buttons.

The group of radio buttons together control a single ADS variable.

Instances of *QRadioButton* will be automatically created through this class. Using literal instances of *TcRadioButton* is not efficient because of duplicate callbacks.

When the remote value changes to a value that is not listed as an option in the radio_toggle, the displayed value simply won't change at all.

The *options* argument is required.

Parameters

- **title** (*str*) – Title of this QGroupBox
- **args** –
- **kwargs** –

Kwargs

- ***options*: List of tuples that form the label-value pairs of the radio,** e.g. *[('Low Velocity', 0.5), ('High Velocity', 3.0)]*
- ***layout_class*: Class of the layout used inside the QGroupBox (default:** *QVBoxLayout*)
- See *TcWidget*

on_click(*button: QAbstractButton*)

Callback when a button of the group was pressed.

twincat_receive(*value*)

Callback for a remote value change.

value_format: Union[str, Callable[[Any], str]]

TcCheckBox

class twinpy.ui.**TcCheckBox**(**args*, ***kwargs*)

Bases: QCheckBox, *TcWidget*

Checkbox to control a symbol.

Set either value to *None* to send nothing on that state. For the best results, use 1 and 0 for a boolean variable instead of *True* and *False*.

Parameters

- **label** (*str*) – Label of this radio button
- **args** –
- **kwargs** –

Kwargs

- *value_checked*: Value when checkbox becomes checked (default: 1)
- *value_unchecked*: Value when checkbox becomes unchecked (default: 0)
- See *TcWidget*

on_toggled()

Callback when box state is togged (either checked or unchecked).

twincat_receive(*value*)

Set checked state if the new value is equal to the is-checked value.

value_format: Union[str, Callable[[Any], str]]

TcSlider

class twinpy.ui.**TcSlider**(**args*, ***kwargs*)

Bases: QWidget, *TcWidget*

Interactive slider.

Also has built-in slider numbers (unlike the basic QSlider). This class extends a plain widget so a layout can be added for any labels.

The basic QSlider only supports integer values. To support floating point numbers too, the slider values are multiplied by a scale (e.g. 100) when writing, and divided again when reading from the slider. Use this with the *float* and *float_scale* options. This is done automatically if *interval* is not an integer.

Variables
slider – QSlider instance

Parameters

- **orientation** – Either *QtCore.Qt.Horizontal* (default) or *Vertical*
- **args** –
- **kwargs** –

Kwargs

- *min*: Slider minimum value (default: 0)
- *max*: Slider maximum value (default: 100)

- *interval*: Slider interval step size (default: 1)
- ***show_labels*: When true (default), show the min and max values with** labels
- ***show_value*: When true (default), show the current slider value with a** label
- ***float*: When true, QSlider values are scaled to suit floats (default:** False)
- ***float_scale*: Factor between QSlider values and real values (default:**

100)

- See *TcWidget*

on_value_changed(*new_value*)

Callback when the slider was changed by the user.

slider_to_value(*value: int*) → Union[float, int]

twincat_receive(*value*) → Any

On remote value change.

This will be triggered by *on_value_changed* too. A small timeout is added to prevent a loop between the two callbacks, received changes right after a user change are ignored.

value_format: Union[str, Callable[[Any], str]]

value_to_slider(*value: float*) → Union[int, float]

TcGraph

class twinpy.ui.**TcGraph**(**args*, ***kwargs*)

Bases: *GraphWidget*, *TcWidget*

Draw rolling graph of symbol values.

TcGraph works only well with *EVENT_TIMER*!

The graph refresh rate is limited to self.FPS, while data is being requested at *update_freq*. For research measurements, use a log file or a TwinCAT measurement project instead. Even with a high *update_freq* there is no guarantee all data is captured!

If no symbol for the x-axis is selected, the local time will be used instead. Note that due to how PyQt events are handled, the local time can be slightly warped with respect the ADS symbol values.

See *GraphWidget* for more options.

Parameters

- **args** –
- **kwargs** –

Kwargs

- *symbols*: List of symbols to plot (for the y-axis)
- *symbol_x*: Symbol to use on the x-axis (optional)

connect_symbol(*new_symbol: Optional[Union[Symbol, List[Symbol]]] = None*, ***kwargs*)

Connect to list of symbols (override).

on_mass_timeout()

Callback for the event timer (override).

This assumes the remote read was already performed!

twincat_receive(*value*)

Abstract implementation.

All useful code is in on_mass_timeout() instead.

value_format: Union[str, Callable[[Any], str]]

class twinpy.ui.**GraphWidget**(*labels: List[str]*, *units: Optional[List[str]] = None*, *buffer_size: int = 100*, *values_in_legend: bool = True*, **args*, ***kwargs*)

Bases: QWidget

Class to make an rolling plot of some data.

When data comes in faster than FS, the plot won't be refreshed. The data will still be stored and show up when it's time.

Variables

FPS – Maximum refresh rate (Hz), faster samples are buffered but not yet plotted.

Parameters

- **labels** – List of strings corresponding to plotted variable names
- **units** – List of display units for each variable (default: None)
- **buffer_size** – Number of points to keep in graph
- **values_in_legend** – When *True* (default), put the last values in the legend

FPS = 30.0

add_data(*y_list: List[float]*, *x: Optional[float] = None*)

Add new datapoint to the rolling graph.

Parameters

- **y_list** – List of new y-values
- **x** – New x-value (default: use system time instead)

update_plot()

Refresh the plot based on *self.data*.

3.2.3 Base GUI

Module containing the Base GUI class.

This class should be extended to make your own GUI.

class twinpy.ui.base_gui.**BaseGUI**(*actuator: Optional[SimulinkModel] = None*, *controller: Optional[SimulinkModel] = None*, ***kwargs*)

Bases: TcMainWindow

Base TwinCAT GUI, specific for the WE2 actuators and controller model.

Extends this class if you want to use those models. For other models, using `twinpy.ui.TcMainWindow` might be more appropriate.

The left side of the main window consists of the basic control elements. The right contains a tabs widget, to which you can add your own custom tabs.

An example of an extended GUI could be:

```
class CustomTab(QWidget):
    # Widget containing a set of TcWidgets

class MyGUI(BaseGUI):

    def __init__():
        super().__init__()

        self.custom_tab = CustomTab()
        # Add new tab to the existing list:
        self.tabs.addTab(custom_tab, "Custom Tab")

        # Make the custom tab the default:
        self.tabs.setCurrentWidget(self.custom_tab)
```

Note: if the *pyqtconsole* package is not found, the console won't be created and the tabs list could be empty.

Parameters

- **actuator** – The WE2_actuators model (or a derivative)
- **controller** – The WE2_controller model (or a derivative)
- **kwargs** –

closeEvent(*event*)

Callback when window is closed.

3.2.4 Base Widgets

These widgets are specific implementations of TcWidgets.

They are not intended to be overridden again. They are separate classes mostly because their specific logic became significant.

TcErrorsLabel

class twinpy.ui.**TcErrorsLabel**(**args*, ***kwargs*)

Bases: *TcLabel*

Extension of TcLabel for a list of joint errors.

This is separate class because the amount of logic got a little bit much.

When clicking on the widget, a new window pops up showing the decoded errors.

Parameters

- **args** –
- **kwargs** –

Kwargs

- ***format*: Callback to format errors**
  (default: show hexadecimal representation of error values)
- ***popup*: Whether or not to enable a detailed popup window**
  (default: True)
- ***play_sound*: When True, play a beep sound on a new error**
  (default: False)
- See *TcLabel*

static format_errors_list(*error_list: List[int]*) → str

Set text for errors label.

mousePressEvent(*event: QMouseEvent*) → None

On clicking on the label.

QLabel does not have an on-click signal already.

static to_hex(*value: int*) → str

Create human-readable hex from integer.

twincat_receive(*value*)

Callback on remote value change.

value_format: Union[str, Callable[[Any], str]]

DrivesWidget

class twinpy.ui.**DrivesWidget**(*actuator: Optional[SimulinkModel] = None*)

Bases: QGroupBox

Group with buttons for the drives.

on_drives_enabled_change(*enabled_list: List[float]*)

An additional callback for the drive state change.

Manual callback instead of TcWidget symbol connection so we can also change button state and label color.

SystemBackpackWidget

class twinpy.ui.**SystemBackpackWidget**(*actuator: Optional[SimulinkModel] = None*)

Bases: QGroupBox

Widget containing labels for the temperature and voltages.

This widget is for the old backpack system.

SystemWRBSWidget

class twinpy.ui.**SystemWRBSWidget**(*actuator: Optional[SimulinkModel] = None*)

Bases: QGroupBox

Widget containing labels for the temperature and voltages.

This widget is for the new wearable robotics base station.

ErrorsWidget

class twinpy.ui.**ErrorsWidget**(*actuator: Optional[SimulinkModel] = None*)

Bases: QWidget

Widget for the current and last errors.

Layout contains two groupboxes, and each can be clicked for a popup with more info.

You might want to call the *close_windows()* method from inside the *closeEvent()* function from the main window, to close the popups when closing the GUI.

close_windows()

Close the popup windows in case they were opened.

3.2.5 Tabs

Module containing complete UI tabs.

ConsoleTab

class twinpy.ui.**ConsoleTab**(*actuator: Optional[SimulinkModel] = None*, *controller: Optional[SimulinkModel] = None*)

Bases: PythonConsole

Tab with the Python console.

The *clear()* command is available in the console.

FirmwareTab

class twinpy.ui.**FirmwareTab**(*actuator: Optional[SimulinkModel] = None*)

Bases: QWidget

Widget for the firmware CtrlWord options (Base Station only).

3.3 Main - Example

Example of a script using a GUI.

You can run the BaseGUI itself (linked to the WE actuator model) by running

$ python -m twinpy

(I.e. by executing the module.) This provides a quick demo of how such a GUI can work, and a quick check to see everything is working on your system.

This script should not be used in your program! Instead, your application should have it's own main script and create a GUI instance from there.

class twinpy.__main__.**GUI**(*actuator: Optional[SimulinkModel] = None*, *controller: Optional[SimulinkModel] = None*, ***kwargs*)

Bases: *BaseGUI*

Some extension of the BaseGUI.

Custom GUIs should extend the base class.

Parameters

- **actuator** – The WE2_actuators model (or a derivative)
- **controller** – The WE2_controller model (or a derivative)
- **kwargs** –

3.4 Module Info

TwinPy package.

author
Robert Roos <robert.soor@gmail.com>

license
MIT, see license file or https://opensource.org/licenses/MIT

created on
2021-01-08 16:13:00

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

O

P

R

S

T

U

V

W