TwinPy

Robert Roos

Apr 25, 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

MODULES

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: pyads.connection.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*) → *twinpy.twincat.symbols.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*) → *twinpy.twincat.symbols.Signal*

Get Signal instance.

See `Signal`.

read_list_of_symbols(*symbols: List[pyads.symbol.AdsSymbol]*, *ads_sub_commands: int = 500*) → Dict[pyads.symbol.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[pyads.symbol.AdsSymbol, Any]*, *ads_sub_commands: int = 500*) → Dict[pyads.symbol.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: *twinpy.twincat.simulink.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:* twinpy.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: xml.etree.ElementTree.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[*twinpy.twincat.connection.TwincatConnection*]

Return Connection (owned by model).

static get_xml_data(*type_name: str*) → xml.etree.ElementTree.Element

Find and parse model XML file.

The block diagram is returned.

SimulinkBlock

class twinpy.twincat.**SimulinkBlock**(*xmltree: xml.etree.ElementTree.Element*, *model:* twinpy.twincat.simulink.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[*twinpy.twincat.connection.TwincatConnection*]

Return Connection (owned by model).

get_symbols_recursive() → List[*twinpy.twincat.symbols.Symbol*]

Recursively navigate subblocks and collect all parameters and signals.

make_parameters(*xmltree: xml.etree.ElementTree.Element*) → dict

Find and create Parameters in the current block.

make_signals(*xmltree: xml.etree.ElementTree.Element*) → dict

Find and create Signals in the current block.

make_subblocks(*xmltree: xml.etree.ElementTree.Element*) → Dict[str, *twinpy.twincat.simulink.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: `pyads.symbol.AdsSymbol`, `abc.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[pyads.structs.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[pyads.connection.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: *twinpy.twincat.symbols.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: *twinpy.twincat.symbols.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

TcWidget

TcLabel

TcLineEdit

TcPushButton

TcRadioButton

TcRadioButtonGroupBox

TcCheckBox

TcSlider

TcGraph

3.2.2 Base GUI

3.2.3 Base Widgets

TcErrorsLabel

DrivesWidget

SystemBackpackWidget

SystemWRBSWidget

ErrorsWidget

3.2.4 Tabs

ConsoleTab

FirmwareTab

3.3 Main - Example

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

W