
HVL Common Code Base Documentation

Release 0.10.1

Mikolaj Rybiński, David Graber, Henrik Menne, Alise Chachereau,

Jan 24, 2022

CONTENTS:

1	HVL Common Code Base	1
1.1	Features	1
1.2	Documentation	5
1.3	Credits	5
2	Installation	7
2.1	Stable release	7
2.2	From sources	7
2.3	Additional system libraries	8
3	Usage	9
4	API Documentation	11
4.1	hvl_ccb	11
5	Contributing	183
5.1	Types of Contributions	183
5.2	Get Started!	184
5.3	Merge Request Guidelines	185
5.4	Tips	185
5.5	Deploying	186
6	Credits	187
6.1	Maintainers	187
6.2	Authors	187
6.3	Contributors	187
7	History	189
7.1	0.10.1 (2022-01-24)	189
7.2	0.10.0 (2022-01-17)	189
7.3	0.9.0 (2022-01-07)	190
7.4	0.8.5 (2021-11-05)	190
7.5	0.8.4 (2021-10-22)	191
7.6	0.8.3 (2021-09-27)	191
7.7	0.8.2 (2021-08-27)	191
7.8	0.8.1 (2021-08-13)	191
7.9	0.8.0 (2021-07-02)	191
7.10	0.7.1 (2021-06-04)	192
7.11	0.7.0 (2021-05-25)	192
7.12	0.6.1 (2021-05-08)	192
7.13	0.6.0 (2021-04-23)	193

7.14	0.5.0 (2020-11-11)	193
7.15	0.4.0 (2020-07-16)	194
7.16	0.3.5 (2020-02-18)	194
7.17	0.3.4 (2019-12-20)	195
7.18	0.3.3 (2019-05-08)	195
7.19	0.3.2 (2019-05-08)	195
7.20	0.3.1 (2019-05-02)	195
7.21	0.3 (2019-05-02)	195
7.22	0.2.1 (2019-04-01)	196
7.23	0.2.0 (2019-03-31)	196
7.24	0.1.0 (2019-02-06)	196
8	Indices and tables	197
	Python Module Index	199
	Index	201

HVL COMMON CODE BASE

Python common code base (CCB) to control devices, which are used in high-voltage research. All implemented devices are used and tested in the High Voltage Laboratory ([HVL](#)) of the Federal Institute of Technology Zurich (ETH Zurich).

- Free software: GNU General Public License v3
- Copyright (c) 2019-2022 ETH Zurich, SIS ID and HVL D-ITET

1.1 Features

For managing multi-device experiments instantiate the `ExperimentManager` utility class.

1.1.1 Devices

The device wrappers in `hvl_ccb` provide a standardised API with configuration dataclasses, various settings and options, as well as start/stop methods. Currently wrappers are available to control the following devices:

Function/Type	Devices
Data acquisition	LabJack (T4, T7, T7-PRO; requires LJM Library) Pico Technology PT-104 Platinum Resistance Data Logger (requires PicoSDK/libusbpt104)
Digital Delay Generator	Highland T560
Digital IO	LabJack (T4, T7, T7-PRO; requires LJM Library)
Experiment control	HVL Cube with and without Power Inverter
Gas Analyser	MBW 973-SF6 gas dew point mirror analyzer Pfeiffer Vacuum TPG (25x, 26x and 36x) controller for compact pressure gauges SST Luminox oxygen sensor
I2C host	TiePie (HS5, WS5; requires LibTiePie SDK)
Laser	CryLaS pulsed laser CryLaS laser attenuator
Oscilloscope	Rhode & Schwarz RTO 1024 TiePie (HS5, HS6, WS5; requires LibTiePie SDK)
Power supply	Elektro-Automatik PSI9000 FuG Elektronik Heinzinger PNC Technix capacitor charger
Stepper motor drive	Newport SMC100PP Schneider Electric ILS2T
Temperature control	Lauda PRO RP 245 E circulation thermostat
Waveform generator	TiePie (HS5, WS5; requires LibTiePie SDK)
2	Chapter 1. HVL Common Code Base

Each device uses at least one standardised communication protocol wrapper.

1.1.2 Communication protocols

In `hvl_ccb` by “communication protocol” we mean different levels of communication standards, from the low level actual communication protocols like serial communication to application level interfaces like VISA TCP standard. There are also devices in `hvl_ccb` that use a dummy communication protocol; this is because these devices are build on proprietary manufacturer libraries that communicate with the corresponding devices, as in the case of TiePie or LabJack devices.

The communication protocol wrappers in `hvl_ccb` provide a standardised API with configuration dataclasses, as well as open/close and read/write/query methods. Currently, wrappers for the following communication protocols are available:

Communication protocol	Devices using
Modbus TCP	Schneider Electric ILS2T stepper motor drive
OPC UA	HVL Cube with and without Power Inverter
Serial	<p>CryLaS pulsed laser and laser attenuator</p> <p>FuG Elektronik power supply (e.g. capacitor charger HCK) using the Probus V protocol</p> <p>Heinzinger PNC power supply using Heinzinger Digital Interface I/II</p> <p>SST Luminos oxygen sensor</p> <p>MBW 973-SF6 gas dew point mirror analyzer</p> <p>Newport SMC100PP single axis driver for 2-phase stepper motors</p> <p>Pfeiffer Vacuum TPG (25x, 26x and 36x) controller for compact pressure gauges</p> <p>Technix capacitor charger</p>
TCP	Lauda PRO RP 245 E circulation thermostat
Telnet	Technix capacitor charger
VISA TCP	<p>Elektro-Automatik PSI9000 DC power supply</p> <p>Rhode & Schwarz RTO 1024 oscilloscope</p>
<i>propriety</i>	<p>LabJack (T4, T7, T7-PRO) devices, which communicate via LJM Library</p> <p>Pico Technology PT-104 Platinum Resistance Data Logger, which communicate via PicoSDK/libusbpt104</p> <p>TiePie (HS5, HS6, WS5) oscilloscopes, generators and I2C hosts, which communicate via LibTiePie SDK</p>

1.1.3 Sensor and Unit Conversion Utility

The Sensor and Unit Conversion Utility is a submodule that allows on the one hand a unified implementation of hardware-sensors and on the other hand provides a unified way to convert units. The utility can be used with single numbers (`int`, `float`) as well as array-like structures containing single numbers (`np.array()`, `list`, `dict`, `tuple`).

Currently the following sensors are implemented:

- LEM LT 4000S
- LMT 70A

The following unit conversion classes are implemented:

- Temperature (Kelvin, Celsius, Fahrenheit)
- Pressure (Pascal, Bar, Atmosphere, Psi, Torr, Millimeter Mercury)

1.2 Documentation

Note: if you're planning to contribute to the `hvl_ccb` project read the **Contributing** section in the HVL CCB documentation.

Do either:

- read [HVL CCB documentation at RTD](#),

or

- build and read HVL CCB documentation locally; install first [Graphviz](#) (make sure to have the `dot` command in the executable search path) and the Python build requirements for documentation:

```
$ pip install docs/requirements.txt
```

and then either on Windows in Git BASH run:

```
$ ./make.sh docs
```

or from any other shell with GNU Make installed run:

```
$ make docs
```

The target index HTML ("`docs/_build/html/index.html`") should open automatically in your Web browser.

1.3 Credits

This package was created with [Cookiecutter](#) and the [audreyr/cookiecutter-pypackage](#) project template.

INSTALLATION

2.1 Stable release

To install HVL Common Code Base, run this command in your terminal:

```
$ pip install hvl_ccb
```

To install HVL Common Code Base with optional Python libraries that require manual installations of additional system libraries, you need to specify on installation extra requirements corresponding to these controllers. For instance, to install Python requirements for LabJack and TiePie devices, run:

```
$ pip install "hvl_ccb[tiepie,labjack]"
```

See below for the info about additional system libraries and the corresponding extra requirements.

To install all extra requirements run:

```
$ pip install "hvl_ccb[all]"
```

This is the preferred method to install HVL Common Code Base, as it will always install the most recent stable release. If you don't have [pip](#) installed, this [Python installation guide](#) can guide you through the process.

2.2 From sources

The sources for HVL Common Code Base can be downloaded from the [GitLab repo](#).

You can either clone the repository:

```
$ git clone git@gitlab.com:ethz_hvl/hvl_ccb.git
```

Or download the [tarball](#):

```
$ curl -OL https://gitlab.com/ethz_hvl/hvl_ccb/-/archive/master/hvl_ccb.tar.gz
```

Once you have a copy of the source, you can install it with:

```
$ pip install .
```

2.3 Additional system libraries

If you have installed *hvl_ccb* with any of the extra features corresponding to device controllers, you must additionally install respective system library; these are:

Extra feature	Additional system library
labjack	LJM Library
picotech	PicoSDK (Windows) / libusbpt104 (Ubuntu/Debian)
tiepie	LibTiePie SDK

For more details on installation of the libraries see docstrings of the corresponding *hvl_ccb* modules.

CHAPTER THREE

USAGE

To use HVL Common Code Base in a project:

```
import hvl_ccb
```


API DOCUMENTATION

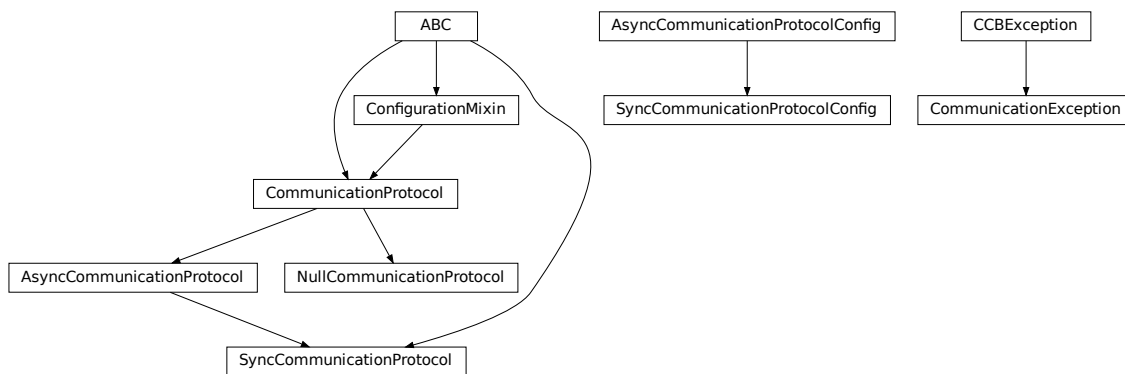
4.1 hvl_ccb

4.1.1 Subpackages

`hvl_ccb.comm`

Submodules

`hvl_ccb.comm.base`



Module with base classes for communication protocols.

class `AsyncCommunicationProtocol`(*config*)

Bases: `hvl_ccb.comm.base.CommunicationProtocol`

Abstract base class for asynchronous communication protocols

static `config_cls()` → `Type[hvl_ccb.comm.base.AsyncCommunicationProtocolConfig]`

Return the default configdataclass class.

Returns a reference to the default configdataclass class

read() → `str`

Read a single line of text as *str* from the communication.

Returns text as *str* including the terminator, which can also be empty “”

read_all(*n_attempts_max*: *Optional[int] = None*, *attempt_interval_sec*: *Optional[Union[int, float]] = None*) → *Optional[str]*

Read all lines of text from the connection till nothing is left to read.

Parameters

- **n_attempts_max** – Amount of attempts how often a non-empty text is tried to be read
- **attempt_interval_sec** – time between the reading attempts

Returns A multi-line *str* including the terminator internally

abstract read_bytes() → *bytes*

Read a single line as *bytes* from the communication.

This method uses *self.access_lock* to ensure thread-safety.

Returns a single line as *bytes* containing the terminator, which can also be empty b''

read_nonempty(*n_attempts_max*: *Optional[int] = None*, *attempt_interval_sec*: *Optional[Union[int, float]] = None*) → *Optional[str]*

Try to read a non-empty single line of text as *str* from the communication. If the host does not reply or reply with white space only, it will return *None*.

Returns a non-empty text as a *str* or *None* in case of an empty string

Parameters

- **n_attempts_max** – Amount of attempts how often a non-empty text is tried to be read
- **attempt_interval_sec** – time between the reading attempts

read_text() → *str*

Read one line of text from the serial port. The input buffer may hold additional data afterwards, since only one line is read.

NOTE: backward-compatibility proxy for *read* method; to be removed in v1.0

Returns String read from the serial port; '' if there was nothing to read.

Raises *SerialCommunicationIOError* – when communication port is not opened

read_text_nonempty(*n_attempts_max*: *Optional[int] = None*, *attempt_interval_sec*: *Optional[Union[int, float]] = None*) → *Optional[str]*

Reads from the serial port, until a non-empty line is found, or the number of attempts is exceeded.

NOTE: backward-compatibility proxy for *read* method; to be removed in v1.0

Attention: in contrast to *read_text*, the returned answer will be stripped of a whitespace newline terminator at the end, if such terminator is set in the initial configuration (default).

Parameters

- **n_attempts_max** – maximum number of read attempts
- **attempt_interval_sec** – time between the reading attempts

Returns String read from the serial port; '' if number of attempts is exceeded or serial port is not opened.

write(*text*: *str*)

Write text as *str* to the communication.

Parameters **text** – test as a *str* to be written

abstract write_bytes(*data: bytes*) → int

Write data as *bytes* to the communication.

This method uses *self.access_lock* to ensure thread-safety.

Parameters **data** – data as *bytes*-string to be written

Returns number of bytes written

write_text(*text: str*)

Write text to the serial port. The text is encoded and terminated by the configured terminator.

NOTE: backward-compatibility proxy for *read* method; to be removed in v1.0

Parameters **text** – Text to send to the port.

Raises *SerialCommunicationIOError* – when communication port is not opened

```
class AsyncCommunicationProtocolConfig(terminator: bytes = b'\r\n', encoding: str = 'utf-8',
                                       encoding_error_handling: str = 'strict',
                                       wait_sec_read_text_nonempty: Union[int, float] = 0.5,
                                       default_n_attempts_read_text_nonempty: int = 10)
```

Bases: object

Base configuration data class for asynchronous communication protocols

clean_values()

default_n_attempts_read_text_nonempty: int = 10

default number of attempts to read a non-empty text

encoding: str = 'utf-8'

Standard encoding of the connection. Typically this is *utf-8*, but can also be *latin-1* or something from here: <https://docs.python.org/3/library/codecs.html#standard-encodings>

encoding_error_handling: str = 'strict'

Encoding error handling scheme as defined here: <https://docs.python.org/3/library/codecs.html#error-handlers> By default strict error handling that raises *UnicodeError*.

force_value(*fieldname, value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

is_configdataclass = True

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

classmethod `required_keys()` → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

terminator: bytes = b'\r\n'

The terminator character. Typically this is b'\r\n' or b'\n', but can also be b'\r' or other combinations. This defines the end of a single line.

wait_sec_read_text_nonempty: Union[int, float] = 0.5

time to wait between attempts of reading a non-empty text

exception `CommunicationException`

Bases: `hvl_ccb.exception.CCBException`

class `CommunicationProtocol(config)`

Bases: `hvl_ccb.configuration.ConfigurationMixin`, `abc.ABC`

Communication protocol abstract base class.

Specifies the methods to implement for communication protocol, as well as implements some default settings and checks.

access_lock

Access lock to use with context manager when accessing the communication protocol (thread safety)

abstract `close()`

Close the communication protocol

abstract `open()`

Open communication protocol

class `NullCommunicationProtocol(config)`

Bases: `hvl_ccb.comm.base.CommunicationProtocol`

Communication protocol that does nothing.

close() → None

Void close function.

static `config_cls()` → Type[`hvl_ccb.configuration.EmptyConfig`]

Empty configuration

Returns EmptyConfig

open() → None

Void open function.

class `SyncCommunicationProtocol(config)`

Bases: `hvl_ccb.comm.base.AsyncCommunicationProtocol`, `abc.ABC`

Abstract base class for synchronous communication protocols with `query()`

static `config_cls()` → Type[`hvl_ccb.comm.base.SyncCommunicationProtocolConfig`]

Return the default configdataclass class.

Returns a reference to the default configdataclass class

query(*command: str*) → Optional[str]

Send a command to the interface and handle the status message. Eventually raises an exception.

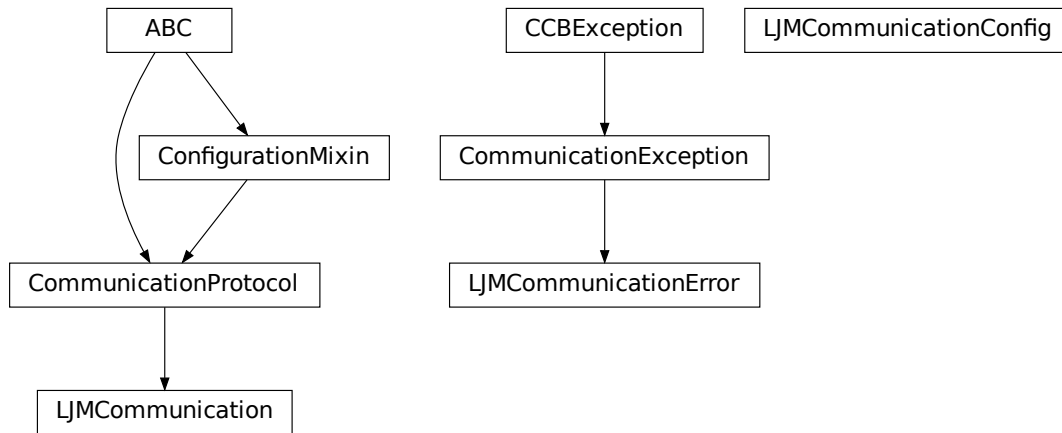
Parameters `command` – Command to send

Returns Answer from the interface, which can be None instead of an empty reply

```
class SyncCommunicationProtocolConfig(terminator: bytes = b'\r\n', encoding: str = 'utf-8',
                                     encoding_error_handling: str = 'strict',
                                     wait_sec_read_text_nonempty: Union[int, float] = 0.5,
                                     default_n_attempts_read_text_nonempty: int = 10)
```

Bases: `hvl_ccb.comm.base.AsyncCommunicationProtocolConfig`

`hvl_ccb.comm.labjack_ljm`



Communication protocol for LabJack using the LJM Library. Originally developed and tested for LabJack T7-PRO.

Makes use of the LabJack LJM Library Python wrapper. This wrapper needs an installation of the LJM Library for Windows, Mac OS X or Linux. Go to: <https://labjack.com/support/software/installers/ljm> and <https://labjack.com/support/software/examples/ljm/python>

```
class LJMCommunication(configuration)
```

Bases: `hvl_ccb.comm.base.CommunicationProtocol`

Communication protocol implementing the LabJack LJM Library Python wrapper.

close() → *None*

Close the communication port.

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

property is_open: **bool**

Flag indicating if the communication port is open.

Returns *True* if the port is open, otherwise *False*

open() → *None*

Open the communication port.

read_name(*names: str, return_num_type: typing.Type[numbers.Real] = <class 'float'>) → Union[numbers.Real, Sequence[numbers.Real]]

Read one or more input numeric values by name.

Parameters

- **names** – one or more names to read out from the LabJack
- **return_num_type** – optional numeric type specification for return values; by default *float*.

Returns answer of the LabJack, either single number or multiple numbers in a sequence, respectively, when one or multiple names to read were given

Raises **TypeError** – if read value of type not compatible with *return_num_type*

write_name(*name: str, value: numbers.Real*) → *None*

Write one value to a named output.

Parameters

- **name** – String or with name of LabJack IO
- **value** – is the value to write to the named IO port

write_names(*name_value_dict: Dict[str, numbers.Real]*) → *None*

Write more than one value at once to named outputs.

Parameters **name_value_dict** – is a dictionary with string names of LabJack IO as keys and corresponding numeric values

```
class LJMCommunicationConfig(device_type: Union[str, hvl_ccb._dev.labjack.DeviceType] = 'ANY',
                             connection_type: Union[str,
                             hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig.ConnectionType] =
                             'ANY', identifier: str = 'ANY')
```

Bases: object

Configuration dataclass for *LJMCommunication*.

```
class ConnectionType(value=<no_arg>, names=None, module=None, type=None, start=1,
                     boundary=None)
```

Bases: *hvl_ccb.utils.enum.AutoNumberNameEnum*

LabJack connection type.

ANY = 1

ETHERNET = 4

TCP = 3

USB = 2

WIFI = 5

```
class DeviceType(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)
```

Bases: *hvl_ccb.utils.enum.AutoNumberNameEnum*

LabJack device types.

Can be also looked up by ambiguous Product ID (*p_id*) or by instance name: ``python LabJackDeviceType(4) is LabJackDeviceType('T4')``

ANY = 1

T4 = 2

T7 = 3

T7_PRO = 4

classmethod `get_by_p_id(p_id: int) → Union[hvl_ccb._dev.labjack.DeviceType, List[hvl_ccb._dev.labjack.DeviceType]]`

Get LabJack device type instance via LabJack product ID.

Note: Product ID is not unambiguous for LabJack devices.

Parameters `p_id` – Product ID of a LabJack device

Returns Instance or list of instances of *LabJackDeviceType*

Raises **ValueError** – when Product ID is unknown

clean_values() → *None*

Performs value checks on `device_type` and `connection_type`.

connection_type: `Union[str, hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig.ConnectionType] = 'ANY'`

Can be either string or of enum *ConnectionType*.

device_type: `Union[str, hvl_ccb._dev.labjack.DeviceType] = 'ANY'`

Can be either string 'ANY', 'T7_PRO', 'T7', 'T4', or of enum *DeviceType*.

force_value(fieldname, value)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

identifier: `str = 'ANY'`

The identifier specifies information for the connection to be used. This can be an IP address, serial number, or device name. See the LabJack docs (<https://labjack.com/support/software/api/ljm/function-reference/ljmopens/identifier-parameter>) for more information.

is_configdataclass = True

classmethod `keys() → Sequence[str]`

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod `optional_defaults() → Dict[str, object]`

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

classmethod `required_keys() → Sequence[str]`

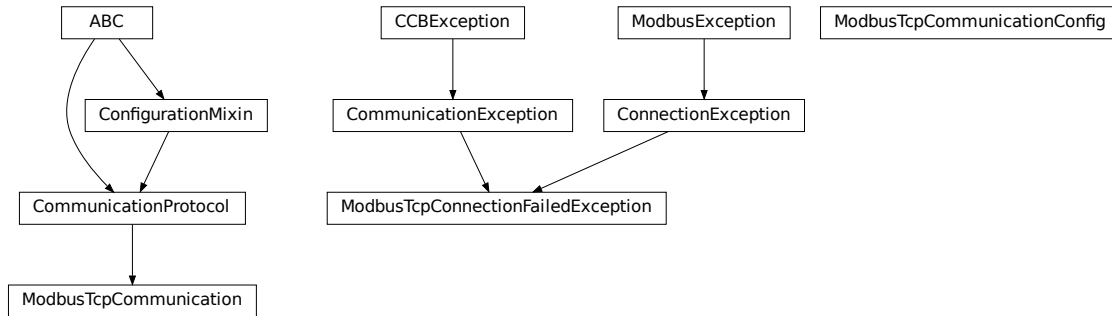
Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

exception **LJMCommunicationError**

Bases: *hvl_ccb.comm.base.CommunicationException*

Errors coming from LJMCommunication.

hvl_ccb.comm.modbus_tcp

Communication protocol for modbus TCP ports. Makes use of the [pymodbus](#) library.

class `ModbusTcpCommunication(configuration)`

Bases: `hvl_ccb.comm.base.CommunicationProtocol`

Implements the Communication Protocol for modbus TCP.

close()

Close the Modbus TCP connection.

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

open() → *None*

Open the Modbus TCP connection.

Raises `ModbusTcpConnectionFailedException` – if the connection fails.

read_holding_registers(*address: int, count: int*) → List[int]

Read specified number of register starting with given address and return the values from each register.

Parameters

- **address** – address of the first register
- **count** – count of registers to read

Returns list of *int* values

read_input_registers(*address: int, count: int*) → List[int]

Read specified number of register starting with given address and return the values from each register in a list.

Parameters

- **address** – address of the first register
- **count** – count of registers to read

Returns list of *int* values

write_registers(*address: int, values: Union[List[int], int]*)

Write values from the specified address forward.

Parameters

- **address** – address of the first register
- **values** – list with all values

```
class ModbusTcpCommunicationConfig(host: Union[str, ipaddress.IPv4Address, ipaddress.IPv6Address], unit:  
                                int, port: int = 502)
```

Bases: object

Configuration dataclass for [ModbusTcpCommunication](#).

clean_values()

force_value(*fieldname, value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

host: Union[str, ipaddress.IPv4Address, ipaddress.IPv6Address]

is_configdataclass = True

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

port: int = 502

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

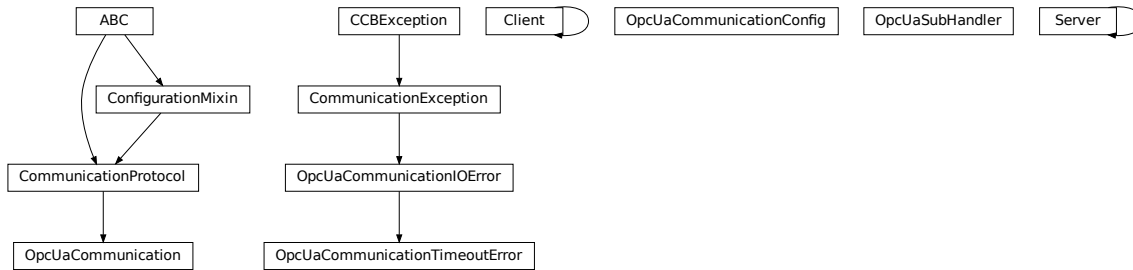
unit: int

```
exception ModbusTcpConnectionFailedException(string="")
```

Bases: pymodbus.exceptions.ConnectionException,
[CommunicationException](#)

[hvl_ccb.comm.base](#).

Exception raised when the connection failed.

hvl_ccb.comm.opc

Communication protocol implementing an OPC UA connection. This protocol is used to interface with the “Super-cube” PLC from Siemens.

```
class Client(url: str, timeout: int = 4)
```

Bases: `asyncua.sync.Client`

```
disconnect()
```

```
get_objects_node()
```

Get Objects node of client. Returns a Node object.

```
property is_open
```

```
send_hello(*args, **kwargs)
```

```
class OpcUaCommunication(config)
```

Bases: `hvl_ccb.comm.base.CommunicationProtocol`

Communication protocol implementing an OPC UA connection. Makes use of the package python-opcua.

```
close() → None
```

Close the connection to the OPC UA server.

```
static config_cls()
```

Return the default configdataclass class.

Returns a reference to the default configdataclass class

```
init_monitored_nodes(node_id: Union[object, Iterable], ns_index: int) → None
```

Initialize monitored nodes.

Parameters

- **node_id** – one or more strings of node IDs; node IDs are always casted via `str()` method here, hence do not have to be strictly string objects.
- **ns_index** – the namespace index the nodes belong to.

Raises `OpcUaCommunicationIOError` – when protocol was not opened or can’t communicate with a OPC UA server

```
property is_open: bool
```

Flag indicating if the communication port is open.

Returns `True` if the port is open, otherwise `False`

```
open() → None
```

Open the communication to the OPC UA server.

Raises *OpcUaCommunicationIOError* – when communication port cannot be opened.

read(*node_id*, *ns_index*)

Read a value from a node with id and namespace index.

Parameters

- **node_id** – the ID of the node to read the value from
- **ns_index** – the namespace index of the node

Returns the value of the node object.

Raises *OpcUaCommunicationIOError* – when protocol was not opened or can't communicate with a OPC UA server

write(*node_id*, *ns_index*, *value*) → *None*

Write a value to a node with name *name*.

Parameters

- **node_id** – the id of the node to write the value to.
- **ns_index** – the namespace index of the node.
- **value** – the value to write.

Raises *OpcUaCommunicationIOError* – when protocol was not opened or can't communicate with a OPC UA server

```
class OpcUaCommunicationConfig(host: typing.Union[str, ipaddress.IPv4Address, ipaddress.IPv6Address],
                               endpoint_name: str, port: int = 4840, sub_handler:
                               hvl_ccb.comm.opc.OpcUaSubHandler =
                               <hvl_ccb.comm.opc.OpcUaSubHandler object>, update_parameter:
                               asyncua.ua.uaproto.protocol_auto.CreateSubscriptionParameters =
                               CreateSubscriptionParameters(RequestedPublishingInterval=1000,
                                                             RequestedLifetimeCount=300, RequestedMaxKeepAliveCount=22,
                                                             MaxNotificationsPerPublish=10000, PublishingEnabled=True, Priority=0),
                               wait_timeout_retry_sec: typing.Union[int, float] = 1,
                               max_timeout_retry_nr: int = 5)
```

Bases: object

Configuration dataclass for OPC UA Communciation.

clean_values()

endpoint_name: str

Endpoint of the OPC server, this is a path like 'OPCUA/SimulationServer'

force_value(*fieldname*, *value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

host: Union[str, ipaddress.IPv4Address, ipaddress.IPv6Address]

Hostname or IP-Address of the OPC UA server.

is_configdataclass = True

classmethod **keys()** → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

max_timeout_retry_nr: int = 5

Maximal number of call re-tries on underlying OPC UA client timeout error

classmethod **optional_defaults()** → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

port: int = 4840

Port of the OPC UA server to connect to.

classmethod **required_keys()** → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

sub_handler: *hvl_ccb.comm.opc.OpcUaSubHandler* = <*hvl_ccb.comm.opc.OpcUaSubHandler* object>

object to use for handling subscriptions.

update_parameter: *asyncua.ua.uaproTOCOL_auto.CreateSubscriptionParameters* = *CreateSubscriptionParameters(RequestedPublishingInterval=1000, RequestedLifetimeCount=300, RequestedMaxKeepAliveCount=22, MaxNotificationsPerPublish=10000, PublishingEnabled=True, Priority=0)*

Values are given as a *ua.CreateSubscriptionParameters* as these parameters are requested by the OPC server. Other values will lead to an automatic revision of the parameters and a warning in the opc-logger, cf. MR !173

wait_timeout_retry_sec: Union[int, float] = 1

Wait time between re-trying calls on underlying OPC UA client timeout error

exception **OpcUaCommunicationIOError**

Bases: *OSError*, *hvl_ccb.comm.base.CommunicationException*

OPC-UA communication I/O error.

exception **OpcUaCommunicationTimeoutError**

Bases: *hvl_ccb.comm.opc.OpcUaCommunicationIOError*

OPC-UA communication timeout error.

class **OpcUaSubHandler**

Bases: object

Base class for subscription handling of OPC events and data change events. Override methods from this class to add own handling capabilities.

To receive events from server for a subscription data_change and event methods are called directly from receiving thread. Do not do expensive, slow or network operation there. Create another thread if you need to do such a thing.

datachange_notification(node, val, data)

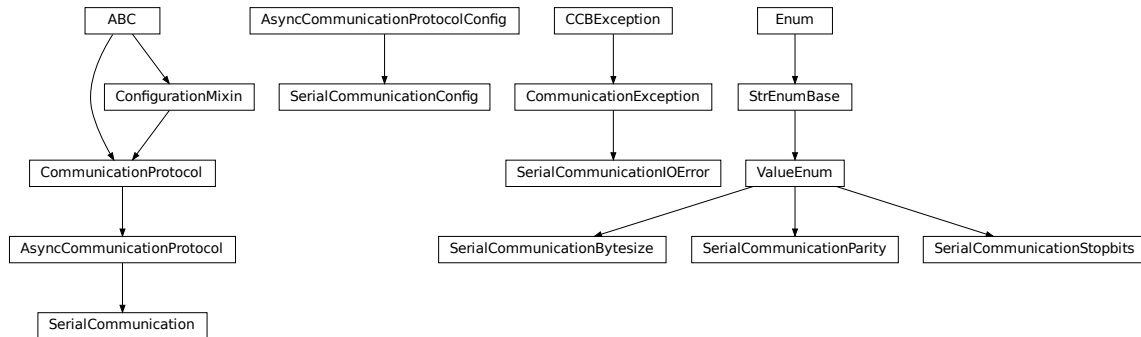
event_notification(event)

class **Server**(shelf_file=None, tloop=None)

Bases: *asyncua.sync.Server*

get_objects_node()

Get Objects node of server. Returns a Node object.

hvl_ccb.comm.serial

Communication protocol for serial ports. Makes use of the `pySerial` library.

class SerialCommunication(configuration)

Bases: `hvl_ccb.comm.base.AsyncCommunicationProtocol`

Implements the Communication Protocol for serial ports.

close()

Close the serial connection.

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

property is_open: bool

Flag indicating if the serial port is open.

Returns *True* if the serial port is open, otherwise *False*

open()

Open the serial connection.

Raises `SerialCommunicationIOError` – when communication port cannot be opened.

read_bytes() → bytes

Read the bytes from the serial port till the terminator is found. The input buffer may hold additional lines afterwards.

This method uses `self.access_lock` to ensure thread-safety.

Returns Bytes read from the serial port; *b''* if there was nothing to read.

Raises `SerialCommunicationIOError` – when communication port is not opened

read_single_bytes(size: int = 1) → bytes

Read the specified number of bytes from the serial port. The input buffer may hold additional data afterwards.

Returns Bytes read from the serial port; *b''* if there was nothing to read.

write_bytes(data: bytes) → int

Write bytes to the serial port.

This method uses *self.access_lock* to ensure thread-safety.

Parameters data – data to write to the serial port

Returns number of bytes written

Raises *SerialCommunicationIOError* – when communication port is not opened

class *SerialCommunicationBytesize*(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)

Bases: *hvl_ccb.utils.enum.ValueEnum*

Serial communication bytesize.

EIGHTBITS = 8

FIVEBITS = 5

SEVENBITS = 7

SIXBITS = 6

class *SerialCommunicationConfig*(terminator: bytes = b'\r\n', encoding: str = 'utf-8', encoding_error_handling: str = 'strict', wait_sec_read_text_nonempty: Union[int, float] = 0.5, default_n_attempts_read_text_nonempty: int = 10, port: Optional[str] = None, baudrate: int = 9600, parity: Union[str, *hvl_ccb.comm.serial.SerialCommunicationParity*] = *SerialCommunicationParity.NONE*, stopbits: Union[int, float, *hvl_ccb.comm.serial.SerialCommunicationStopbits*] = *SerialCommunicationStopbits.ONE*, bytesize: Union[int, *hvl_ccb.comm.serial.SerialCommunicationBytesize*] = *SerialCommunicationBytesize.EIGHTBITS*, timeout: Union[int, float] = 2)

Bases: *hvl_ccb.comm.base.AsyncCommunicationProtocolConfig*

Configuration dataclass for *SerialCommunication*.

Bytesize

alias of *hvl_ccb.comm.serial.SerialCommunicationBytesize*

Parity

alias of *hvl_ccb.comm.serial.SerialCommunicationParity*

Stopbits

alias of *hvl_ccb.comm.serial.SerialCommunicationStopbits*

baudrate: int = 9600

Baudrate of the serial port

bytesize: Union[int, *hvl_ccb.comm.serial.SerialCommunicationBytesize*] = 8

Size of a byte, 5 to 8

clean_values()

create_serial_port() → serial.serialposix.Serial

Create a serial port instance according to specification in this configuration

Returns Closed serial port instance

force_value(fieldname, value)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

parity: Union[str, [hvl_ccb.comm.serial.SerialCommunicationParity](#)] = 'N'

Parity to be used for the connection.

port: Optional[str] = None

Port is a string referring to a COM-port (e.g. 'COM3') or a URL. The full list of capabilities is found on [the pyserial documentation](#).

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

stopbits: Union[int, float, [hvl_ccb.comm.serial.SerialCommunicationStopbits](#)] = 1

Stopbits setting, can be 1, 1.5 or 2.

terminator_str() → str

timeout: Union[int, float] = 2

Timeout in seconds for the serial port

exception SerialCommunicationIOError

Bases: OSError, [hvl_ccb.comm.base.CommunicationException](#)

Serial communication related I/O errors.

class SerialCommunicationParity(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)

Bases: [hvl_ccb.utils.enum.ValueEnum](#)

Serial communication parity.

EVEN = 'E'

MARK = 'M'

NAMES = {'E': 'Even', 'M': 'Mark', 'N': 'None', 'O': 'Odd', 'S': 'Space'}

NONE = 'N'

ODD = 'O'

SPACE = 'S'

```
class SerialCommunicationStopbits(value=<no_arg>, names=None, module=None, type=None, start=1,
                                   boundary=None)
```

Bases: `hvl_ccb.utils.enum.ValueEnum`

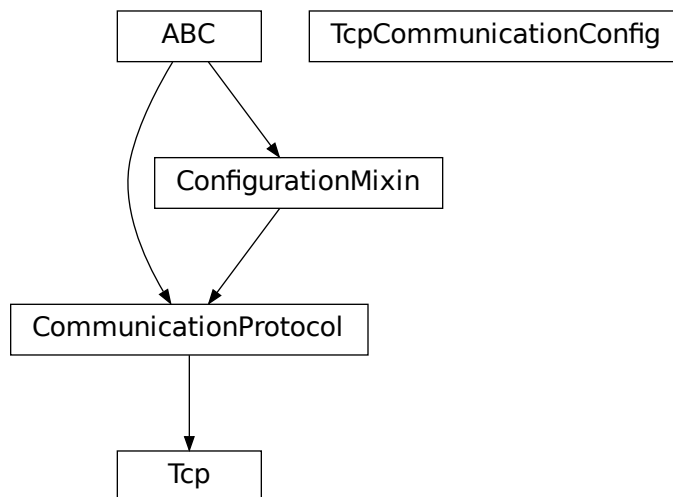
Serial communication stopbits.

ONE = 1

ONE_POINT_FIVE = 1.5

TWO = 2

`hvl_ccb.comm.tcp`



TCP communication protocol.

Makes use of the socket library.

```
class Tcp(configuration)
```

Bases: `hvl_ccb.comm.base.CommunicationProtocol`

Tcp Communication Protocol.

close() → *None*

Close TCP connection.

static config_cls() → `Type[hvl_ccb.comm.tcp.TcpCommunicationConfig]`

Return the default configdataclass class.

Returns a reference to the default configdataclass class

open() → *None*

Open TCP connection.

read() → `str`

TCP read function :return: information read from TCP buffer formatted as string

write(*command: str = ""*) → *None*

TCP write function :param command: command string to be sent :return: none

class TcpCommunicationConfig(*host: Union[str, ipaddress.IPv4Address, ipaddress.IPv6Address], port: int = 54321, bufsize: int = 1024*)

Bases: object

Configuration dataclass for TcpCommunication.

bufsize: int = 1024

clean_values()

force_value(*fieldname, value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

host: Union[str, ipaddress.IPv4Address, ipaddress.IPv6Address]

is_configdataclass = True

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

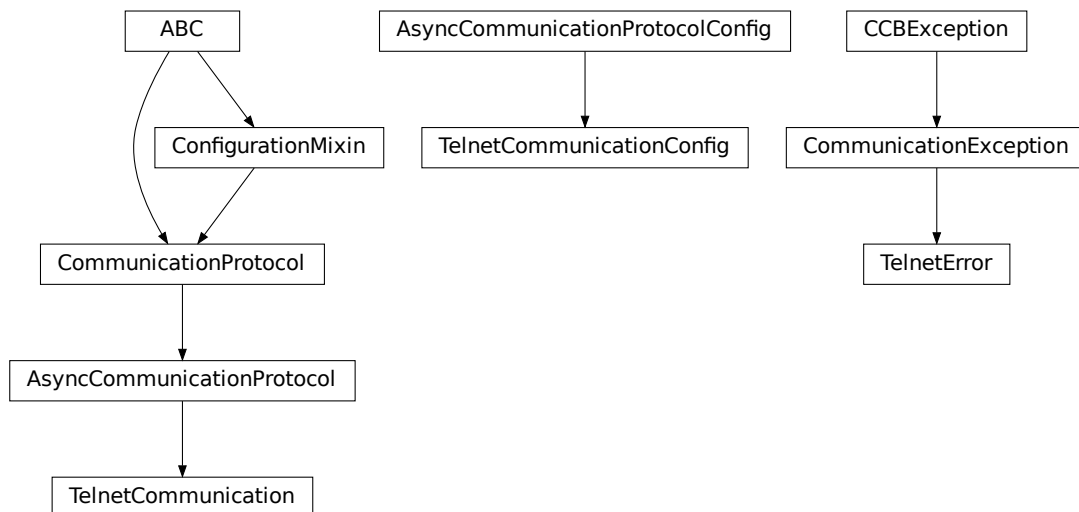
port: int = 54321

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

hvl_ccb.comm.telnet



Communication protocol for telnet. Makes use of the [telnetlib](#) library.

class `TelnetCommunication(configuration)`

Bases: `hvl_ccb.comm.base.AsyncCommunicationProtocol`

Implements the Communication Protocol for telnet.

close()

Close the telnet connection unless it is not closed.

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

property is_open: bool

Is the connection open?

Returns True for an open connection

open()

Open the telnet connection unless it is not yet opened.

read_bytes() → bytes

Read data as *bytes* from the telnet connection.

Returns data from telnet connection

Raises `TelnetError` – when connection is not open, raises an Error during the communication

write_bytes(data: bytes)

Write the data as *bytes* to the telnet connection.

Parameters `data` – Data to be sent.

Raises `TelnetError` – when connection is not open, raises an Error during the communication


```
class TelnetCommunicationConfig(terminator: bytes = b'\n\n', encoding: str = 'utf-8',
                               encoding_error_handling: str = 'strict', wait_sec_read_text_nonempty:
                               Union[int, float] = 0.5, default_n_attempts_read_text_nonempty: int = 10,
                               host: Optional[Union[str, ipaddress.IPv4Address, ipaddress.IPv6Address]]
                               = None, port: int = 0, timeout: Union[int, float] = 0.2)
```

Bases: [hvl_ccb.comm.base.AsyncCommunicationProtocolConfig](#)

Configuration dataclass for [TelnetCommunication](#).

clean_values()

create_telnet() → Optional[telnetlib.Telnet]

Create a telnet client :return: Opened Telnet object or None if connection is not possible

force_value(fieldname, value)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

host: Optional[Union[str, ipaddress.IPv4Address, ipaddress.IPv6Address]] = None

Host to connect to can be localhost or

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

port: int = 0

Port at which the host is listening

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

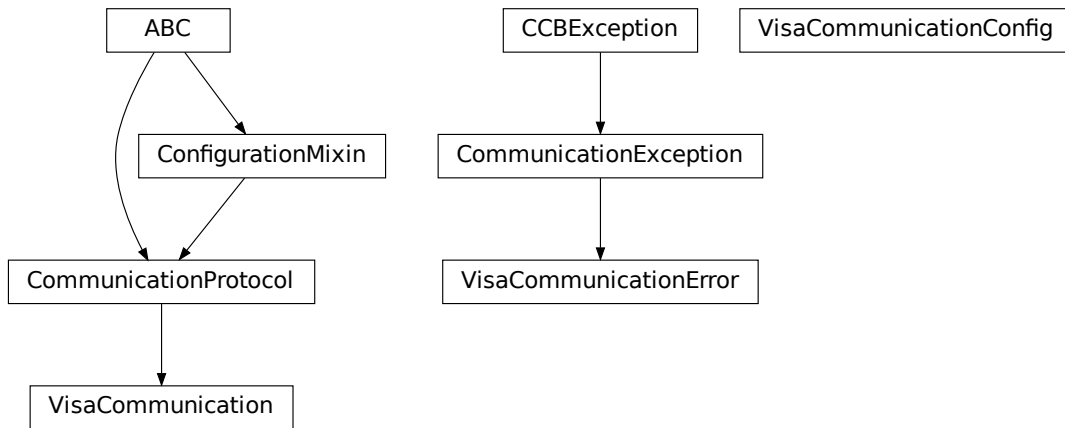
timeout: Union[int, float] = 0.2

Timeout for reading a line

exception TelnetError

Bases: OSError, [hvl_ccb.comm.base.CommunicationException](#)

Telnet communication related errors.

hvl_ccb.comm.visa

Communication protocol for VISA. Makes use of the pyvisa library. The backend can be NI-Visa or pyvisa-py.

Information on how to install a VISA backend can be found here: https://pyvisa.readthedocs.io/en/master/getting_nivisa.html

So far only TCPIP SOCKET and TCPIP INSTR interfaces are supported.

class VisaCommunication(*configuration*)

Bases: *hvl_ccb.comm.base.CommunicationProtocol*

Implements the Communication Protocol for VISA / SCPI.

MULTI_COMMANDS_MAX = 5

The maximum of commands that can be sent in one round is 5 according to the VISA standard.

MULTI_COMMANDS_SEPARATOR = ';'

The character to separate two commands is ; according to the VISA standard.

WAIT_AFTER_WRITE = 0.08

Small pause in seconds to wait after write operations, allowing devices to really do what we tell them before continuing with further tasks.

close() → *None*

Close the VISA connection and invalidates the handle.

static config_cls() → *Type[hvl_ccb.comm.visa.VisaCommunicationConfig]*

Return the default configdataclass class.

Returns a reference to the default configdataclass class

open() → *None*

Open the VISA connection and create the resource.

query(**commands*: *str*) → *Union[str, Tuple[str, ...]]*

A combination of write(message) and read.

Parameters **commands** – list of commands

Returns list of values

Raises *VisaCommunicationError* – when connection was not started, or when trying to issue too many commands at once.

spoll() → int

Execute serial poll on the device. Reads the status byte register STB. This is a fast function that can be executed periodically in a polling fashion.

Returns integer representation of the status byte

Raises *VisaCommunicationError* – when connection was not started

write(*commands: str) → None

Write commands. No answer is read or expected.

Parameters **commands** – one or more commands to send

Raises *VisaCommunicationError* – when connection was not started

```
class VisaCommunicationConfig(host: Union[str, ipaddress.IPv4Address, ipaddress.IPv6Address],
                              interface_type: Union[str,
                              hvl_ccb.comm.visa.VisaCommunicationConfig.InterfaceType], board: int =
                              0, port: int = 5025, timeout: int = 5000, chunk_size: int = 204800,
                              open_timeout: int = 1000, write_termination: str = '\n', read_termination: str
                              = '\n', visa_backend: str = "")
```

Bases: object

VisaCommunication configuration dataclass.

```
class InterfaceType(value=<no_arg>, names=None, module=None, type=None, start=1,
                    boundary=None)
```

Bases: *hvl_ccb.utils.enum.AutoNumberNameEnum*

Supported VISA Interface types.

TCPIP_INSTR = 2

VXI-11 protocol

TCPIP_SOCKET = 1

VISA-RAW protocol

address(host: str, port: Optional[int] = None, board: Optional[int] = None) → str

Address string specific to the VISA interface type.

Parameters

- **host** – host IP address
- **port** – optional TCP port
- **board** – optional board number

Returns address string

property address: str

Address string depending on the VISA protocol's configuration.

Returns address string corresponding to current configuration

board: int = 0

Board number is typically 0 and comes from old bus systems.

chunk_size: int = 204800

Chunk size is the allocated memory for read operations. The standard is 20kB, and is increased per default here to 200kB. It is specified in bytes.

clean_values()

force_value(*fieldname*, *value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

host: Union[str, `ipaddress.IPv4Address`, `ipaddress.IPv6Address`]

interface_type: Union[str, `hvl_ccb.comm.visa.VisaCommunicationConfig.InterfaceType`]

Interface type of the VISA connection, being one of `InterfaceType`.

is_configdataclass = True

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

open_timeout: int = 1000

Timeout for opening the connection, in milli seconds.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

port: int = 5025

TCP port, standard is 5025.

read_termination: str = '\n'

Read termination character.

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

timeout: int = 5000

Timeout for commands in milli seconds.

visa_backend: str = ''

Specifies the path to the library to be used with PyVISA as a backend. Defaults to None, which is NI-VISA (if installed), or pyvisa-py (if NI-VISA is not found). To force the use of pyvisa-py, specify '@py' here.

write_termination: str = '\n'

Write termination character.

exception VisaCommunicationError

Bases: `OSError`, `hvl_ccb.comm.base.CommunicationException`

Base class for VisaCommunication errors.

Module contents

Communication protocols subpackage.

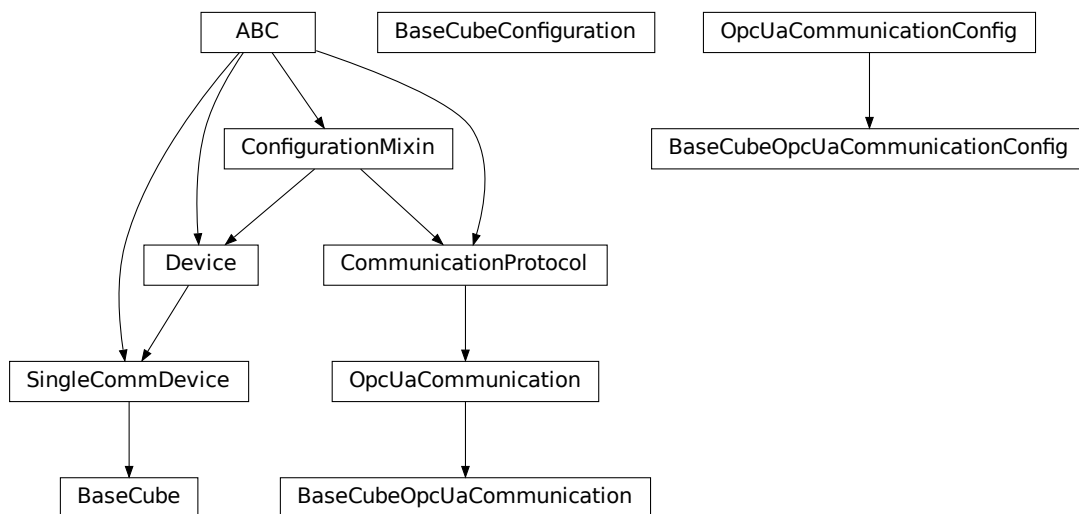
hvl_ccb.dev

Subpackages

hvl_ccb.dev.cube

Submodules

hvl_ccb.dev.cube.base



Classes for the BaseCube device.

class `BaseCube`(*com*, *dev_config=None*)

Bases: `hvl_ccb.dev.base.SingleCommDevice`

Base class for Cube variants.

OPC_MAX_YEAR = 2089

OPC_MIN_YEAR = 1990

active_alarms(*human_readable: bool = True*) → List[Union[int, str]]

Displays all active alarms / messages.

Parameters *human_readable* – *True* for human readable message, *False* for corresponding integer

Returns list with active alarms

property breakdown_detection_active: bool

Get the state of the breakdown detection functionality. Returns True if it is enabled, False otherwise.

Returns state of the breakdown detection functionality

breakdown_detection_reset() → None

Reset the breakdown detection circuitry so that it is ready to detect breakdowns again.

property breakdown_detection_triggered: bool

See if breakdown detection unit has been triggered. Returns True if it is triggered, False otherwise.

Returns trigger status of the breakdown detection unit

property cee16_socket

Read the on-state of the IEC CEE16 three-phase power socket.

Returns the on-state of the CEE16 power socket

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

classmethod datetime_to_opc(time_dt: datetime.datetime) → List[int]

Converts python datetime format into opc format (list of 8 integers) as defined in the following link: <https://support.industry.siemens.com/cs/mdm/109798671?c=133950752267&lc=de-WW> Each byte corresponds to one list entry. [yy, MM, dd, hh, mm, ss, milliseconds, weekday] Milliseconds and Weekday are not used, as this precision / information is not needed. The conversion of the numbers is special. Each decimal number is treated as it would be a hex-number and then converted back to decimal. This is tested with the used PLC in the BaseCube. yy: 0 to 99 (0 -> 2000, 89 -> 2089, 90 -> 1990, 99 -> 1999) MM: 1 to 12 dd: 1 to 31 hh: 0 to 23 mm: 0 to 59 ss: 0 to 59

Parameters **time_dt** – time to be converted

Returns time in opc list format

static default_com_cls()

Get the class for the default communication protocol used with this device.

Returns the type of the standard communication protocol for this device

display_message_board() → None

Display 15 newest messages

display_status_board() → None

Display status board.

door_1_status

Get the status of a safety fence door. See constants.DoorStatus for possible returned door statuses.

door_2_status

Get the status of a safety fence door. See constants.DoorStatus for possible returned door statuses.

door_3_status

Get the status of a safety fence door. See constants.DoorStatus for possible returned door statuses.

earthing_rod_1_status

Get the status of a earthing rod. See constants.EarthingRodStatus for possible returned earthing rod statuses.

earthing_rod_2_status

Get the status of a earthing rod. See constants.EarthingRodStatus for possible returned earthing rod statuses.

earthing_rod_3_status

Get the status of a earthing rod. See `constants.EarthingRodStatus` for possible returned earthing rod statuses.

property operate: Optional[bool]

Indicates if 'operate' is activated. 'operate' means locket safety circuit, red lamps, high voltage on and locked safety switches.

Returns *True* if operate is activated (`RED_OPERATE`), *False* if ready is deactivated (`RED_READY`), *None* otherwise

quit_error() → *None*

Quits errors that are active on the Cube.

read(node_id: str)

Local wrapper for the OPC UA communication protocol read method.

Parameters `node_id` – the id of the node to read.

Returns the value of the variable

property ready: Optional[bool]

Indicates if 'ready' is activated. 'ready' means locket safety circuit, red lamps, but high voltage still off.

Returns *True* if ready is activated (`RED_READY`), *False* if ready is deactivated (`GREEN_READY`), *None* otherwise

set_message_board(msgs: List[str], display_board: bool = True) → *None*

Fills messages into message board that display that 15 newest messages with a timestamp.

Parameters

- `msgs` – list of strings
- `display_board` – display 15 newest messages if *True* (default)

Raises **ValueError** – if there are too many messages or the positions indices are invalid.

set_status_board(msgs: List[str], pos: Optional[List[int]] = None, clear_board: bool = True, display_board: bool = True) → *None*

Sets and displays a status board. The messages and the position of the message can be defined.

Parameters

- `msgs` – list of strings
- `pos` – list of integers [0...14]
- `clear_board` – clear unspecified lines if *True* (default), keep otherwise
- `display_board` – display new status board if *True* (default)

Raises **ValueError** – if there are too many messages or the positions indices are invalid.

start() → *None*

Starts the device. Sets the root node for all OPC read and write commands to the Siemens PLC object node which holds all our relevant objects and variables.

property status: hvl_ccb.dev.cube.constants.SafetyStatus

Get the safety circuit status of the Cube. This methods is for the user.

Returns the safety status of the Cube's state machine.

stop() → *None*

Stop the Cube device. Deactivates the remote control and closes the communication protocol.

Raises *CubeStopError* – when the cube is not in the correct status to stop the operation

t13_socket_1

Set and get the state of a SEV T13 power socket.

t13_socket_2

Set and get the state of a SEV T13 power socket.

t13_socket_3

Set and get the state of a SEV T13 power socket.

write(*node_id*, *value*) → *None*

Local wrapper for the OPC UA communication protocol write method.

Parameters

- **node_id** – the id of the node to write
- **value** – the value to write to the variable

```
class BaseCubeConfiguration(namespace_index: int = 3, polling_delay_sec: Union[int, float] = 5.0,  
                             polling_interval_sec: Union[int, float] = 1.0, timeout_status_change:  
                             Union[int, float] = 6, timeout_interval: Union[int, float] = 0.1,  
                             noise_level_measurement_channel_1: Union[int, float] = 100,  
                             noise_level_measurement_channel_2: Union[int, float] = 100,  
                             noise_level_measurement_channel_3: Union[int, float] = 100,  
                             noise_level_measurement_channel_4: Union[int, float] = 100)
```

Bases: object

Configuration dataclass for the BaseCube devices.

clean_values()

force_value(*fieldname*, *value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

is_configdataclass = **True**

classmethod **keys**() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

namespace_index: **int** = **3**

Namespace of the OPC variables, typically this is 3 (coming from Siemens)

noise_level_measurement_channel_1: **Union[int, float]** = **100**

noise_level_measurement_channel_2: **Union[int, float]** = **100**

noise_level_measurement_channel_3: **Union[int, float]** = **100**

noise_level_measurement_channel_4: **Union[int, float]** = **100**

classmethod **optional_defaults**() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

polling_delay_sec: `Union[int, float] = 5.0`

polling_interval_sec: `Union[int, float] = 1.0`

classmethod required_keys() \rightarrow Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

timeout_interval: `Union[int, float] = 0.1`

timeout_status_change: `Union[int, float] = 6`

class BaseCubeOpcUaCommunication(*config*)

Bases: `hvl_ccb.comm.opc.OpcUaCommunication`

Communication protocol specification for BaseCube devices.

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

```
class BaseCubeOpcUaCommunicationConfig(host: typing.Union[str, ipaddress.IPv4Address,
ipaddress.IPv6Address], endpoint_name:
hvl_ccb.dev.cube.constants._CubeOpcEndpoint =
_CubeOpcEndpoint.BASE_CUBE, port: int = 4840, sub_handler:
hvl_ccb.comm.opc.OpcUaSubHandler =
<hvl_ccb.dev.cube.base._BaseCubeSubscriptionHandler object>,
update_parameter:
asynua.ua.uaprotocol_auto.CreateSubscriptionParameters =
CreateSubscriptionParameters(RequestedPublishingInterval=1000,
RequestedLifetimeCount=300,
RequestedMaxKeepAliveCount=22,
MaxNotificationsPerPublish=10000, PublishingEnabled=True,
Priority=0), wait_timeout_retry_sec: typing.Union[int, float] = 1,
max_timeout_retry_nr: int = 5)
```

Bases: `hvl_ccb.comm.opc.OpcUaCommunicationConfig`

Communication protocol configuration for OPC UA, specifications for the BaseCube devices.

endpoint_name: `hvl_ccb.dev.cube.constants._CubeOpcEndpoint = 'BaseCube'`

Endpoint of the OPC server, this is a path like 'OPCUA/SimulationServer'

force_value(*fieldname, value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

classmethod keys() \rightarrow Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod `optional_defaults()` → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

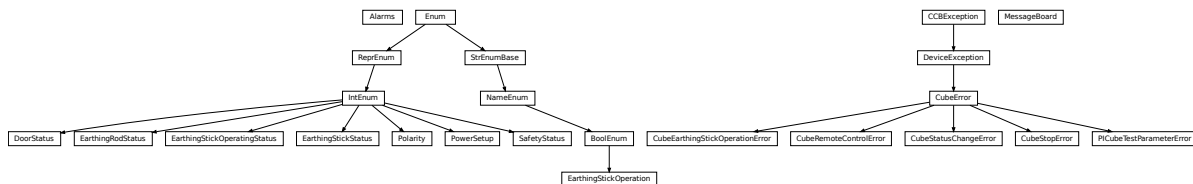
classmethod `required_keys()` → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

sub_handler: `hvl_ccb.comm.opc.OpcUaSubHandler =`
`<hvl_ccb.dev.cube.base._BaseCubeSubscriptionHandler object>`
 Subscription handler for data change events

`hvl_ccb.dev.cube.constants`



Constants, variable names for the BaseCube OPC-connected devices.

exception `CubeEarthingStickOperationError`

Bases: `hvl_ccb.dev.cube.constants.CubeError`

exception `CubeError`

Bases: `hvl_ccb.dev.base.DeviceException`

exception `CubeRemoteControlError`

Bases: `hvl_ccb.dev.cube.constants.CubeError`

exception `CubeStatusChangeError`

Bases: `hvl_ccb.dev.cube.constants.CubeError`

exception `CubeStopError`

Bases: `hvl_ccb.dev.cube.constants.CubeError`

class `DoorStatus(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)`

Bases: `aenum.IntEnum`

Possible status values for doors.

CLOSED = 2

Door is closed, but not locked.

ERROR = 4

Door has an error or was opened in locked state (either with emergency stop or from the inside).

INACTIVE = 0

not enabled in BaseCube HMI setup, this door is not supervised.

LOCKED = 3

Door is closed and locked (safe state).

OPEN = 1

Door is open.

```
class EarthingRodStatus(value=<no_arg>, names=None, module=None, type=None, start=1,  
                        boundary=None)
```

Bases: `aenum.IntEnum`

Possible status values for earthing rods.

EXPERIMENT_BLOCKED = 0

earthing rod is somewhere in the experiment and blocks the start of the experiment

EXPERIMENT_READY = 1

earthing rod is hanging next to the door, experiment is ready to operate

```
class EarthingStickOperatingStatus(value=<no_arg>, names=None, module=None, type=None, start=1,  
                                   boundary=None)
```

Bases: `aenum.IntEnum`

Operating Status for an earthing stick. Stick can be used in auto or manual mode.

AUTO = 0

MANUAL = 1

```
class EarthingStickOperation(value=<no_arg>, names=None, module=None, type=None, start=1,  
                              boundary=None)
```

Bases: `hvl_ccb.utils.enum.BoolEnum`

Operation of the earthing stick in manual operating mode. Can be closed or opened.

CLOSE = True

OPEN = False

```
class EarthingStickStatus(value=<no_arg>, names=None, module=None, type=None, start=1,  
                           boundary=None)
```

Bases: `aenum.IntEnum`

Status of an earthing stick. These are the possible values in the status integer e.g. in `_EarthingStick.status`.

CLOSED = 1

ERROR = 3

INACTIVE = 0

OPEN = 2

```
class MessageBoard(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)
```

Bases: `hvl_ccb.dev.cube.constants._LineEnumBase`

Variable NodeID strings for message board lines.

LINE_1 = "DB_OPC_Connection"."Is_status_Line_1"

LINE_10 = "DB_OPC_Connection"."Is_status_Line_10"

LINE_11 = "DB_OPC_Connection"."Is_status_Line_11"

LINE_12 = "DB_OPC_Connection"."Is_status_Line_12"

LINE_13 = "DB_OPC_Connection"."Is_status_Line_13"

LINE_14 = "DB_OPC_Connection"."Is_status_Line_14"

```
LINE_15 = '"DB_OPC_Connection"."Is_status_Line_15"'
LINE_2 = '"DB_OPC_Connection"."Is_status_Line_2"'
LINE_3 = '"DB_OPC_Connection"."Is_status_Line_3"'
LINE_4 = '"DB_OPC_Connection"."Is_status_Line_4"'
LINE_5 = '"DB_OPC_Connection"."Is_status_Line_5"'
LINE_6 = '"DB_OPC_Connection"."Is_status_Line_6"'
LINE_7 = '"DB_OPC_Connection"."Is_status_Line_7"'
LINE_8 = '"DB_OPC_Connection"."Is_status_Line_8"'
LINE_9 = '"DB_OPC_Connection"."Is_status_Line_9"'
```

exception `PICubeTestParameterError`

Bases: [hvl_ccb.dev.cube.constants.CubeError](#)

class `Polarity`(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `aenum.IntEnum`

An enumeration.

NEGATIVE = 0

POSITIVE = 1

class `PowerSetup`(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `aenum.IntEnum`

Possible power setups corresponding to the value of variable `Power.setup`. The values for `slope_min` are experimentally defined, below these values the slope is more like a staircase

The name of the first argument needs to be 'value', otherwise the `IntEnum` is not working correctly.

AC_100KV = 3

AC_150KV = 4

AC_200KV = 5

AC_50KV = 2

DC_140KV = 7

DC_280KV = 8

EXTERNAL_SOURCE = 1

IMPULSE_140KV = 9

NO_SOURCE = 0

POWER_INVERTER_220V = 6

STOP_SAFETY_STATUSES: `Tuple[hvl_ccb.dev.cube.constants.SafetyStatus, ...]` =

(`<SafetyStatus.GREEN_NOT_READY: 1>`, `<SafetyStatus.GREEN_READY: 2>`)

BaseCube's safety statuses required to close the connection to the device.

class `SafetyStatus`(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `aenum.IntEnum`

Safety status values that are possible states returned from [hvl_ccb.dev.cube.base.BaseCube.status\(\)](#). These values correspond to the states of the BaseCube's safety circuit statemachine.

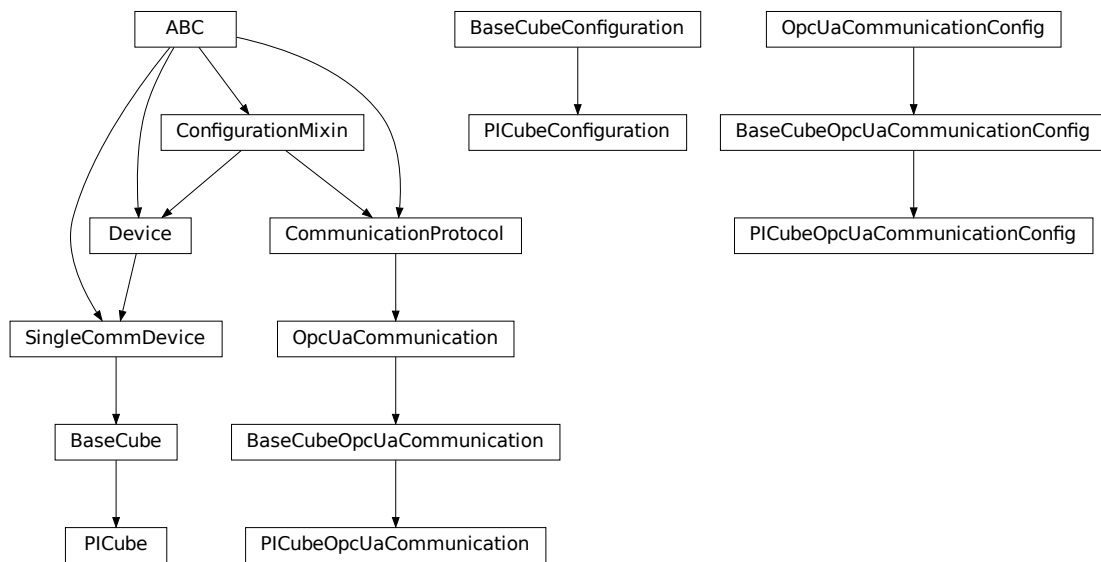
ERROR = 6

```

GREEN_NOT_READY = 1
GREEN_READY = 2
INITIALIZING = 0
QUICK_STOP = 5
RED_OPERATE = 4
RED_READY = 3

```

hvl_ccb.dev.cube.picube



A PICube is a BaseCube with build in Power Inverter

class `PICube`(*com*, *dev_config*=None)

Bases: `hvl_ccb.dev.cube.base.BaseCube`

Variant of the BaseCube with build in Power Inverter

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

property current_primary: `float`

Read the current primary current at the output of the frequency converter (before transformer).

Returns primary current in A

static default_com_cls()

Get the class for the default communication protocol used with this device.

Returns the type of the standard communication protocol for this device

property frequency: `float`

Read the electrical frequency of the current PICube setup.

Returns the frequency in Hz

property operate: `Optional[bool]`

Indicates if 'operate' is activated. 'operate' means locket safety circuit, red lamps, high voltage on and locked safety switches.

Returns *True* if operate is activated (RED_OPERATE), *False* if ready is deactivated (RED_READY), *None* otherwise

property polarity: `Optional[hvl_ccb.dev.cube.constants.Polarity]`

Polarity of a DC setup. :return: if a DC setup is programmed the polarity is returned, else *None*.

property power_setup: `hvl_ccb.dev.cube.constants.PowerSetup`

Return the power setup selected in the PICube's settings.

Returns the power setup

property voltage_actual: `float`

Reads the actual measured voltage and returns the value in V.

Returns the actual voltage of the setup in V.

property voltage_max: `float`

Reads the maximum voltage of the setup and returns in V.

Returns the maximum voltage of the setup in V.

property voltage_primary: `float`

Read the current primary voltage at the output of the frequency converter (before transformer).

Returns primary voltage in V

```
class PICubeConfiguration(namespace_index: int = 3, polling_delay_sec: Union[int, float] = 5.0,
                           polling_interval_sec: Union[int, float] = 1.0, timeout_status_change: Union[int,
                                                                                                     float] = 6, timeout_interval: Union[int, float] = 0.1,
                           noise_level_measurement_channel_1: Union[int, float] = 100,
                           noise_level_measurement_channel_2: Union[int, float] = 100,
                           noise_level_measurement_channel_3: Union[int, float] = 100,
                           noise_level_measurement_channel_4: Union[int, float] = 100,
                           timeout_test_parameters: 'Number' = 2.0)
```

Bases: `hvl_ccb.dev.cube.base.BaseCubeConfiguration`

clean_values()

force_value(*fieldname*, *value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

classmethod keys() → `Sequence[str]`

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → `Dict[str, object]`

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

timeout_test_parameters: Union[int, float] = 2.0

class PICubeOpcUaCommunication(*config*)

Bases: [hvl_ccb.dev.cube.base.BaseCubeOpcUaCommunication](#)

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

class PICubeOpcUaCommunicationConfig(*host: Union[str, ipaddress.IPv4Address, ipaddress.IPv6Address], endpoint_name: '_CubeOpcEndpoint' = <_CubeOpcEndpoint.PI_CUBE: 'PICube'>, port: int = 4840, sub_handler: hvl_ccb.comm.opc.OpcUaSubHandler = <hvl_ccb.dev.cube.base._BaseCubeSubscriptionHandler object at 0x7f295edb9310>, update_parameter: asyncua.ua.uaprotoocol_auto.CreateSubscriptionParameters = CreateSubscriptionParameters(RequestedPublishingInterval=1000, RequestedLifetimeCount=300, RequestedMaxKeepAliveCount=22, MaxNotificationsPerPublish=10000, PublishingEnabled=True, Priority=0), wait_timeout_retry_sec: Union[int, float] = 1, max_timeout_retry_nr: int = 5)*)

Bases: [hvl_ccb.dev.cube.base.BaseCubeOpcUaCommunicationConfig](#)

endpoint_name: _CubeOpcEndpoint = 'PICube'

Endpoint of the OPC server, this is a path like 'OPCUA/SimulationServer'

force_value(*fieldname, value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

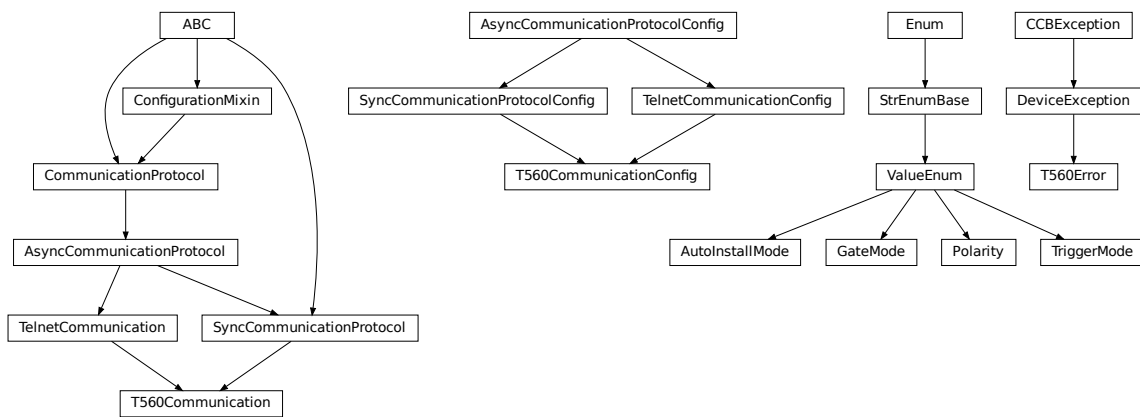
Module contents

Cube package with implementation for system versions from 2019 on (new concept with hard-PLC Siemens S7-1500 as CPU).

hvl_ccb.dev.highland_t560

Submodules

hvl_ccb.dev.highland_t560.base



Module containing base device and communication classes and enums.

Communication with device is performed via its ethernet port and a Telnet connection.

class `AutoInstallMode`(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)
Bases: `hvl_ccb.utils.enum.ValueEnum`

Modes for installing configuration settings to the device.

INSTALL = 1

OFF = 0

QUEUE = 2

class `GateMode`(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)
Bases: `hvl_ccb.utils.enum.ValueEnum`

Available T560 gate modes

INPUT = 'INP'

OFF = 'OFF'

OUTPUT = 'OUT'

class `Polarity`(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)
Bases: `hvl_ccb.utils.enum.ValueEnum`

Possible channel polarity states

ACTIVE_HIGH = 'POS'

ACTIVE_LOW = 'NEG'

class T560Communication(*configuration*)

Bases: [hvl_ccb.comm.base.SyncCommunicationProtocol](#), [hvl_ccb.comm.telnet.TelnetCommunication](#)

Communication class for T560. It uses a TelnetCommunication with the SyncCommunicationProtocol

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

query(*command: str*) → str

Send a command to the device and handle the response.

For device setting queries, response will be 'OK' if successful, or '??' if setting cannot be carried out, raising an error.

Parameters **command** – Command string to be sent

Raises [T560Error](#) – if no response is received, or if the device responds with an error message.

Returns Response from the device.

class T560CommunicationConfig(*terminator: bytes = b'\r', encoding: str = 'utf-8', encoding_error_handling: str = 'strict', wait_sec_read_text_nonempty: Union[int, float] = 0.5, default_n_attempts_read_text_nonempty: int = 10, host: Union[str, ipaddress.IPv4Address, ipaddress.IPv6Address, NoneType] = None, port: int = 2000, timeout: Union[int, float] = 0.2*)

Bases: [hvl_ccb.comm.base.SyncCommunicationProtocolConfig](#), [hvl_ccb.comm.telnet.TelnetCommunicationConfig](#)

force_value(*fieldname, value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

port: int = 2000

Port at which the host is listening

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

terminator: `bytes = b'\r'`

The terminator character. Typically this is `b'\r\n'` or `b'\n'`, but can also be `b'\r'` or other combinations.

This defines the end of a single line.

exception T560Error

Bases: `hvl_ccb.dev.base.DeviceException`

T560 related errors.

class TriggerMode(*value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None*)

Bases: `hvl_ccb.utils.enum.ValueEnum`

Available T560 trigger modes

`COMMAND = 'REM'`

`EXT_FALLING_EDGE = 'NEG'`

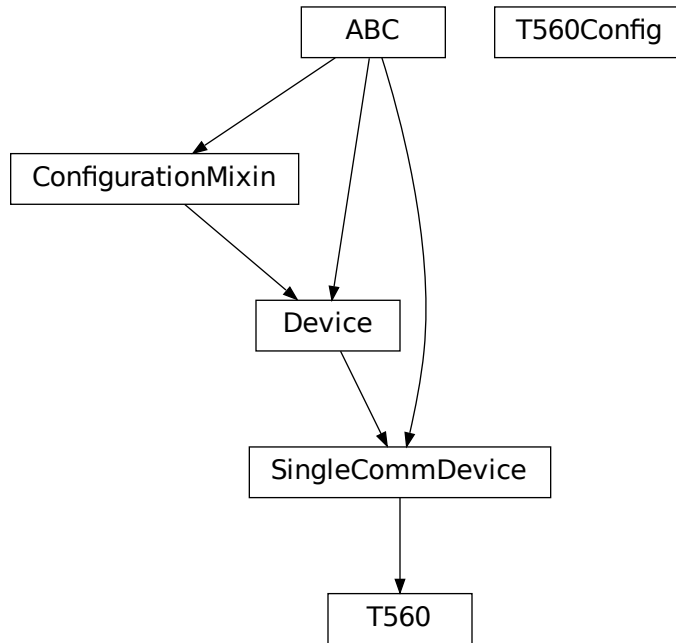
`EXT_RISING_EDGE = 'POS'`

`INT_SYNTHESIZER = 'SYN'`

`OFF = 'OFF'`

`hvl_ccb.dev.highland_t560.channel`

Module for controlling pulse output channels A, B, C and D.

hvl_ccb.dev.highland_t560.device

including TRIG, CLOCK and GATE I/Os.

Module for controlling device,

class T560(com, dev_config=None)

Bases: *hvl_ccb.dev.base.SingleCommDevice*

activate_clock_output()

Outputs 10 MHz clock signal

property auto_install_mode: *hvl_ccb.dev.highland_t560.base.AutoInstallMode*

Check the autoinstall settings of the T560. The autoinstall mode sets how changes to device settings are applied. See manual section 4.7.2 for more information about these modes.

property ch_a: *hvl_ccb.dev.highland_t560.channel._Channel*

Channel A of T560

property ch_b: *hvl_ccb.dev.highland_t560.channel._Channel*

Channel B of T560

property ch_c: *hvl_ccb.dev.highland_t560.channel._Channel*

Channel C of T560

property ch_d: *hvl_ccb.dev.highland_t560.channel._Channel*

Channel D of T560

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

static default_com_cls()

Get the class for the default communication protocol used with this device.

Returns the type of the standard communication protocol for this device

disarm_trigger()

Disarm DDG by disabling all trigger sources.

fire_trigger()

Fire a software trigger.

property frequency: float

The frequency of the timing cycle in Hz.

property gate_mode: *hvl_ccb.dev.highland_t560.base.GateMode*

Check the mode setting of the GATE I/O port.

property gate_polarity: *hvl_ccb.dev.highland_t560.base.Polarity*

Check the polarity setting of the GATE I/O port.

load_device_configuration()

Load the settings saved in nonvolatile memory.

property period: float

The period of the timing cycle (time between triggers) in seconds.

save_device_configuration()

Save the current settings to nonvolatile memory.

property trigger_level

Get external trigger level.

property trigger_mode

Get device trigger source.

use_external_clock()

Finds and accepts an external clock signal to the CLOCK input

class T560Config

Bases: object

auto_install_mode = 1

clean_values()

Cleans and enforces configuration values. Does nothing by default, but may be overridden to add custom configuration value checks.

force_value(*fieldname*, *value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

is_configdataclass = True

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

Module contents

This module establishes methods for interfacing with the Highland Technology T560-2 via its ethernet adapter with a telnet communication protocol.

The T560 is a small digital delay & pulse generator. It outputs up to four individually timed pulses with 10-ps precision, given an internal or external trigger.

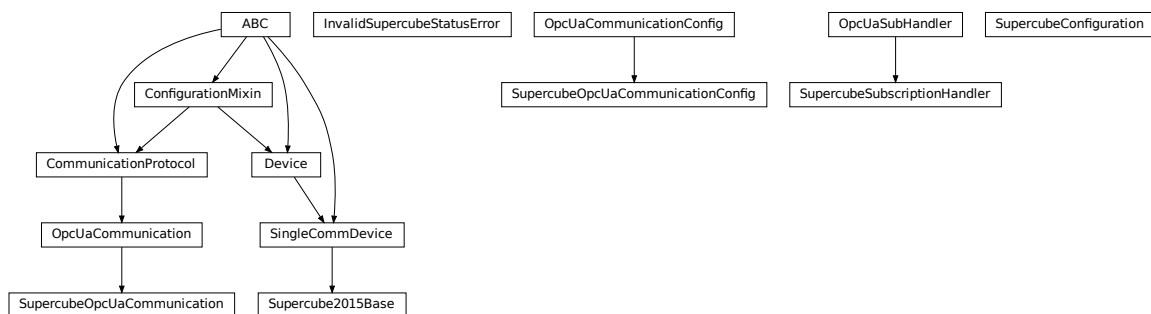
This module introduces methods for configuring channels, gating, and triggering. Further documentation and a more extensive command list may be obtained from:

<https://www.highlandtechnology.com/DSS/T560DS.shtml>

hvl_ccb.dev.supercube2015

Submodules

hvl_ccb.dev.supercube2015.base



Base classes for the Supercube device.

exception InvalidSupercubeStatusError

Bases: Exception

Exception raised when supercube has invalid status.

class Supercube2015Base(com, dev_config=None)

Bases: *hvl_ccb.dev.base.SingleCommDevice*

Base class for Supercube variants.

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

static default_com_cls()

Get the class for the default communication protocol used with this device.

Returns the type of the standard communication protocol for this device

get_cee16_socket() → bool

Read the on-state of the IEC CEE16 three-phase power socket.

Returns the on-state of the CEE16 power socket

get_door_status(door: int) → [*hvl_ccb.dev.supercube2015.constants.DoorStatus*](#)

Get the status of a safety fence door. See `constants.DoorStatus` for possible returned door statuses.

Parameters **door** – the door number (1..3)

Returns the door status

get_earthing_manual(number: int) → bool

Get the manual status of an earthing stick. If an earthing stick is set to manual, it is closed even if the system is in states RedReady or RedOperate.

Parameters **number** – number of the earthing stick (1..6)

Returns earthing stick manual status

get_earthing_status(number: int) → int

Get the status of an earthing stick, whether it is closed, open or undefined (moving).

Parameters **number** – number of the earthing stick (1..6)

Returns earthing stick status; see `constants.EarthingStickStatus`

get_measurement_ratio(channel: int) → float

Get the set measurement ratio of an AC/DC analog input channel. Every input channel has a divider ratio assigned during setup of the Supercube system. This ratio can be read out.

Attention: Supercube 2015 does not have a separate ratio for every analog input. Therefore there is only one ratio for `channel = 1`.

Parameters **channel** – number of the input channel (1..4)

Returns the ratio

get_measurement_voltage(channel: int) → float

Get the measured voltage of an analog input channel. The voltage read out here is already scaled by the configured divider ratio.

Attention: In contrast to the *new* Supercube, the old one returns here the input voltage read at the ADC. It is not scaled by a factor.

Parameters **channel** – number of the input channel (1..4)

Returns measured voltage

get_status() → int

Get the safety circuit status of the Supercube.

Returns the safety status of the supercube's state machine; see `constants.SafetyStatus`.

get_support_input(port: int, contact: int) → bool

Get the state of a support socket input.

Parameters

- **port** – is the socket number (1..6)
- **contact** – is the contact on the socket (1..2)

Returns digital input read state

get_support_output(*port: int, contact: int*) → bool

Get the state of a support socket output.

Parameters

- **port** – is the socket number (1..6)
- **contact** – is the contact on the socket (1..2)

Returns digital output read state

get_t13_socket(*port: int*) → bool

Read the state of a SEV T13 power socket.

Parameters **port** – is the socket number, one of *constants.T13_SOCKET_PORTS*

Returns on-state of the power socket

horn(*state: bool*) → *None*

Turns acoustic horn on or off.

Parameters **state** – Turns horn on (True) or off (False)

operate(*state: bool*) → *None*

Set operate state. If the state is RedReady, this will turn on the high voltage and close the safety switches.

Parameters **state** – set operate state

quit_error() → *None*

Quits errors that are active on the Supercube.

read(*node_id: str*)

Local wrapper for the OPC UA communication protocol read method.

Parameters **node_id** – the id of the node to read.

Returns the value of the variable

ready(*state: bool*) → *None*

Set ready state. Ready means locket safety circuit, red lamps, but high voltage still off.

Parameters **state** – set ready state

set_cee16_socket(*state: bool*) → *None*

Switch the IEC CEE16 three-phase power socket on or off.

Parameters **state** – desired on-state of the power socket

Raises **TypeError** – if state is not of type bool

set_earthing_manual(*number: int, manual: bool*) → *None*

Set the manual status of an earthing stick. If an earthing stick is set to manual, it is closed even if the system is in states RedReady or RedOperate.

Parameters

- **number** – number of the earthing stick (1..6)
- **manual** – earthing stick manual status (True or False)

set_remote_control(state: bool) → *None*

Enable or disable remote control for the Supercube. This will effectively display a message on the touch-screen HMI.

Parameters **state** – desired remote control state

set_support_output(port: int, contact: int, state: bool) → *None*

Set the state of a support output socket.

Parameters

- **port** – is the socket number (1..6)
- **contact** – is the contact on the socket (1..2)
- **state** – is the desired state of the support output

Raises **TypeError** – when state is not of type bool

set_support_output_impulse(port: int, contact: int, duration: float = 0.2, pos_pulse: bool = True) → *None*

Issue an impulse of a certain duration on a support output contact. The polarity of the pulse (On-wait-Off or Off-wait-On) is specified by the pos_pulse argument.

This function is blocking.

Parameters

- **port** – is the socket number (1..6)
- **contact** – is the contact on the socket (1..2)
- **duration** – is the length of the impulse in seconds
- **pos_pulse** – is True, if the pulse shall be HIGH, False if it shall be LOW

Raises **TypeError** – when state is not of type bool

set_t13_socket(port: int, state: bool) → *None*

Set the state of a SEV T13 power socket.

Parameters

- **port** – is the socket number, one of *constants.T13_SOCKET_PORTS*
- **state** – is the desired on-state of the socket

Raises

- **ValueError** – when port is not valid
- **TypeError** – when state is not of type bool

start() → *None*

Starts the device. Sets the root node for all OPC read and write commands to the Siemens PLC object node which holds all our relevant objects and variables.

stop() → *None*

Stop the Supercube device. Deactivates the remote control and closes the communication protocol.

write(node_id, value) → *None*

Local wrapper for the OPC UA communication protocol write method.

Parameters

- **node_id** – the id of the node to read
- **value** – the value to write to the variable

class SupercubeConfiguration(*namespace_index: int = 7*)

Bases: object

Configuration dataclass for the Supercube devices.

clean_values()

Cleans and enforces configuration values. Does nothing by default, but may be overridden to add custom configuration value checks.

force_value(*fieldname, value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

is_configdataclass = True

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

namespace_index: int = 7

Namespace of the OPC variables, typically this is 3 (coming from Siemens)

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

class SupercubeOpcUaCommunication(*config*)

Bases: *hvl_ccb.comm.opc.OpcUaCommunication*

Communication protocol specification for Supercube devices.

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

```
class SupercubeOpcUaCommunicationConfig(host: typing.Union[str, ipaddress.IPv4Address,  
                                         ipaddress.IPv6Address], endpoint_name: str, port: int = 4845,  
                                         sub_handler: hvl_ccb.comm.opc.OpcUaSubHandler =  
                                         <hvl_ccb.dev.supercube2015.base.SupercubeSubscriptionHandler  
                                         object>, update_parameter:  
                                         asynua.ua.uaprotocol_auto.CreateSubscriptionParameters =  
                                         CreateSubscriptionParame-  
                                         ters(RequestedPublishingInterval=1000,  
                                         RequestedLifetimeCount=300,  
                                         RequestedMaxKeepAliveCount=22,  
                                         MaxNotificationsPerPublish=10000, PublishingEnabled=True,  
                                         Priority=0), wait_timeout_retry_sec: typing.Union[int, float] =  
                                         1, max_timeout_retry_nr: int = 5)
```

Bases: `hvl_ccb.comm.opc.OpcUaCommunicationConfig`

Communication protocol configuration for OPC UA, specifications for the Supercube devices.

force_value(*fieldname, value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

classmethod **keys**() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod **optional_defaults**() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

port: **int** = 4845

Port of the OPC UA server to connect to.

classmethod **required_keys**() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

sub_handler: `hvl_ccb.comm.opc.OpcUaSubHandler =`
`<hvl_ccb.dev.supercube2015.base.SupercubeSubscriptionHandler object>`

Subscription handler for data change events

class SupercubeSubscriptionHandler

Bases: `hvl_ccb.comm.opc.OpcUaSubHandler`

OPC Subscription handler for datachange events and normal events specifically implemented for the Supercube devices.

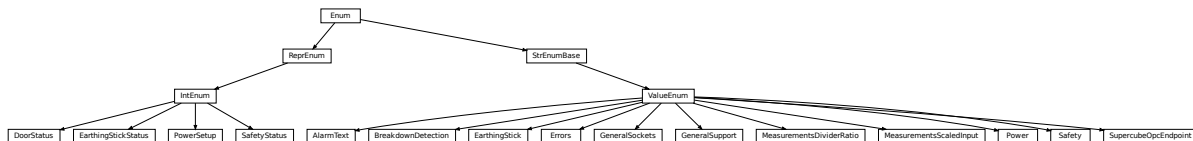
datachange_notification(*node: asynua.sync.SyncNode, val, data*)

In addition to the standard operation (debug logging entry of the datachange), alarms are logged at INFO level using the alarm text.

Parameters

- **node** – the node object that triggered the datachange event
- **val** – the new value
- **data** –

`hvl_ccb.dev.supercube2015.constants`



Constants, variable names for the Supercube OPC-connected devices.

class `AlarmText`(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)
 Bases: `hvl_ccb.utils.enum.ValueEnum`

This enumeration contains textual representations for all error classes (stop, warning and message) of the Supercube system. Use the `AlarmText.get()` method to retrieve the enum of an alarm number.

```

Alarm0 = 'No Alarm.'
Alarm1 = 'STOP Safety switch 1 error'
Alarm10 = 'STOP Earthing stick 2 error'
Alarm11 = 'STOP Earthing stick 3 error'
Alarm12 = 'STOP Earthing stick 4 error'
Alarm13 = 'STOP Earthing stick 5 error'
Alarm14 = 'STOP Earthing stick 6 error'
Alarm17 = 'STOP Source switch error'
Alarm19 = 'STOP Fence 1 error'
Alarm2 = 'STOP Safety switch 2 error'
Alarm20 = 'STOP Fence 2 error'
Alarm21 = 'STOP Control error'
Alarm22 = 'STOP Power outage'
Alarm3 = 'STOP Emergency Stop 1'
Alarm4 = 'STOP Emergency Stop 2'
Alarm5 = 'STOP Emergency Stop 3'
Alarm6 = 'STOP Door 1 lock supervision'
Alarm7 = 'STOP Door 2 lock supervision'
Alarm8 = 'STOP Door 3 lock supervision'
Alarm9 = 'STOP Earthing stick 1 error'
  
```

classmethod `get(alarm: int)`

Get the attribute of this enum for an alarm number.

Parameters `alarm` – the alarm number

Returns the enum for the desired alarm number

`not_defined = 'NO ALARM TEXT DEFINED'`

class `BreakdownDetection(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)`

Bases: `hvl_ccb.utils.enum.ValueEnum`

Node ID strings for the breakdown detection.

`activated = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.Breakdowndetection.connect'`

Boolean read-only variable indicating whether breakdown detection and fast switchoff is enabled in the system or not.

`reset = 'hvl-ipc.WINAC.Support60utA'`

Boolean writable variable to reset the fast switch-off. Toggle to re-enable.

`triggered = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.Breakdowndetection.triggered'`

Boolean read-only variable telling whether the fast switch-off has triggered. This can also be seen using the safety circuit state, therefore no method is implemented to read this out directly.

class `DoorStatus(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)`

Bases: `aenum.IntEnum`

Possible status values for doors.

`closed = 2`

Door is closed, but not locked.

`error = 4`

Door has an error or was opened in locked state (either with emergency stop or from the inside).

`inactive = 0`

not enabled in Supercube HMI setup, this door is not supervised.

`locked = 3`

Door is closed and locked (safe state).

`open = 1`

Door is open.

class `EarthingStick(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)`

Bases: `hvl_ccb.utils.enum.ValueEnum`

Variable NodeID strings for all earthing stick statuses (read-only integer) and writable booleans for setting the earthing in manual mode.

classmethod `manual(number: int)`

Get the manual enum attribute for an earthing stick number.

Parameters `number` – the earthing stick (1..6)

Returns the manual enum

`manual_1 = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_1.MANUAL'`

`manual_2 = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_2.MANUAL'`

`manual_3 = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_3.MANUAL'`

`manual_4 = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_4.MANUAL'`

```

manual_5 = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_5.MANUAL'
manual_6 = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_6.MANUAL'
status_1_closed = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_1.CLOSE'
status_1_connected = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_1.CONNECT'
status_1_open = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_1.OPEN'
status_2_closed = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_2.CLOSE'
status_2_connected = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_2.CONNECT'
status_2_open = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_2.OPEN'
status_3_closed = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_3.CLOSE'
status_3_connected = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_3.CONNECT'
status_3_open = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_3.OPEN'
status_4_closed = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_4.CLOSE'
status_4_connected = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_4.CONNECT'
status_4_open = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_4.OPEN'
status_5_closed = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_5.CLOSE'
status_5_connected = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_5.CONNECT'
status_5_open = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_5.OPEN'
status_6_closed = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_6.CLOSE'
status_6_connected = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_6.CONNECT'
status_6_open = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_6.OPEN'

```

classmethod status_closed(*number: int*)

Get the status enum attribute for an earthing stick number.

Parameters **number** – the earthing stick (1..6)

Returns the status enum

classmethod status_connected(*number: int*)

Get the status enum attribute for an earthing stick number.

Parameters **number** – the earthing stick (1..6)

Returns the status enum

classmethod status_open(*number: int*)

Get the status enum attribute for an earthing stick number.

Parameters **number** – the earthing stick (1..6)

Returns the status enum

class EarthingStickStatus(*value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None*)

Bases: aenum.IntEnum

Status of an earthing stick. These are the possible values in the status integer e.g. in `EarthingStick.status_1`.

closed = 1

Earthing is closed (safe).

error = 3

Earthing is in error, e.g. when the stick did not close correctly or could not open.

inactive = 0

Earthing stick is deselected and not enabled in safety circuit. To get out of this state, the earthing has to be enabled in the Supercube HMI setup.

open = 2

Earthing is open (not safe).

class Errors(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)

Bases: [hvl_ccb.utils.enum.ValueEnum](#)

Variable NodeID strings for information regarding error, warning and message handling.

quit = 'hvl-ipc.WINAC.SYSTEMSTATE.Faultconfirmation'

Writable boolean for the error quit button.

stop = 'hvl-ipc.WINAC.SYSTEMSTATE.ERROR'

Boolean read-only variable telling if a stop is active.

stop_number = 'hvl-ipc.WINAC.SYSTEMSTATE.Errornumber'

class GeneralSockets(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)

Bases: [hvl_ccb.utils.enum.ValueEnum](#)

NodeID strings for the power sockets (3x T13 and 1xCEE16).

cee16 = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.CEE16'

CEE16 socket (writeable boolean).

t13_1 = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.T13_1'

SEV T13 socket No. 1 (writable boolean).

t13_2 = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.T13_2'

SEV T13 socket No. 2 (writable boolean).

t13_3 = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.T13_3'

SEV T13 socket No. 3 (writable boolean).

class GeneralSupport(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)

Bases: [hvl_ccb.utils.enum.ValueEnum](#)

NodeID strings for the support inputs and outputs.

in_1_1 = 'hvl-ipc.WINAC.Support1InA'

in_1_2 = 'hvl-ipc.WINAC.Support1InB'

in_2_1 = 'hvl-ipc.WINAC.Support2InA'

in_2_2 = 'hvl-ipc.WINAC.Support2InB'

in_3_1 = 'hvl-ipc.WINAC.Support3InA'

in_3_2 = 'hvl-ipc.WINAC.Support3InB'

in_4_1 = 'hvl-ipc.WINAC.Support4InA'

in_4_2 = 'hvl-ipc.WINAC.Support4InB'

in_5_1 = 'hvl-ipc.WINAC.Support5InA'

in_5_2 = 'hvl-ipc.WINAC.Support5InB'

```
in_6_1 = 'hvl-ipc.WINAC.Support6InA'
```

```
in_6_2 = 'hvl-ipc.WINAC.Support6InB'
```

```
classmethod input(port, contact)
```

Get the NodeID string for a support input.

Parameters

- **port** – the desired port (1..6)
- **contact** – the desired contact at the port (1..2)

Returns the node id string

```
out_1_1 = 'hvl-ipc.WINAC.Support10OutA'
```

```
out_1_2 = 'hvl-ipc.WINAC.Support10OutB'
```

```
out_2_1 = 'hvl-ipc.WINAC.Support20OutA'
```

```
out_2_2 = 'hvl-ipc.WINAC.Support20OutB'
```

```
out_3_1 = 'hvl-ipc.WINAC.Support30OutA'
```

```
out_3_2 = 'hvl-ipc.WINAC.Support30OutB'
```

```
out_4_1 = 'hvl-ipc.WINAC.Support40OutA'
```

```
out_4_2 = 'hvl-ipc.WINAC.Support40OutB'
```

```
out_5_1 = 'hvl-ipc.WINAC.Support50OutA'
```

```
out_5_2 = 'hvl-ipc.WINAC.Support50OutB'
```

```
out_6_1 = 'hvl-ipc.WINAC.Support60OutA'
```

```
out_6_2 = 'hvl-ipc.WINAC.Support60OutB'
```

```
classmethod output(port, contact)
```

Get the NodeID string for a support output.

Parameters

- **port** – the desired port (1..6)
- **contact** – the desired contact at the port (1..2)

Returns the node id string

```
class MeasurementsDividerRatio(value=<no_arg>, names=None, module=None, type=None, start=1,
                                boundary=None)
```

Bases: [hvl_ccb.utils.enum.ValueEnum](#)

Variable NodeID strings for the measurement input scaling ratios. These ratios are defined in the Supercube HMI setup and are provided in the python module here to be able to read them out, allowing further calculations.

```
classmethod get(channel: int)
```

Get the attribute for an input number.

Parameters **channel** – the channel number (1..4)

Returns the enum for the desired channel.

```
input_1 = 'hvl-ipc.WINAC.SYSTEM_INTERN.DivididerRatio'
```

```
class MeasurementsScaledInput(value=<no_arg>, names=None, module=None, type=None, start=1,
                               boundary=None)
```

Bases: [hvl_ccb.utils.enum.ValueEnum](#)

Variable NodeID strings for the four analog BNC inputs for measuring voltage. The voltage returned in these variables is already scaled with the set ratio, which can be read using the variables in [MeasurementsDividerRatio](#).

```
classmethod get(channel: int)
```

Get the attribute for an input number.

Parameters `channel` – the channel number (1..4)

Returns the enum for the desired channel.

```
input_1 = 'hvl-ipc.WINAC.SYSTEM_INTERN.AI1Volt'
```

```
input_2 = 'hvl-ipc.WINAC.SYSTEM_INTERN.AI2Volt'
```

```
input_3 = 'hvl-ipc.WINAC.SYSTEM_INTERN.AI3Volt'
```

```
input_4 = 'hvl-ipc.WINAC.SYSTEM_INTERN.AI4Volt'
```

```
class Power(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)
```

Bases: [hvl_ccb.utils.enum.ValueEnum](#)

Variable NodeID strings concerning power data.

```
current_primary = 'hvl-ipc.WINAC.SYSTEM_INTERN.FUCurrentprim'
```

Primary current in ampere, measured by the frequency converter. (read-only)

```
frequency = 'hvl-ipc.WINAC.FU.Frequency'
```

Frequency converter output frequency. (read-only)

```
setup = 'hvl-ipc.WINAC.FU.TrafoSetup'
```

Power setup that is configured using the Supercube HMI. The value corresponds to the ones in [PowerSetup](#). (read-only)

```
voltage_max = 'hvl-ipc.WINAC.FU.maxVoltagekV'
```

Maximum voltage allowed by the current experimental setup. (read-only)

```
voltage_primary = 'hvl-ipc.WINAC.SYSTEM_INTERN.FUVoltageprim'
```

Primary voltage in volts, measured by the frequency converter at its output. (read-only)

```
voltage_slope = 'hvl-ipc.WINAC.FU.dUdt_-1'
```

Voltage slope in V/s.

```
voltage_target = 'hvl-ipc.WINAC.FU.SOLL'
```

Target voltage setpoint in V.

```
class PowerSetup(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)
```

Bases: [aenum.IntEnum](#)

Possible power setups corresponding to the value of variable [Power.setup](#).

```
AC_DoubleStage_150kV = 3
```

AC voltage with two MWB transformers, one at 100kV and the other at 50kV, resulting in a total maximum voltage of 150kV.

```
AC_DoubleStage_200kV = 4
```

AC voltage with two MWB transformers both at 100kV, resulting in a total maximum voltage of 200kV

```
AC_SingleStage_100kV = 2
```

AC voltage with MWB transformer set to 100kV maximum voltage.

AC_SingleStage_50kV = 1

AC voltage with MWB transformer set to 50kV maximum voltage.

DC_DoubleStage_280kV = 7

DC voltage with two AC transformers set to 100kV AC each, resulting in 280kV DC in total (or a single stage transformer with Greinacher voltage doubling rectifier)

DC_SingleStage_140kV = 6

DC voltage with one AC transformer set to 100kV AC, resulting in 140kV DC

External = 0

External power supply fed through blue CEE32 input using isolation transformer and safety switches of the Supercube, or using an external safety switch attached to the Supercube Type B.

Internal = 5

Internal usage of the frequency converter, controlling to the primary voltage output of the supercube itself (no measurement transformer used)

class Safety(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)

Bases: [hvl_ccb.utils.enum.ValueEnum](#)

NodeID strings for the basic safety circuit status and green/red switches “ready” and “operate”.

horn = 'hvl-ipc.WINAC.SYSTEM_INTERN.hornen'

Writable boolean to manually turn on or off the horn

status_error = 'hvl-ipc.WINAC.SYSTEMSTATE.ERROR'

status_green = 'hvl-ipc.WINAC.SYSTEMSTATE.GREEN'

status_ready_for_red = 'hvl-ipc.WINAC.SYSTEMSTATE.ReadyForRed'

Status is a read-only integer containing the state number of the supercube-internal state machine. The values correspond to numbers in [SafetyStatus](#).

status_red = 'hvl-ipc.WINAC.SYSTEMSTATE.RED'

switchto_green = 'hvl-ipc.WINAC.SYSTEMSTATE.GREEN_REQUEST'

switchto_operate = 'hvl-ipc.WINAC.SYSTEMSTATE.switchon'

Writable boolean for switching to Red Operate (locket, HV on) state.

switchto_ready = 'hvl-ipc.WINAC.SYSTEMSTATE.RED_REQUEST'

Writable boolean for switching to Red Ready (locked, HV off) state.

class SafetyStatus(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)

Bases: [aenum.IntEnum](#)

Safety status values that are possible states returned from `hvl_ccb.dev.supercube.base.Supercube.get_status()`. These values correspond to the states of the Supercube’s safety circuit statemachine.

Error = 6

System is in error mode.

GreenNotReady = 1

System is safe, lamps are green and some safety elements are not in place such that it cannot be switched to red currently.

GreenReady = 2

System is safe and all safety elements are in place to be able to switch to *ready*.

Initializing = 0

System is initializing or booting.

QuickStop = 5

Fast turn off triggered and switched off the system. Reset FSO to go back to a normal state.

RedOperate = 4

System is locked in red state and in *operate* mode, i.e. high voltage on.

RedReady = 3

System is locked in red state and *ready* to go to *operate* mode.

class SupercubeOpcEndpoint(*value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None*)

Bases: [hvl_ccb.utils.enum.ValueEnum](#)

OPC Server Endpoint strings for the supercube variants.

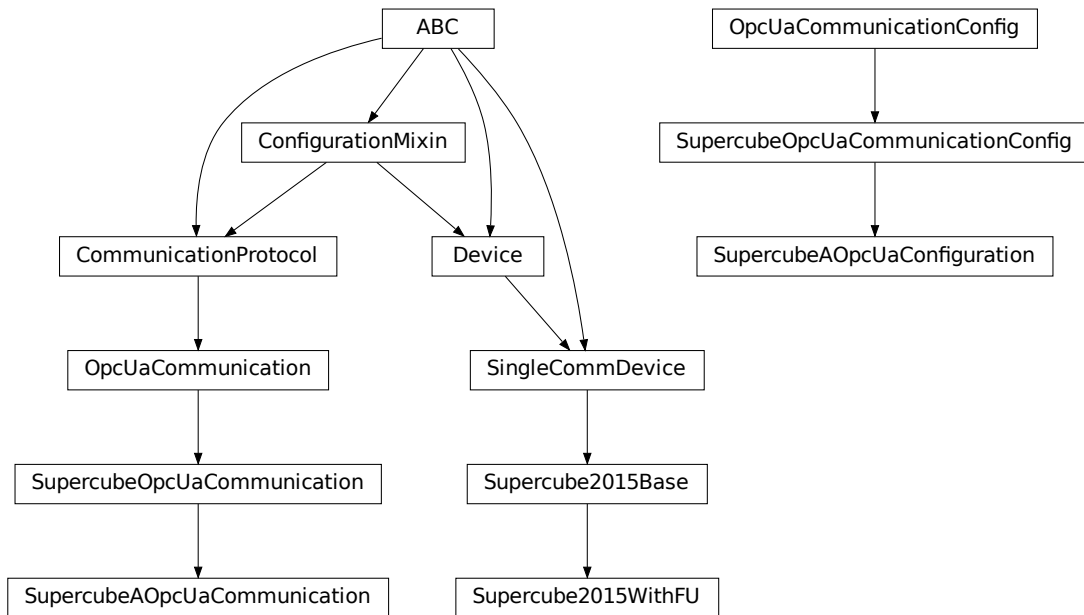
A = 'OPC.SimaticNET.S7'

B = 'OPC.SimaticNET.S7'

T13_SOCKET_PORTS = (1, 2, 3)

Port numbers of SEV T13 power socket

[hvl_ccb.dev.supercube2015.typ_a](#)



Supercube Typ A module.

class Supercube2015WithFU(*com, dev_config=None*)

Bases: [hvl_ccb.dev.supercube2015.base.Supercube2015Base](#)

Variant A of the Supercube with frequency converter.

static default_com_cls()

Get the class for the default communication protocol used with this device.

Returns the type of the standard communication protocol for this device

fso_reset() → *None*

Reset the fast switch off circuitry to go back into normal state and allow to re-enable operate mode.

get_frequency() → float

Read the electrical frequency of the current Supercube setup.

Returns the frequency in Hz

get_fso_active() → bool

Get the state of the fast switch off functionality. Returns True if it is enabled, False otherwise.

Returns state of the FSO functionality

get_max_voltage() → float

Reads the maximum voltage of the setup and returns in V.

Returns the maximum voltage of the setup in V.

get_power_setup() → *hvl_ccb.dev.supercube2015.constants.PowerSetup*

Return the power setup selected in the Supercube's settings.

Returns the power setup

get_primary_current() → float

Read the current primary current at the output of the frequency converter (before transformer).

Returns primary current in A

get_primary_voltage() → float

Read the current primary voltage at the output of the frequency converter (before transformer).

Returns primary voltage in V

get_target_voltage() → float

Gets the current setpoint of the output voltage value in V. This is not a measured value but is the corresponding function to *set_target_voltage()*.

Returns the setpoint voltage in V.

set_slope(slope: float) → *None*

Sets the dV/dt slope of the Supercube frequency converter to a new value in V/s.

Parameters **slope** – voltage slope in V/s (0..15'000)

set_target_voltage(volt_v: float) → *None*

Set the output voltage to a defined value in V.

Parameters **volt_v** – the desired voltage in V

class SupercubeA0pcUaCommunication(config)

Bases: *hvl_ccb.dev.supercube2015.base.SupercubeOpcUaCommunication*

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

```
class SupercubeA0pcUaConfiguration(host: Union[str, ipaddress.IPv4Address, ipaddress.IPv6Address],
                                   endpoint_name: str = 'OPC.SimaticNET.S7', port: int = 4845,
                                   sub_handler: hvl_ccb.comm.opc.OpcUaSubHandler =
                                   <hvl_ccb.dev.supercube2015.base.SupercubeSubscriptionHandler
                                   object at 0x7f29600e3710>, update_parameter:
                                   asyncua.ua.uaproto.col_auto.CreateSubscriptionParameters =
                                   CreateSubscriptionParameters(RequestedPublishingInterval=1000,
                                   RequestedLifetimeCount=300, RequestedMaxKeepAliveCount=22,
                                   MaxNotificationsPerPublish=10000, PublishingEnabled=True,
                                   Priority=0), wait_timeout_retry_sec: Union[int, float] = 1,
                                   max_timeout_retry_nr: int = 5)
```

Bases: [hvl_ccb.dev.supercube2015.base.SupercubeOpcUaCommunicationConfig](#)

endpoint_name: str = 'OPC.SimaticNET.S7'

Endpoint of the OPC server, this is a path like 'OPCUA/SimulationServer'

force_value(fieldname, value)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

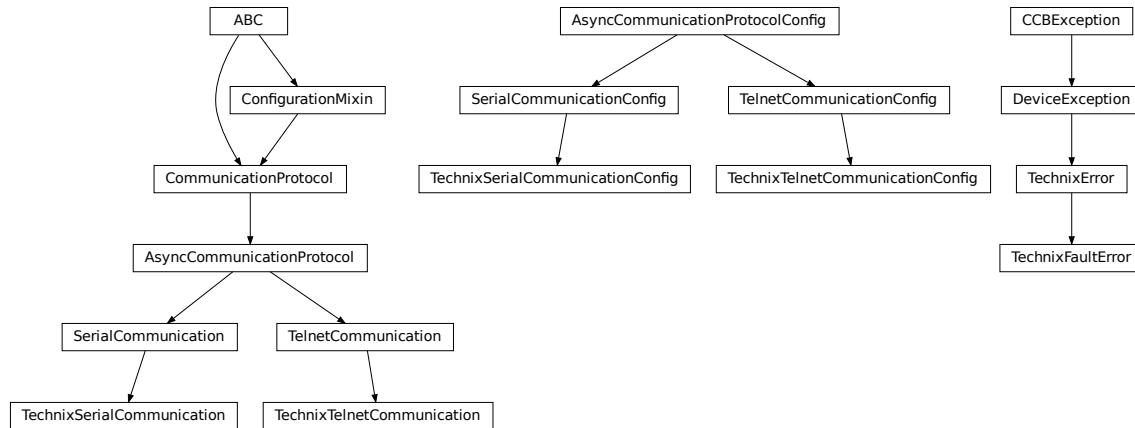
classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

Module contents

Supercube package with implementation for the old system version from 2015 based on Siemens WinAC soft-PLC on an industrial 32bit Windows computer.

hvl_ccb.dev.technix**Submodules****hvl_ccb.dev.technix.base**

Communication and auxiliary classes for Technix

exception TechnixErrorBases: [hvl_ccb.dev.base.DeviceException](#)

Technix related errors.

exception TechnixFaultErrorBases: [hvl_ccb.dev.technix.base.TechnixError](#)

Raised when the fault flag was detected while the interlock is closed

class TechnixSerialCommunication(configuration)Bases: [hvl_ccb.dev.technix.base._TechnixCommunication](#), [hvl_ccb.comm.serial.SerialCommunication](#)

Serial communication for Technix

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

```
class TechnixSerialCommunicationConfig(terminator: bytes = b'\r', encoding: str = 'utf-8',
                                       encoding_error_handling: str = 'strict',
                                       wait_sec_read_text_nonempty: Union[int, float] = 0.5,
                                       default_n_attempts_read_text_nonempty: int = 10, port:
                                       Optional[str] = None, baudrate: int = 9600, parity: Union[str,
                                       hvl_ccb.comm.serial.SerialCommunicationParity] =
                                       SerialCommunicationParity.NONE, stopbits: Union[int, float,
                                       hvl_ccb.comm.serial.SerialCommunicationStopbits] =
                                       SerialCommunicationStopbits.ONE, bytesize: Union[int,
                                       hvl_ccb.comm.serial.SerialCommunicationBytesize] =
                                       SerialCommunicationBytesize.EIGHTBITS, timeout: Union[int,
                                       float] = 2)

Bases:      hvl_ccb.dev.technix.base._TechnixCommunicationConfig,  hvl_ccb.comm.serial.
SerialCommunicationConfig
```

Configuration for the serial communication for Technix

force_value(*fieldname*, *value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

```
class TechnixTelnetCommunication(configuration)
```

Bases: [hvl_ccb.comm.telnet.TelnetCommunication](#), [hvl_ccb.dev.technix.base._TechnixCommunication](#)

Telnet communication for Technix

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

```
class TechnixTelnetCommunicationConfig(terminator: bytes = b'\r', encoding: str = 'utf-8',
                                     encoding_error_handling: str = 'strict',
                                     wait_sec_read_text_nonempty: Union[int, float] = 0.5,
                                     default_n_attempts_read_text_nonempty: int = 10, host:
                                     Optional[Union[str, ipaddress.IPv4Address,
                                     ipaddress.IPv6Address]] = None, port: int = 4660, timeout:
                                     Union[int, float] = 0.2)
```

Bases: `hvl_ccb.dev.technix.base._TechnixCommunicationConfig`, `hvl_ccb.comm.telnet.TelnetCommunicationConfig`

Configuration for the telnet communication for Technix

force_value(*fieldname*, *value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

classmethod **keys**() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod **optional_defaults**() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

port: **int** = **4660**

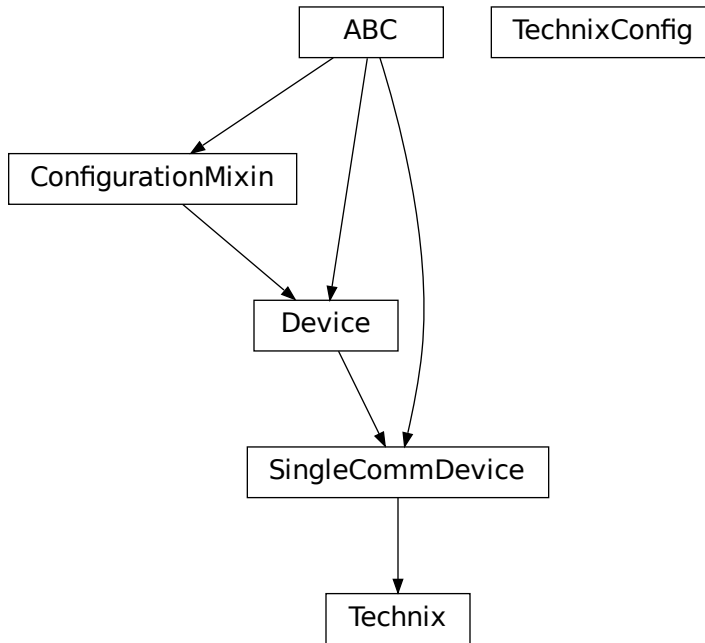
Port at which Technix is listening

classmethod **required_keys**() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

`hvl_ccb.dev.technix.device`



its corresponding configuration class

The device class *Technix* and

```
class Technix(com, dev_config)
```

Bases: `hvl_ccb.dev.base.SingleCommDevice`

Device class to control capacitor chargers from Technix

```
static config_cls()
```

Return the default configdataclass class.

Returns a reference to the default configdataclass class

```
property current: Optional[Union[int, float]]
```

Actual current of the output in A

```
default_com_cls() → Union[Type[hvl_ccb.dev.technix.base.TechnixSerialCommunication],  
                        Type[hvl_ccb.dev.technix.base.TechnixTelnetCommunication]]
```

Get the class for the default communication protocol used with this device.

Returns the type of the standard communication protocol for this device

```
property inhibit: Optional[bool]
```

Is the output of the voltage inhibited? The output stage can still be active.

```
property is_started: bool
```

Is the device started?

```
property max_current: Union[int, float]
```

Maximal output current of the hardware in A

property max_voltage: Union[int, float]

Maximal output voltage of the hardware in V

property open_interlock: Optional[bool]

Is the interlock open? (in safe mode)

property output: Optional[bool]

State of the high voltage output

query_status(*, _retry: bool = False)

Query the status of the device.

Returns This function returns nothing

property remote: Optional[bool]

Is the device in remote control mode?

start()

Start the device and set it into the remote controllable mode. The high voltage is turn off, and the status poller is started.

property status: Optional[hvl_ccb.dev.technix.base._Status]

The status of the device with the different states as sub-fields

stop()

Stop the device. The status poller is stopped and the high voltage output is turn off.

property voltage: Optional[Union[int, float]]

Actual voltage at the output in V

property voltage_regulation: Optional[bool]

Status if the output is in voltage regulation mode (or current regulation)

class TechnixConfig(communication_channel:

Union[Type[hvl_ccb.dev.technix.base.TechnixSerialCommunication],
Type[hvl_ccb.dev.technix.base.TechnixTelnetCommunication]], max_voltage: Union[int,
float], max_current: Union[int, float], polling_interval_sec: Union[int, float] = 4,
post_stop_pause_sec: Union[int, float] = 1, register_pulse_time: Union[int, float] = 0.1,
read_output_while_polling: bool = False)

Bases: object

clean_values()

Cleans and enforces configuration values. Does nothing by default, but may be overridden to add custom configuration value checks.

communication_channel:

Union[Type[hvl_ccb.dev.technix.base.TechnixSerialCommunication],

Type[hvl_ccb.dev.technix.base.TechnixTelnetCommunication]]

communication channel between computer and Technix

force_value(fieldname, value)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

is_configdataclass = True

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

max_current: Union[int, float]

Maximal Output current

max_voltage: Union[int, float]

Maximal Output voltage

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

polling_interval_sec: Union[int, float] = 4

Polling interval in s to maintain to watchdog of the device

post_stop_pause_sec: Union[int, float] = 1

Time to wait after stopping the device

read_output_while_polling: bool = False

Read output voltage and current within the polling event

register_pulse_time: Union[int, float] = 0.1

Time for pulsing a register

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

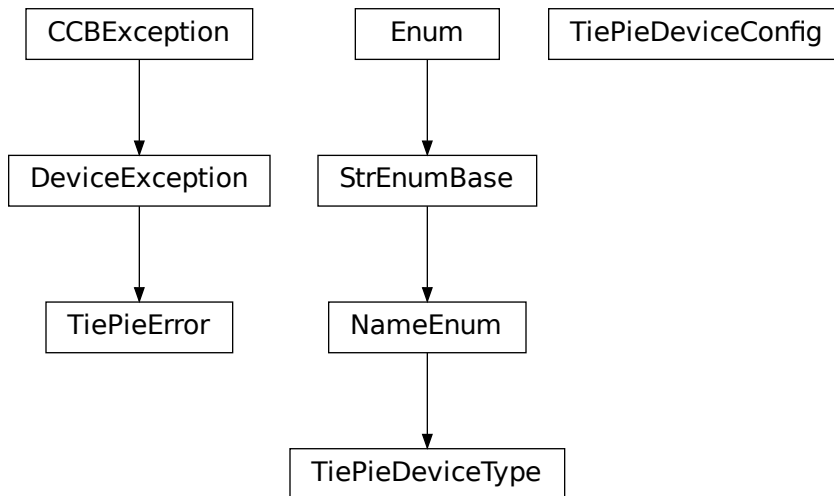
Module contents

Device classes for “RS 232” and “Ethernet” interfaces, which are used to control power supplies from Technix. Manufacturer homepage: <https://www.technix-hv.com>

The regulated power supplies series and capacitor chargers series from Technix are series of low and high voltage direct current power supplies as well as capacitor chargers. The class *Technix* is tested with a CCR10KV-7,5KJ via an ethernet connection as well as a CCR15-P-2500-OP via a serial connection. Check the code carefully before using it with other devices or device series

This Python package may support the following interfaces from Technix:

- [Remote Interface RS232](#)
- [Ethernet Remote Interface](#)
- [Optic Fiber Remote Interface](#)

hvl_ccb.dev.tiepie**Submodules****hvl_ccb.dev.tiepie.base**

```

class TiePieDeviceConfig(serial_number: int, require_block_measurement_support: bool = True,
                        n_max_try_get_device: int = 10, wait_sec_retry_get_device: Union[int, float] =
                        1.0, is_data_ready_polling_interval_sec: Union[int, float] = 0.01)

```

Bases: object

Configuration dataclass for TiePie

clean_values()

force_value(*fieldname, value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

is_configdataclass = **True**

is_data_ready_polling_interval_sec: **Union[int, float] = 0.01**

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

n_max_try_get_device: int = 10

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

require_block_measurement_support: bool = True

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

serial_number: int

wait_sec_retry_get_device: Union[int, float] = 1.0

class TiePieDeviceType(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)

Bases: [hvl_ccb.utils.enum.NameEnum](#)

TiePie device type.

GENERATOR = 2

I2C = 4

OSCILLOSCOPE = 1

exception TiePieError

Bases: [hvl_ccb.dev.base.DeviceException](#)

Error of the class TiePie

get_device_by_serial_number(serial_number: int, device_type: Union[str, Tuple[int, [hvl_ccb.dev.tiepie.base._LtpDeviceReturnType](#)]], n_max_try_get_device: int = 10, wait_sec_retry_get_device: float = 1.0) → [hvl_ccb.dev.tiepie.base._LtpDeviceReturnType](#)

Open and return handle of TiePie device with a given serial number

Parameters

- **serial_number** – int serial number of the device
- **device_type** – a *TiePieDeviceType* instance containing device identifier (int number) and its corresponding class, both from *libtiepie*, or a string name of such instance
- **n_max_try_get_device** – maximal number of device list updates (int number)
- **wait_sec_retry_get_device** – waiting time in seconds between retries (int number)

Returns Instance of a *libtiepie* device class according to the specified *device_type*

Raises

- **[TiePieError](#)** – when there is no device with given serial number
- **ValueError** – when *device_type* is not an instance of *TiePieDeviceType*

wrap_libtiepie_exception(func: Callable) → Callable

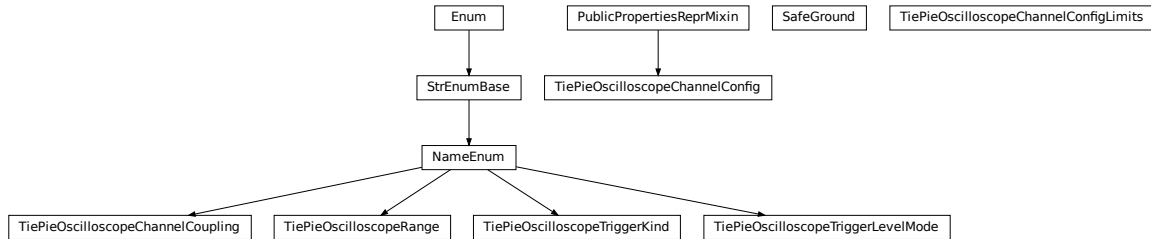
Decorator wrapper for *libtiepie* methods that use *libtiepie.library.check_last_status_raise_on_error()* calls.

Parameters **func** – Function or method to be wrapped

Raises *TiePieError* – instead of *LibTiePieException* or one of its subtypes.

Returns whatever *func* returns

`hvl_ccb.dev.tiepie.channel`



class `SafeGround`

Bases: `object`

Class that dynamically adds the `safe_ground_enabled` attribute getter/setter if the bound oscilloscope has the safe ground option.

class `TiePieOscilloscopeChannelConfig`(*ch_number: int, channel: libtiepie.oscilloscopechannel.OscilloscopeChannel*)

Bases: `hvl_ccb.dev.tiepie.utils.PublicPropertiesReprMixin`

Oscilloscope's channel configuration, with cleaning of values in properties setters as well as setting and reading them on and from the device's channel.

static `clean_coupling`(*coupling: Union[str, hvl_ccb.dev.tiepie.channel.TiePieOscilloscopeChannelCoupling]*) → `hvl_ccb.dev.tiepie.channel.TiePieOscilloscopeChannelCoupling`

static `clean_enabled`(*enabled: bool*) → `bool`

static `clean_input_range`(*input_range: Union[float, hvl_ccb.dev.tiepie.channel.TiePieOscilloscopeRange]*) → `hvl_ccb.dev.tiepie.channel.TiePieOscilloscopeRange`

static `clean_probe_offset`(*probe_offset: float*) → `float`

static `clean_trigger_enabled`(*trigger_enabled*)

static `clean_trigger_hysteresis`(*trigger_hysteresis: float*) → `float`

static `clean_trigger_kind`(*trigger_kind: Union[str, hvl_ccb.dev.tiepie.channel.TiePieOscilloscopeTriggerKind]*) → `hvl_ccb.dev.tiepie.channel.TiePieOscilloscopeTriggerKind`

static `clean_trigger_level`(*trigger_level: Union[int, float]*) → `float`

static `clean_trigger_level_mode`(*level_mode: Union[str, hvl_ccb.dev.tiepie.channel.TiePieOscilloscopeTriggerLevelMode]*) → `hvl_ccb.dev.tiepie.channel.TiePieOscilloscopeTriggerLevelMode`

property `coupling`: `hvl_ccb.dev.tiepie.channel.TiePieOscilloscopeChannelCoupling`

property `enabled`: `bool`

property **has_safe_ground**: bool

Check whether bound oscilloscope device has “safe ground” option

Returns bool: 1=safe ground available

property **input_range**: [hvl_ccb.dev.tiepie.channel.TiePieOscilloscopeRange](#)

property **probe_offset**: float

property **trigger_enabled**: bool

property **trigger_hysteresis**: float

property **trigger_kind**: [hvl_ccb.dev.tiepie.channel.TiePieOscilloscopeTriggerKind](#)

property **trigger_level**: float

property **trigger_level_mode**:

[hvl_ccb.dev.tiepie.channel.TiePieOscilloscopeTriggerLevelMode](#)

class **TiePieOscilloscopeChannelConfigLimits**(*osc_channel*:
[libtiepie.oscilloscopechannel.OscilloscopeChannel](#))

Bases: object

Default limits for oscilloscope channel parameters.

class **TiePieOscilloscopeChannelCoupling**(*value=<no_arg>*, *names=None*, *module=None*, *type=None*,
start=1, *boundary=None*)

Bases: [hvl_ccb.utils.enum.NameEnum](#)

An enumeration.

ACA = 8

ACV = 2

DCA = 4

DCV = 1

class **TiePieOscilloscopeRange**(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*,
boundary=None)

Bases: [hvl_ccb.utils.enum.NameEnum](#)

An enumeration.

EIGHTY_VOLT = 80

EIGHT_HUNDRED_MILLI_VOLT = 0.8

EIGHT_VOLT = 8

FORTY_VOLT = 40

FOUR_HUNDRED_MILLI_VOLT = 0.4

FOUR_VOLT = 4

TWENTY_VOLT = 20

TWO_HUNDRED_MILLI_VOLT = 0.2

TWO_VOLT = 2

static **suitable_range**(*value*)

```
class TiePieOscilloscopeTriggerKind(value=<no_arg>, names=None, module=None, type=None, start=1,
                                   boundary=None)
```

Bases: `hvl_ccb.utils.enum.NameEnum`

An enumeration.

ANY = 16

FALLING = 2

RISING = 1

RISING_OR_FALLING = 16

```
class TiePieOscilloscopeTriggerLevelMode(value=<no_arg>, names=None, module=None, type=None,
                                         start=1, boundary=None)
```

Bases: `hvl_ccb.utils.enum.NameEnum`

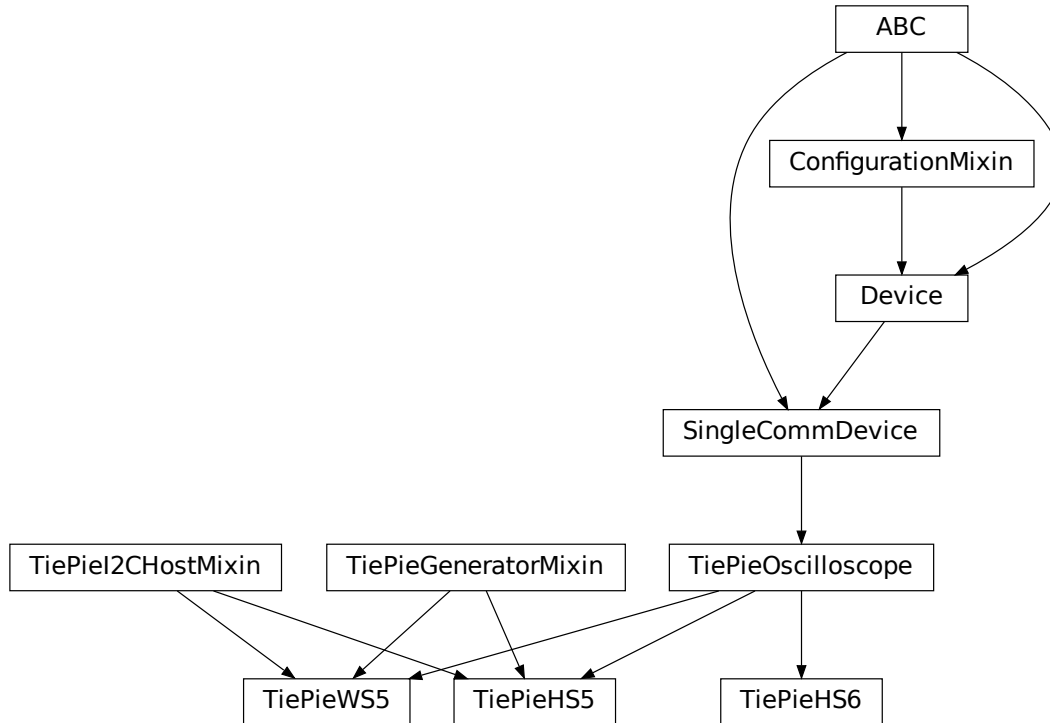
An enumeration.

ABSOLUTE = 2

RELATIVE = 1

UNKNOWN = 0

`hvl_ccb.dev.tiepie.device`



TiePie devices.

```
class TiePieHS5(com, dev_config)
```

```
    Bases: hvl_ccb.dev.tiepie.i2c.TiePieI2CHostMixin, hvl_ccb.dev.tiepie.generator.
            TiePieGeneratorMixin, hvl_ccb.dev.tiepie.oscilloscope.TiePieOscilloscope
```

TiePie HS5 device.

```
class TiePieHS6(com, dev_config)
```

```
    Bases: hvl_ccb.dev.tiepie.oscilloscope.TiePieOscilloscope
```

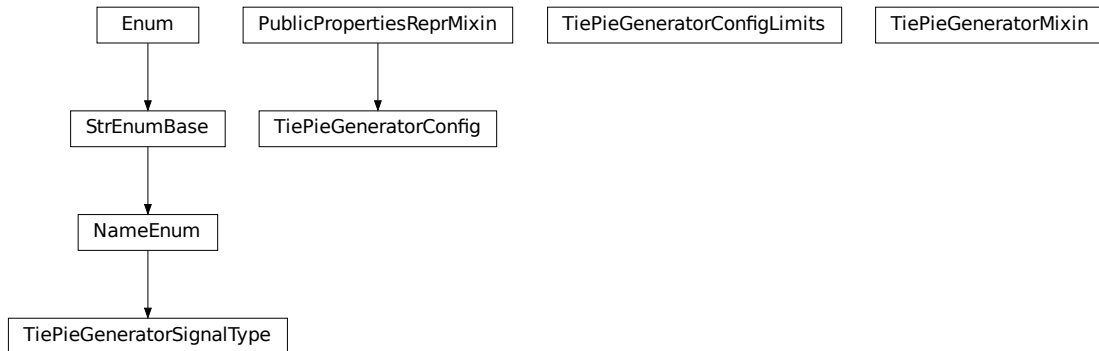
TiePie HS6 DIFF device.

```
class TiePieWS5(com, dev_config)
```

```
    Bases: hvl_ccb.dev.tiepie.i2c.TiePieI2CHostMixin, hvl_ccb.dev.tiepie.generator.
            TiePieGeneratorMixin, hvl_ccb.dev.tiepie.oscilloscope.TiePieOscilloscope
```

TiePie WS5 device.

`hvl_ccb.dev.tiepie.generator`



```
class TiePieGeneratorConfig(dev_gen: libtiepie.generator.Generator)
```

```
    Bases: hvl_ccb.dev.tiepie.utils.PublicPropertiesReprMixin
```

Generator's configuration with cleaning of values in properties setters.

```
    property amplitude: float
```

```
    clean_amplitude(amplitude: float) → float
```

```
    static clean_enabled(enabled: bool) → bool
```

```
    clean_frequency(frequency: float) → float
```

```
    clean_offset(offset: float) → float
```

```
    static clean_signal_type(signal_type: Union[int,  
                             hvl_ccb.dev.tiepie.generator.TiePieGeneratorSignalType]) →  
                             hvl_ccb.dev.tiepie.generator.TiePieGeneratorSignalType
```

```
    clean_waveform(waveform: numpy.ndarray[Any, numpy.dtype[numpy.typing._generic_alias.ScalarType]])  
        → numpy.ndarray[Any, numpy.dtype[numpy.typing._generic_alias.ScalarType]]
```

```
    property enabled: bool
```


property frequency: float

property offset: float

property signal_type: [hvl_ccb.dev.tiepie.generator.TiePieGeneratorSignalType](#)

property waveform: Optional[numpy.ndarray[Any,
numpy.dtype[numpy.typing._generic_alias.ScalarType]]]

class TiePieGeneratorConfigLimits(*dev_gen: libtiepie.generator.Generator*)

Bases: object

Default limits for generator parameters.

class TiePieGeneratorMixin(*com, dev_config*)

Bases: object

TiePie Generator sub-device.

A wrapper for the *libtiepie.generator.Generator* class. To be mixed in with *TiePieOscilloscope* base class.

config_gen: Optional[[hvl_ccb.dev.tiepie.generator.TiePieGeneratorConfig](#)]

Generator's dynamical configuration.

generator_start()

Start signal generation.

generator_stop()

Stop signal generation.

start() → *None*

Start the Generator.

stop() → *None*

Stop the generator.

class TiePieGeneratorSignalType(*value=<no_arg>, names=None, module=None, type=None, start=1,
boundary=None*)

Bases: [hvl_ccb.utils.enum.NameEnum](#)

An enumeration.

ARBITRARY = 32

DC = 8

NOISE = 16

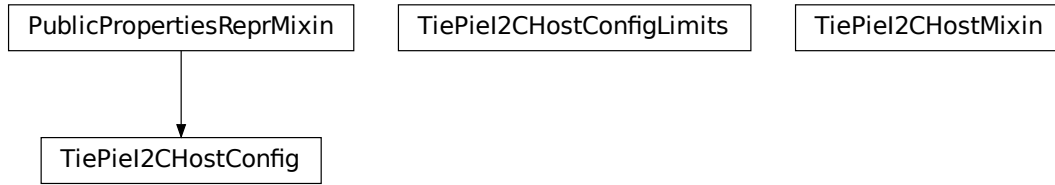
PULSE = 64

SINE = 1

SQUARE = 4

TRIANGLE = 2

UNKNOWN = 0

hvl_ccb.dev.tiepie.i2c

class TiePieI2CHostConfig(*dev_i2c: libtiepie.i2chost.I2CHost*)
Bases: [hvl_ccb.dev.tiepie.utils.PublicPropertiesReprMixin](#)
I2C Host's configuration with cleaning of values in properties setters.

class TiePieI2CHostConfigLimits(*dev_i2c: libtiepie.i2chost.I2CHost*)
Bases: object
Default limits for I2C host parameters.

class TiePieI2CHostMixin(*com, dev_config*)
Bases: object
TiePie I2CHost sub-device.
A wrapper for the *libtiepie.i2chost.I2CHost* class. To be mixed in with *TiePieOscilloscope* base class.

config_i2c: [Optional\[hvl_ccb.dev.tiepie.i2c.TiePieI2CHostConfig\]](#)
I2C host's dynamical configuration.

start() → *None*
Start the I2C Host.

stop() → *None*
Stop the I2C host.

static default_com_cls() → Type[hvl_ccb.comm.base.NullCommunicationProtocol]

Get the class for the default communication protocol used with this device.

Returns the type of the standard communication protocol for this device

force_trigger() → None

Forces the TiePie to trigger with a software sided trigger event.

Return None

Raises *TiePieError* – when device is not started or status of underlying device gives an error

is_measurement_data_ready() → bool

Reports if TiePie has data which is ready to collect

Returns if the data is ready to collect.

Raises *TiePieError* – when device is not started or status of underlying device gives an error

is_triggered() → bool

Reports if TiePie has triggered. Maybe data is not yet available. One can check with the function *is_measurement_data_ready()*.

Returns if a trigger event occurred

static list_devices() → libtiepie.devicelist.DeviceList

List available TiePie devices.

Returns libtiepie up to date list of devices

property n_channels

Number of channels in the oscilloscope.

Returns Number of channels.

start() → None

Start the oscilloscope.

start_measurement() → None

Start a measurement using set configuration.

Raises *TiePieError* – when device is not started or status of underlying device gives an error

stop() → None

Stop the oscilloscope.

class TiePieOscilloscopeAutoResolutionModes(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)

Bases: *hvl_ccb.utils.enum.NameEnum*

An enumeration.

ALL = 4

DISABLED = 1

NATIVEONLY = 2

UNKNOWN = 0

class TiePieOscilloscopeConfig(dev_osc: libtiepie.oscilloscope.Oscilloscope)

Bases: *hvl_ccb.dev.tiepie.utils.PublicPropertiesReprMixin*

Oscilloscope's configuration with cleaning of values in properties setters.

property auto_resolution_mode:

hvl_ccb.dev.tiepie.oscilloscope.TiePieOscilloscopeAutoResolutionModes

```

static clean_auto_resolution_mode(auto_resolution_mode: Union[int,
                                hvl_ccb.dev.tiepie.oscilloscope.TiePieOscilloscopeAutoResolutionModes])
                                →
                                hvl_ccb.dev.tiepie.oscilloscope.TiePieOscilloscopeAutoResolutionModes

clean_pre_sample_ratio(pre_sample_ratio: float) → float

clean_record_length(record_length: Union[int, float]) → int

static clean_resolution(resolution: Union[int,
                                hvl_ccb.dev.tiepie.oscilloscope.TiePieOscilloscopeResolution]) →
                                hvl_ccb.dev.tiepie.oscilloscope.TiePieOscilloscopeResolution

clean_sample_frequency(sample_frequency: float) → float

clean_trigger_timeout(trigger_timeout: Optional[Union[int, float]]) → float

property pre_sample_ratio: float
property record_length: int
property resolution: hvl_ccb.dev.tiepie.oscilloscope.TiePieOscilloscopeResolution
property sample_frequency: float
property trigger_timeout: Optional[float]

class TiePieOscilloscopeConfigLimits(dev_osc: libtiepie.oscilloscope.Oscilloscope)
    Bases: object

    Default limits for oscilloscope parameters.

class TiePieOscilloscopeResolution(value=<no_arg>, names=None, module=None, type=None, start=1,
                                   boundary=None)
    Bases: aenum.IntEnum

    An enumeration.

    EIGHT_BIT = 8
    FOURTEEN_BIT = 14
    SIXTEEN_BIT = 16
    TWELVE_BIT = 12

```

`hvl_ccb.dev.tiepie.utils`

PublicPropertiesReprMixin

```
class PublicPropertiesReprMixin
```

Bases: object

General purpose utility mixin that overwrites object representation to a one analogous to *dataclass* instances, but using public properties and their values instead of *fields*.

Module contents

This module is a wrapper around LibTiePie SDK devices; see <https://www.tiepie.com/en/libtiepie-sdk> .

The device classes adds simplifications for starting of the device (using serial number) and managing mutable configuration of both the device and oscilloscope's channels. This includes extra validation and typing hints support.

Extra installation

LibTiePie SDK library is available only on Windows and on Linux.

To use this LibTiePie SDK devices wrapper:

1. install the `hvl_ccb` package with a `tiepie` extra feature:

```
$ pip install "hvl_ccb[tiepie]"
```

this will install the Python bindings for the library.

2. install the library

- on Linux: follow instructions in <https://www.tiepie.com/en/libtiepie-sdk/linux> ;
- on Windows: the additional DLL is included in Python bindings package.

Troubleshooting

On a Windows system, if you encounter an `OSError` like this:

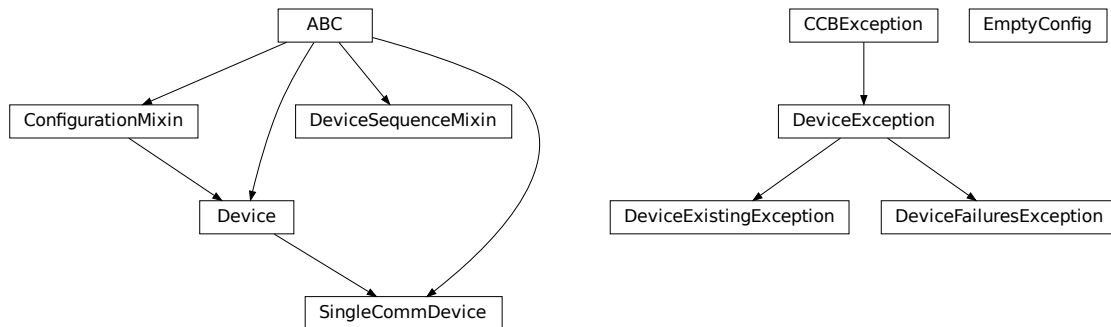
```
...  
    self._handle = _dlopen(self._name, mode)  
OSError: [WinError 126] The specified module could not be found
```

most likely the `python-libtiepie` package was installed in your `site-packages/` directory as a `python-libtiepie-*.egg` file via `python setup.py install` or `python setup.py develop` command. In such case uninstall the library and re-install it using `pip`:

```
$ pip uninstall python-libtiepie  
$ pip install python-libtiepie
```

This should create `libtiepie/` folder. Alternatively, manually move the folder `libtiepie/` from inside of the `.egg` archive file to the containing it `site-packages/` directory (PyCharm's Project tool window supports reading and extracting from `.egg` archives).

Submodules

hvl_ccb.dev.base

Module with base classes for devices.

class Device(*dev_config=None*)

Bases: [hvl_ccb.configuration.ConfigurationMixin](#), [abc.ABC](#)

Base class for devices. Implement this class for a concrete device, such as measurement equipment or voltage sources.

Specifies the methods to implement for a device.

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

abstract start() → *None*

Start or restart this Device. To be implemented in the subclass.

abstract stop() → *None*

Stop this Device. To be implemented in the subclass.

exception DeviceException

Bases: [hvl_ccb.exception.CCBException](#)

exception DeviceExistingException

Bases: [hvl_ccb.dev.base.DeviceException](#)

Exception to indicate that a device with that name already exists.

exception DeviceFailuresException(*failures: Dict[str, Exception], *args*)

Bases: [hvl_ccb.dev.base.DeviceException](#)

Exception to indicate that one or several devices failed.

failures: **Dict[str, Exception]**

A dictionary of named devices failures (exceptions).

class DeviceSequenceMixin(*devices: Dict[str, hvl_ccb.dev.base.Device]*)

Bases: [abc.ABC](#)

Mixin that can be used on a device or other classes to provide facilities for handling multiple devices in a sequence.

add_device(*name: str, device: hvl_ccb.dev.base.Device*) → *None*

Add a new device to the device sequence.

Parameters

- **name** – is the name of the device.
- **device** – is the instantiated Device object.

Raises *DeviceExistingException* –

devices_failed_start: Dict[str, *hvl_ccb.dev.base.Device*]

Dictionary of named device instances from the sequence for which the most recent *start()* attempt failed.

Empty if *stop()* was called last; cf. *devices_failed_stop*.

devices_failed_stop: Dict[str, *hvl_ccb.dev.base.Device*]

Dictionary of named device instances from the sequence for which the most recent *stop()* attempt failed.

Empty if *start()* was called last; cf. *devices_failed_start*.

get_device(*name: str*) → *hvl_ccb.dev.base.Device*

Get a device by name.

Parameters **name** – is the name of the device.

Returns the device object from this sequence.

get_devices() → List[Tuple[str, *hvl_ccb.dev.base.Device*]]

Get list of name, device pairs according to current sequence.

Returns A list of tuples with name and device each.

remove_device(*name: str*) → *hvl_ccb.dev.base.Device*

Remove a device from this sequence and return the device object.

Parameters **name** – is the name of the device.

Returns device object or *None* if such device was not in the sequence.

Raises **ValueError** – when device with given name was not found

start() → *None*

Start all devices in this sequence in their added order.

Raises *DeviceFailuresException* – if one or several devices failed to start

stop() → *None*

Stop all devices in this sequence in their reverse order.

Raises *DeviceFailuresException* – if one or several devices failed to stop

class **EmptyConfig**

Bases: object

Empty configuration dataclass that is the default configuration for a Device.

clean_values()

Cleans and enforces configuration values. Does nothing by default, but may be overridden to add custom configuration value checks.

force_value(*fieldname, value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field

- **value** – value to assign

is_configdataclass = True

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

class SingleCommDevice(com, dev_config=None)

Bases: [hvl_ccb.dev.base.Device](#), [abc.ABC](#)

Base class for devices with a single communication protocol.

property com

Get the communication protocol of this device.

Returns an instance of CommunicationProtocol subtype

abstract static default_com_cls() → Type[[hvl_ccb.comm.base.CommunicationProtocol](#)]

Get the class for the default communication protocol used with this device.

Returns the type of the standard communication protocol for this device

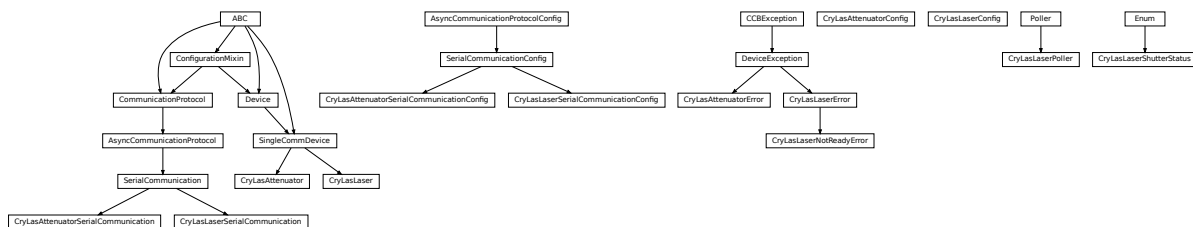
start() → None

Open the associated communication protocol.

stop() → None

Close the associated communication protocol.

hvl_ccb.dev.crylas



Device classes for a CryLas pulsed laser controller and a CryLas laser attenuator, using serial communication.

There are three modes of operation for the laser 1. Laser-internal hardware trigger (default): fixed to 20 Hz and max energy per pulse. 2. Laser-internal software trigger (for diagnosis only). 3. External trigger: required for arbitrary pulse energy or repetition rate. Switch to “external” on the front panel of laser controller for using option 3.

After switching on the laser with `laser_on()`, the system must stabilize for some minutes. Do not apply abrupt changes of pulse energy or repetition rate.

Manufacturer homepage: https://www.crylas.de/products/pulsed_laser.html

class CryLasAttenuator(*com, dev_config=None*)

Bases: [hvl_ccb.dev.base.SingleCommDevice](#)

Device class for the CryLas laser attenuator.

property attenuation: `Union[int, float]`

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

static default_com_cls()

Get the class for the default communication protocol used with this device.

Returns the type of the standard communication protocol for this device

set_attenuation(*percent: Union[int, float]*) \rightarrow *None*

Set the percentage of attenuated light (inverse of `set_transmission`). :param percent: percentage of attenuation, number between 0 and 100 :raises ValueError: if param percent not between 0 and 100 :raises SerialCommunicationIOError: when communication port is not opened :raises CryLasAttenuatorError: if the device does not confirm success

set_init_attenuation()

Sets the attenuation to its configured initial/default value

Raises [SerialCommunicationIOError](#) – when communication port is not opened

set_transmission(*percent: Union[int, float]*) \rightarrow *None*

Set the percentage of transmitted light (inverse of `set_attenuation`). :param percent: percentage of transmitted light :raises ValueError: if param percent not between 0 and 100 :raises SerialCommunicationIOError: when communication port is not opened :raises CryLasAttenuatorError: if the device does not confirm success

start() \rightarrow *None*

Open the com, apply the config value 'init_attenuation'

Raises [SerialCommunicationIOError](#) – when communication port cannot be opened

property transmission: `Union[int, float]`

class CryLasAttenuatorConfig(*init_attenuation: Union[int, float] = 0, response_sleep_time: Union[int, float] = 1*)

Bases: object

Device configuration dataclass for CryLas attenuator.

clean_values()

force_value(*fieldname, value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define `post_force_value` method with same signature as this method to do extra processing after `value` has been forced on `fieldname`.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

init_attenuation: `Union[int, float] = 0`

is_configdataclass = `True`

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

response_sleep_time: Union[int, float] = 1

exception CryLasAttenuatorError

Bases: [hvl_ccb.dev.base.DeviceException](#)

General error with the CryLas Attenuator.

class CryLasAttenuatorSerialCommunication(configuration)

Bases: [hvl_ccb.comm.serial.SerialCommunication](#)

Specific communication protocol implementation for the CryLas attenuator. Already predefines device-specific protocol parameters in config.

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

```
class CryLasAttenuatorSerialCommunicationConfig(terminator: bytes = b'', encoding: str = 'utf-8',
                                                encoding_error_handling: str = 'strict',
                                                wait_sec_read_text_nonempty: Union[int, float] =
                                                0.5, default_n_attempts_read_text_nonempty: int =
                                                10, port: Union[str, NoneType] = None, baudrate: int
                                                = 9600, parity: Union[str,
                                                hvl_ccb.comm.serial.SerialCommunicationParity] =
                                                <SerialCommunicationParity.NONE: 'N'>, stopbits:
                                                Union[int,
                                                hvl_ccb.comm.serial.SerialCommunicationStopbits] =
                                                <SerialCommunicationStopbits.ONE: 1>, bytesize:
                                                Union[int,
                                                hvl_ccb.comm.serial.SerialCommunicationBytesize]
                                                = <SerialCommunicationBytesize.EIGHTBITS: 8>,
                                                timeout: Union[int, float] = 3)
```

Bases: [hvl_ccb.comm.serial.SerialCommunicationConfig](#)

baudrate: int = 9600

Baudrate for CryLas attenuator is 9600 baud

bytesize: Union[int, [hvl_ccb.comm.serial.SerialCommunicationBytesize](#)] = 8

One byte is eight bits long

force_value(fieldname, value)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

parity: Union[str, [hvl_ccb.comm.serial.SerialCommunicationParity](#)] = 'N'

CryLas attenuator does not use parity

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

stopbits: Union[int, [hvl_ccb.comm.serial.SerialCommunicationStopbits](#)] = 1

CryLas attenuator uses one stop bit

terminator: bytes = b''

No terminator

timeout: Union[int, float] = 3

use 3 seconds timeout as default

class CryLasLaser(*com*, *dev_config=None*)

Bases: [hvl_ccb.dev.base.SingleCommDevice](#)

CryLas laser controller device class.

class AnswersShutter(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)

Bases: [aenum.Enum](#)

Standard answers of the CryLas laser controller to 'Shutter' command passed via *com*.

CLOSED = 'Shutter inaktiv'

OPENED = 'Shutter aktiv'

class AnswersStatus(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)

Bases: [aenum.Enum](#)

Standard answers of the CryLas laser controller to 'STATUS' command passed via *com*.

ACTIVE = 'STATUS: Laser active'

HEAD = 'STATUS: Head ok'

INACTIVE = 'STATUS: Laser inactive'

READY = 'STATUS: System ready'

```

    TEC1 = 'STATUS: TEC1 Regulation ok'
    TEC2 = 'STATUS: TEC2 Regulation ok'

class LaserStatus(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)
    Bases: aenum.Enum
    Status of the CryLas laser

    READY_ACTIVE = 2
    READY_INACTIVE = 1
    UNREADY_INACTIVE = 0
    property is_inactive
    property is_ready

class RepetitionRates(value=<no_arg>, names=None, module=None, type=None, start=1,
    boundary=None)
    Bases: aenum.IntEnum
    Repetition rates for the internal software trigger in Hz

    HARDWARE = 0
    SOFTWARE_INTERNAL_SIXTY = 60
    SOFTWARE_INTERNAL_TEN = 10
    SOFTWARE_INTERNAL_TWENTY = 20

ShutterStatus
    alias of hvl_ccb.dev.crylas.CryLasLaserShutterStatus

close_shutter() → None
    Close the laser shutter.

    Raises
        • SerialCommunicationIOError – when communication port is not opened
        • CryLasLaserError – if success is not confirmed by the device

static config_cls()
    Return the default configdataclass class.

    Returns a reference to the default configdataclass class

static default_com_cls()
    Get the class for the default communication protocol used with this device.

    Returns the type of the standard communication protocol for this device

get_pulse_energy_and_rate() → Tuple[int, int]
    Use the debug mode, return the measured pulse energy and rate.

    Returns (energy in micro joule, rate in Hz)

    Raises
        • SerialCommunicationIOError – when communication port is not opened
        • CryLasLaserError – if the device does not answer the query

laser_off() → None
    Turn the laser off.

```

Raises

- ***SerialCommunicationIOError*** – when communication port is not opened
- ***CryLasLaserError*** – if success is not confirmed by the device

laser_on() → *None*

Turn the laser on.

Raises

- ***SerialCommunicationIOError*** – when communication port is not opened
- ***CryLasLaserNotReadyError*** – if the laser is not ready to be turned on
- ***CryLasLaserError*** – if success is not confirmed by the device

open_shutter() → *None*

Open the laser shutter.

Raises

- ***SerialCommunicationIOError*** – when communication port is not opened
- ***CryLasLaserError*** – if success is not confirmed by the device

set_init_shutter_status() → *None*

Open or close the shutter, to match the configured shutter_status.

Raises

- ***SerialCommunicationIOError*** – when communication port is not opened
- ***CryLasLaserError*** – if success is not confirmed by the device

set_pulse_energy(*energy: int*) → *None*

Sets the energy of pulses (works only with external hardware trigger). Proceed with small energy steps, or the regulation may fail.

Parameters **energy** – energy in micro joule**Raises**

- ***SerialCommunicationIOError*** – when communication port is not opened
- ***CryLasLaserError*** – if the device does not confirm success

set_repetition_rate(*rate: Union[int, hvl_ccb.dev.crylas.CryLasLaser.RepetitionRates]*) → *None*

Sets the repetition rate of the internal software trigger.

Parameters **rate** – frequency (Hz) as an integer**Raises**

- ***ValueError*** – if rate is not an accepted value in RepetitionRates Enum
- ***SerialCommunicationIOError*** – when communication port is not opened
- ***CryLasLaserError*** – if success is not confirmed by the device

start() → *None*

Opens the communication protocol and configures the device.

Raises ***SerialCommunicationIOError*** – when communication port cannot be opened**stop()** → *None*

Stops the device and closes the communication protocol.

Raises

- **`SerialCommunicationIOError`** – if com port is closed unexpectedly
- **`CryLasLaserError`** – if `laser_off()` or `close_shutter()` fail

property target_pulse_energy

update_laser_status() → *None*

Update the laser status to *LaserStatus.NOT_READY* or *LaserStatus.INACTIVE* or *LaserStatus.ACTIVE*.

Note: laser never explicitly says that it is not ready (*LaserStatus.NOT_READY*) in response to ‘STATUS’ command. It only says that it is ready (heated-up and implicitly inactive/off) or active (on). If it’s not either of these then the answer is *Answers.HEAD*. Moreover, the only time the laser explicitly says that its status is inactive (*Answers.INACTIVE*) is after issuing a ‘LASER OFF’ command.

Raises **`SerialCommunicationIOError`** – when communication port is not opened

update_repetition_rate() → *None*

Query the laser repetition rate.

Raises

- **`SerialCommunicationIOError`** – when communication port is not opened
- **`CryLasLaserError`** – if success is not confirmed by the device

update_shutter_status() → *None*

Update the shutter status (OPENED or CLOSED)

Raises

- **`SerialCommunicationIOError`** – when communication port is not opened
- **`CryLasLaserError`** – if success is not confirmed by the device

update_target_pulse_energy() → *None*

Query the laser pulse energy.

Raises

- **`SerialCommunicationIOError`** – when communication port is not opened
- **`CryLasLaserError`** – if success is not confirmed by the device

wait_until_ready() → *None*

Block execution until the laser is ready

Raises **`CryLasLaserError`** – if the polling thread stops before the laser is ready

```
class CryLasLaserConfig(calibration_factor: Union[int, float] = 4.35, polling_period: Union[int, float] = 12,
                        polling_timeout: Union[int, float] = 300, auto_laser_on: bool = True,
                        init_shutter_status: Union[int, hvl_ccb.dev.crylas.CryLasLaserShutterStatus] =
                        CryLasLaserShutterStatus.CLOSED)
```

Bases: object

Device configuration dataclass for the CryLas laser controller.

ShutterStatus

alias of *hvl_ccb.dev.crylas.CryLasLaserShutterStatus*

auto_laser_on: bool = True

calibration_factor: Union[int, float] = 4.35

clean_values()

force_value(*fieldname*, *value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

init_shutter_status: Union[int, [hvl_ccb.dev.crylas.CryLasLaserShutterStatus](#)] = 0

is_configdataclass = True

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

polling_period: Union[int, float] = 12

polling_timeout: Union[int, float] = 300

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

exception CryLasLaserError

Bases: [hvl_ccb.dev.base.DeviceException](#)

General error with the CryLas Laser.

exception CryLasLaserNotReadyError

Bases: [hvl_ccb.dev.crylas.CryLasLaserError](#)

Error when trying to turn on the CryLas Laser before it is ready.

class CryLasLaserPoller(*spoll_handler: Callable*, *check_handler: Callable*, *check_laser_status_handler: Callable*, *polling_delay_sec: Union[int, float] = 0*, *polling_interval_sec: Union[int, float] = 1*, *polling_timeout_sec: Optional[Union[int, float]] = None*)

Bases: [hvl_ccb.dev.utils.Poller](#)

Poller class for polling the laser status until the laser is ready.

Raises

- [CryLasLaserError](#) – if the timeout is reached before the laser is ready
- [SerialCommunicationIOError](#) – when communication port is closed.

class CryLasLaserSerialCommunication(*configuration*)

Bases: [hvl_ccb.comm.serial.SerialCommunication](#)

Specific communication protocol implementation for the CryLas laser controller. Already predefines device-specific protocol parameters in config.

READ_TEXT_SKIP_PREFIXES = ('>', 'MODE:')

Prefixes of lines that are skipped when read from the serial port.

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

query(cmd: str, prefix: str, post_cmd: Optional[str] = None) → str

Send a command, then read the com until a line starting with prefix, or an empty line, is found. Returns the line in question.

Parameters

- **cmd** – query message to send to the device
- **prefix** – start of the line to look for in the device answer
- **post_cmd** – optional additional command to send after the query

Returns line in question as a string

Raises *SerialCommunicationIOError* – when communication port is not opened

query_all(cmd: str, prefix: str)

Send a command, then read the com until a line starting with prefix, or an empty line, is found. Returns a list of successive lines starting with prefix.

Parameters

- **cmd** – query message to send to the device
- **prefix** – start of the line to look for in the device answer

Returns line in question as a string

Raises *SerialCommunicationIOError* – when communication port is not opened

read() → str

Read first line of text from the serial port that does not start with any of *self.READ_TEXT_SKIP_PREFIXES*.

Returns String read from the serial port; '' if there was nothing to read.

Raises *SerialCommunicationIOError* – when communication port is not opened

```
class CryLasLaserSerialCommunicationConfig(terminator: bytes = b'\n', encoding: str = 'utf-8',
                                           encoding_error_handling: str = 'strict',
                                           wait_sec_read_text_nonempty: Union[int, float] = 0.5,
                                           default_n_attempts_read_text_nonempty: int = 10, port:
                                           Union[str, NoneType] = None, baudrate: int = 19200,
                                           parity: Union[str,
                                           hvl_ccb.comm.serial.SerialCommunicationParity] =
                                           <SerialCommunicationParity.NONE: 'N'>, stopbits:
                                           Union[int,
                                           hvl_ccb.comm.serial.SerialCommunicationStopbits] =
                                           <SerialCommunicationStopbits.ONE: 1>, bytesize:
                                           Union[int,
                                           hvl_ccb.comm.serial.SerialCommunicationBytesize] =
                                           <SerialCommunicationBytesize.EIGHTBITS: 8>, timeout:
                                           Union[int, float] = 10)
```

Bases: *hvl_ccb.comm.serial.SerialCommunicationConfig*

baudrate: `int = 19200`

Baudrate for CryLas laser is 19200 baud

bytesize: `Union[int, hvl_ccb.comm.serial.SerialCommunicationBytesize] = 8`

One byte is eight bits long

force_value(*fieldname*, *value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

classmethod **keys**() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod **optional_defaults**() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

parity: `Union[str, hvl_ccb.comm.serial.SerialCommunicationParity] = 'N'`

CryLas laser does not use parity

classmethod **required_keys**() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

stopbits: `Union[int, hvl_ccb.comm.serial.SerialCommunicationStopbits] = 1`

CryLas laser uses one stop bit

terminator: `bytes = b'\n'`

The terminator is LF

timeout: `Union[int, float] = 10`

use 10 seconds timeout as default (a long timeout is needed!)

class **CryLasLaserShutterStatus**(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)

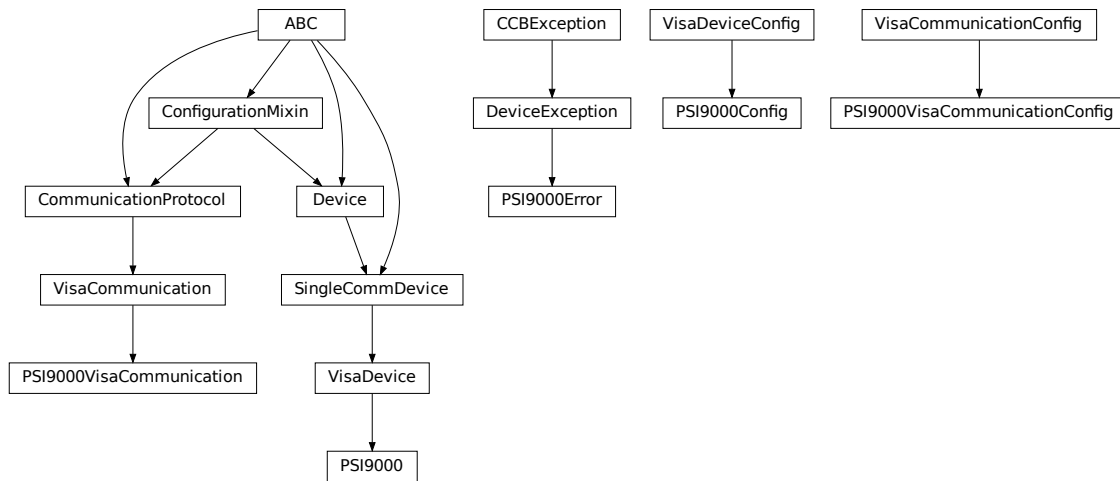
Bases: `aenum.Enum`

Status of the CryLas laser shutter

CLOSED = 0

OPENED = 1

hvl_ccb.dev.ea_psi9000



Device class for controlling a Elektro Automatik PSI 9000 power supply over VISA.

It is necessary that a backend for pyvisa is installed. This can be NI-Visa oder pyvisa-py (up to now, all the testing was done with NI-Visa)

```
class PSI9000(com: Union[hvl_ccb.dev.ea_psi9000.PSI9000VisaCommunication,
                        hvl_ccb.dev.ea_psi9000.PSI9000VisaCommunicationConfig, dict], dev_config:
                        Optional[Union[hvl_ccb.dev.ea_psi9000.PSI9000Config, dict]] = None)
```

Bases: `hvl_ccb.dev.visa.VisaDevice`

Elektro Automatik PSI 9000 power supply.

MS_NOMINAL_CURRENT = 2040

MS_NOMINAL_VOLTAGE = 80

SHUTDOWN_CURRENT_LIMIT = 0.1

SHUTDOWN_VOLTAGE_LIMIT = 0.1

check_master_slave_config() → *None*

Checks if the master / slave configuration and initializes if successful

Raises `PSI9000Error` – if master-slave configuration failed

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

static default_com_cls()

Return the default communication protocol for this device type, which is VisaCommunication.

Returns the VisaCommunication class

get_output() → bool

Reads the current state of the DC output of the source. Returns True, if it is enabled, false otherwise.

Returns the state of the DC output

get_system_lock() → bool

Get the current lock state of the system. The lock state is true, if the remote control is active and false, if not.

Returns the current lock state of the device

get_ui_lower_limits() → Tuple[float, float]

Get the lower voltage and current limits. A lower power limit does not exist.

Returns Umin in V, Imin in A

get_uip_upper_limits() → Tuple[float, float, float]

Get the upper voltage, current and power limits.

Returns Umax in V, Imax in A, Pmax in W

get_voltage_current_setpoint() → Tuple[float, float]

Get the voltage and current setpoint of the current source.

Returns Uset in V, Iset in A

measure_voltage_current() → Tuple[float, float]

Measure the DC output voltage and current

Returns Umeas in V, Imeas in A

set_lower_limits(*voltage_limit: Optional[float] = None, current_limit: Optional[float] = None*) → *None*

Set the lower limits for voltage and current. After writing the values a check is performed if the values are set correctly.

Parameters

- **voltage_limit** – is the lower voltage limit in V
- **current_limit** – is the lower current limit in A

Raises *PSI9000Error* – if the limits are out of range

set_output(*target_onstate: bool*) → *None*

Enables / disables the DC output.

Parameters **target_onstate** – enable or disable the output power

Raises *PSI9000Error* – if operation was not successful

set_system_lock(*lock: bool*) → *None*

Lock / unlock the device, after locking the control is limited to this class unlocking only possible when voltage and current are below the defined limits

Parameters **lock** – True: locking, False: unlocking

set_upper_limits(*voltage_limit: Optional[float] = None, current_limit: Optional[float] = None, power_limit: Optional[float] = None*) → *None*

Set the upper limits for voltage, current and power. After writing the values a check is performed if the values are set. If a parameter is left blank, the maximum configurable limit is set.

Parameters

- **voltage_limit** – is the voltage limit in V
- **current_limit** – is the current limit in A
- **power_limit** – is the power limit in W

Raises *PSI9000Error* – if limits are out of range

set_voltage_current(*volt: float, current: float*) → *None*

Set voltage and current setpoints.

After setting voltage and current, a check is performed if writing was successful.

Parameters

- **volt** – is the setpoint voltage: 0..81.6 V (1.02 * 0-80 V) (absolute max, can be smaller if limits are set)
- **current** – is the setpoint current: 0..2080.8 A (1.02 * 0 - 2040 A) (absolute max, can be smaller if limits are set)

Raises **PSI9000Error** – if the desired setpoint is out of limits

start() → *None*

Start this device.

stop() → *None*

Stop this device. Turns off output and lock, if enabled.

```
class PSI9000Config(spoll_interval: Union[int, float] = 0.5, spoll_start_delay: Union[int, float] = 2,
                  power_limit: Union[int, float] = 43500, voltage_lower_limit: Union[int, float] = 0.0,
                  voltage_upper_limit: Union[int, float] = 10.0, current_lower_limit: Union[int, float] = 0.0,
                  current_upper_limit: Union[int, float] = 2040.0, wait_sec_system_lock: Union[int, float]
                  = 0.5, wait_sec_settings_effect: Union[int, float] = 1, wait_sec_initialisation: Union[int,
                  float] = 2)
```

Bases: *hvl_ccb.dev.visa.VisaDeviceConfig*

Elektro Automatik PSI 9000 power supply device class. The device is communicating over a VISA TCP socket.

Using this power supply, DC voltage and current can be supplied to a load with up to 2040 A and 80 V (using all four available units in parallel). The maximum power is limited by the grid, being at 43.5 kW available through the CEE63 power socket.

clean_values() → *None*

Cleans and enforces configuration values. Does nothing by default, but may be overridden to add custom configuration value checks.

current_lower_limit: Union[int, float] = 0.0

Lower current limit in A, depending on the experimental setup.

current_upper_limit: Union[int, float] = 2040.0

Upper current limit in A, depending on the experimental setup.

force_value(*fieldname, value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

power_limit: Union[int, float] = 43500

Power limit in W depending on the experimental setup. With 3x63A, this is 43.5kW. Do not change this value, if you do not know what you are doing. There is no lower power limit.

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

voltage_lower_limit: Union[int, float] = 0.0

Lower voltage limit in V, depending on the experimental setup.

voltage_upper_limit: Union[int, float] = 10.0

Upper voltage limit in V, depending on the experimental setup.

wait_sec_initialisation: Union[int, float] = 2

wait_sec_settings_effect: Union[int, float] = 1

wait_sec_system_lock: Union[int, float] = 0.5

exception PSI9000Error

Bases: [hvl_ccb.dev.base.DeviceException](#)

Base error class regarding problems with the PSI 9000 supply.

class PSI9000VisaCommunication(configuration)

Bases: [hvl_ccb.comm.visa.VisaCommunication](#)

Communication protocol used with the PSI 9000 power supply.

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

class PSI9000VisaCommunicationConfig(*host: Union[str, ipaddress.IPv4Address, ipaddress.IPv6Address], interface_type: Union[str, [hvl_ccb.comm.visa.VisaCommunicationConfig.InterfaceType](#)] = InterfaceType.TCPIP_SOCKET, board: int = 0, port: int = 5025, timeout: int = 5000, chunk_size: int = 204800, open_timeout: int = 1000, write_termination: str = '\n', read_termination: str = '\n', visa_backend: str = ''*)

Bases: [hvl_ccb.comm.visa.VisaCommunicationConfig](#)

Visa communication protocol config dataclass with specification for the PSI 9000 power supply.

force_value(fieldname, value)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

Returns a list of strings containing all required keys.

[illegible]**Returns** FuGProbusVConfigRegisters

property current: *hvl_ccb.dev.fug.FuGProbusVSetRegisters*

Returns the registers for the current output

Returns

property current_monitor: *hvl_ccb.dev.fug.FuGProbusVMonitorRegisters*

Returns the registers for the current monitor.

A typically usage will be “self.current_monitor.value” to measure the output current

Returns

property di: *hvl_ccb.dev.fug.FuGProbusVDIRegisters*

Returns the registers for the digital inputs

Returns FuGProbusVDIRegisters

identify_device() → *None*

Identify the device nominal voltage and current based on its model number.

Raises *SerialCommunicationIOError* – when communication port is not opened

property max_current: *Union[int, float]*

Returns the maximal current which could provided within the test setup

Returns

property max_current_hardware: *Union[int, float]*

Returns the maximal current which could provided with the power supply

Returns

property max_voltage: *Union[int, float]*

Returns the maximal voltage which could provided within the test setup

Returns

property max_voltage_hardware: *Union[int, float]*

Returns the maximal voltage which could provided with the power supply

Returns

property on: *hvl_ccb.dev.fug.FuGProbusVDORegisters*

Returns the registers for the output switch to turn the output on or off

Returns FuGProbusVDORegisters

property outX0: *hvl_ccb.dev.fug.FuGProbusVDORegisters*

Returns the registers for the digital output X0

Returns FuGProbusVDORegisters

property outX1: *hvl_ccb.dev.fug.FuGProbusVDORegisters*

Returns the registers for the digital output X1

Returns FuGProbusVDORegisters

property outX2: *hvl_ccb.dev.fug.FuGProbusVDORegisters*

Returns the registers for the digital output X2

Returns FuGProbusVDORegisters

property outXCMD: *hvl_ccb.dev.fug.FuGProbusVDORegisters*

Returns the registers for the digital outputX-CMD

Returns FuGProbusVDORegisters

start(*max_voltage=0, max_current=0*) → *None*

Opens the communication protocol and configures the device.

Parameters

- **max_voltage** – Configure here the maximal permissible voltage which is allowed in the given experimental setup
- **max_current** – Configure here the maximal permissible current which is allowed in the given experimental setup

property voltage: *hvl_ccb.dev.fug.FuGProbusVSetRegisters*

Returns the registers for the voltage output

Returns

property voltage_monitor: *hvl_ccb.dev.fug.FuGProbusVMonitorRegisters*

Returns the registers for the voltage monitor.

A typically usage will be “self.voltage_monitor.value” to measure the output voltage

Returns

class FuGConfig(*wait_sec_stop_commands: Union[int, float] = 0.5*)

Bases: object

Device configuration dataclass for FuG power supplies.

clean_values()

force_value(*fieldname, value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

is_configdataclass = True

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

wait_sec_stop_commands: Union[int, float] = 0.5

Time to wait after subsequent commands during stop (in seconds)

```
class FuGDigitalVal(value)
```

```
    Bases: enum.IntEnum
```

```
    An enumeration.
```

```
    NO = 0
```

```
    OFF = 0
```

```
    ON = 1
```

```
    YES = 1
```

```
exception FuGError(*args, **kwargs)
```

```
    Bases: hvl\_ccb.dev.base.DeviceException
```

```
    Error with the FuG voltage source.
```

```
    errorcode: str
```

```
        Errorcode from the Probus, see documentation of Probus V chapter 5. Errors with three-digit errorcodes are thrown by this python module.
```

```
class FuErrorcodes(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)
```

```
    Bases: hvl\_ccb.utils.enum.NameEnum
```

```
    The power supply can return an errorcode. These errorcodes are handled by this class. The original errorcodes from the source are with one or two digits, see documentation of Probus V chapter 5. All three-digit errorcodes are from this python module.
```

```
    E0 = ('no error', 'standard response on each command')
```

```
    E1 = ('no data available', 'Customer tried to read from GPIB but there were no data prepared. (IBIG50 sent command ~T2 to ADDA)')
```

```
    E10 = ('unknown SCPI command', 'This SCPI command is not implemented')
```

```
    E100 = ('Command is not implemented', 'You tried to execute a command, which is not implemented or does not exist')
```

```
    E106 = ('The rampstate is a read-only register', 'You tried to write data to the register, which can only give you the status of the ramping.')
```

```
    E11 = ('not allowed Trigger-on-Talk', 'Not allowed attempt to Trigger-on-Talk (~T1) while ADDA was in addressable mode.')
```

```
    E115 = ('The given index to select a digital value is out of range', 'Only integer values between 0 and 1 are allowed.')
```

```
    E12 = ('invalid argument in ~Tn command', 'Only ~T1 and ~T2 is implemented.')
```

```
    E125 = ('The given index to select a ramp mode is out of range', 'Only integer values between 0 and 4 are allowed.')
```

```
    E13 = ('invalid N-value', 'Register > K8 contained an invalid value. Error code is output on an attempt to query data with ? or ~T1')
```

```
    E135 = ('The given index to select the readback channel is out of range', 'Only integer values between 0 and 6 are allowed.')
```

```
    E14 = ('register is write only', 'Some registers can only be writte to (i.e.> H0)')
```

```
    E145 = ('The given value for the AD-conversion is unknown', 'Valid values for the ad-conversion are integer values from "0" to "7".')
```

```
    E15 = ('string too long', 'i.e.serial number string too long during calibration')
```

E155 = ('The given value to select a polarity is out range.', 'The value should be 0 or 1.')

E16 = ('wrong checksum', 'checksum over command string was not correct, refer also to 4.4 of the Probus V documentation')

E165 = ('The given index to select the terminator string is out of range', '')

E2 = ('unknown register type', "No valid register type after '>'")

E206 = ('This status register is read-only', 'You tried to write data to this register, which can only give you the actual status of the corresponding digital output.')

E306 = ('The monitor register is read-only', 'You tried to write data to a monitor, which can only give you measured data.')

E4 = ('invalid argument', 'The argument of the command was rejected .i.e. malformed number')

E5 = ('argument out of range', 'i.e. setvalue higher than type value')

E504 = ('Empty string as response', 'The connection is broken.')

E505 = ('The returned register is not the requested.', 'Maybe the connection is overburden.')

E6 = ('register is read only', 'Some registers can only be read but not written to. (i.e. monitor registers)')

E666 = ('You cannot overwrite the most recent error in the interface of the power supply. But, well: You created an error anyway...', '')

E7 = ('Receive Overflow', 'Command string was longer than 50 characters.')

E8 = ('EEPROM is write protected', 'Write attempt to calibration data while the write protection switch was set to write protected.')

E9 = ('address error', 'A non addressed command was sent to ADDA while it was in addressable mode (and vice versa).')

raise_()

class FuGMonitorModes(*value*)

Bases: enum.IntEnum

An enumeration.

T1MS = 1

15 bit + sign, 1 ms integration time

T200MS = 6

typ. 19 bit + sign, 200 ms integration time

T20MS = 3

17 bit + sign, 20 ms integration time

T256US = 0

14 bit + sign, 256 us integration time

T40MS = 4

17 bit + sign, 40 ms integration time

T4MS = 2

15 bit + sign, 4 ms integration time

T800MS = 7
typ. 20 bit + sign, 800 ms integration time

T80MS = 5
typ. 18 bit + sign, 80 ms integration time

class **FuGPolarities**(*value*)

Bases: `enum.IntEnum`

An enumeration.

NEGATIVE = 1

POSITIVE = 0

class **FuGProbusIV**(*com*, *dev_config=None*)

Bases: `hvl_ccb.dev.base.SingleCommDevice`, `abc.ABC`

FuG Probus IV device class

Sends basic SCPI commands and reads the answer. Only the special commands and PROBUS IV instruction set is implemented.

command(*command*: `hvl_ccb.dev.fug.FuGProbusIVCommands`, *value=None*) → str

Parameters

- **command** – one of the commands given within `FuGProbusIVCommands`
- **value** – an optional value, depending on the command

Returns a String if a query was performed

static **config_cls**()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

static **default_com_cls**()

Get the class for the default communication protocol used with this device.

Returns the type of the standard communication protocol for this device

output_off() → *None*

Switch DC voltage output off.

reset() → *None*

Reset of the interface: All setvalues are set to zero

abstract **start**()

Open the associated communication protocol.

stop() → *None*

Close the associated communication protocol.

class **FuGProbusIVCommands**(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `hvl_ccb.utils.enum.NameEnum`

An enumeration.

ADMODE = ('S', (<enum 'FuGMonitorModes'>, <class 'int'>))

CURRENT = ('I', (<class 'int'>, <class 'float'>))

EXECUTE = ('X', None)

```
EXECUTEONX = ('G', (<enum 'FuGDigitalVal'>, <class 'int'>))
```

Wait for “X” to execute pending commands

```
ID = ('*IDN?', None)
```

```
OUTPUT = ('F', (<enum 'FuGDigitalVal'>, <class 'int'>))
```

```
POLARITY = ('P', (<enum 'FuGPolarities'>, <class 'int'>))
```

```
QUERY = ('?', None)
```

```
READBACKCHANNEL = ('N', (<enum 'FuGReadbackChannels'>, <class 'int'>))
```

```
RESET = ('=', None)
```

```
TERMINATOR = ('Y', (<enum 'FuGTerminators'>, <class 'int'>))
```

```
VOLTAGE = ('U', (<class 'int'>, <class 'float'>))
```

```
XOUTPUTS = ('R', <class 'int'>)
```

TODO: the possible values are limited to 0..13

```
class FuGProbusV(com, dev_config=None)
```

Bases: [hvl_ccb.dev.fug.FuGProbusIV](#)

FuG Probus V class which uses register based commands to control the power supplies

```
get_register(register: str) → str
```

get the value from a register

Parameters **register** – the register from which the value is requested

Returns the value of the register as a String

```
set_register(register: str, value: Union[int, float, str]) → None
```

generic method to set value to register

Parameters

- **register** – the name of the register to set the value
- **value** – which should be written to the register

```
class FuGProbusVConfigRegisters(fug, super_register: hvl_ccb.dev.fug.FuGProbusVRegisterGroups)
```

Bases: object

Configuration and Status values, acc. 4.2.5

```
property execute_on_x: hvl\_ccb.dev.fug.FuGDigitalVal
```

status of Execute-on-X

Returns FuGDigitalVal of the status

```
property most_recent_error: hvl\_ccb.dev.fug.FuErrorcodes
```

Reads the Error-Code of the most recent command

Return FuGError

Raises [FuGError](#) – if code is not “E0”

```
property readback_data: hvl\_ccb.dev.fug.FuGReadbackChannels
```

Preselection of readout data for Trigger-on-Talk

Returns index for the readback channel

```
property srq_mask: int
```

SRQ-Mask, Service-Request Enable status bits for SRQ 0: no SRQ Bit 2: SRQ on change of status to CC
Bit 1: SRQ on change to CV

Returns representative integer value

property srq_status: **str**

SRQ-Statusbyte output as a decimal number: Bit 2: PS is in CC mode Bit 1: PS is in CV mode

Returns representative string

property status: **str**

Statusbyte as a string of 0/1. Combined status (compatibel to Probus IV), MSB first: Bit 7: I-REG Bit 6: V-REG Bit 5: ON-Status Bit 4: 3-Reg Bit 3: X-Stat (polarity) Bit 2: Cal-Mode Bit 1: unused Bit 0: SEL-D

Returns string of 0/1

property terminator: *hvl_ccb.dev.fug.FuGTerminators*

Terminator character for answer strings from ADDA

Returns FuGTerminators

class FuGProbusVDIRegisters(*fug, super_register: hvl_ccb.dev.fug.FuGProbusVRegisterGroups*)

Bases: object

Digital Inputs acc. 4.2.4

property analog_control: *hvl_ccb.dev.fug.FuGDigitalVal*

Returns shows 1 if power supply is controlled by the analog interface

property calibration_mode: *hvl_ccb.dev.fug.FuGDigitalVal*

Returns shows 1 if power supply is in calibration mode

property cc_mode: *hvl_ccb.dev.fug.FuGDigitalVal*

Returns shows 1 if power supply is in CC mode

property cv_mode: *hvl_ccb.dev.fug.FuGDigitalVal*

Returns shows 1 if power supply is in CV mode

property digital_control: *hvl_ccb.dev.fug.FuGDigitalVal*

Returns shows 1 if power supply is digitally controlled

property on: *hvl_ccb.dev.fug.FuGDigitalVal*

Returns shows 1 if power supply ON

property reg_3: *hvl_ccb.dev.fug.FuGDigitalVal*

For special applications.

Returns input from bit 3-REG

property x_stat: *hvl_ccb.dev.fug.FuGPolarities*

Returns polarity of HVPS with polarity reversal

class FuGProbusVDORegisters(*fug, super_register: hvl_ccb.dev.fug.FuGProbusVRegisterGroups*)

Bases: object

Digital outputs acc. 4.2.2

property out: Union[int, *hvl_ccb.dev.fug.FuGDigitalVal*]

Status of the output according to the last setting. This can differ from the actual state if output should only pulse.

Returns FuGDigitalVal

property status: *hvl_ccb.dev.fug.FuGDigitalVal*

Returns the actual value of output. This can differ from the set value if pulse function is used.

Returns FuGDigitalVal

class FuGProbusVMonitorRegisters(*fug, super_register: hvl_ccb.dev.fug.FuGProbusVRegisterGroups*)

Bases: object

Analog monitors acc. 4.2.3

property adc_mode: *hvl_ccb.dev.fug.FuGMonitorModes*

The programmed resolution and integration time of the AD converter

Returns FuGMonitorModes

property value: float

Value from the monitor.

Returns a float value in V or A

property value_raw: float

uncalibrated raw value from AD converter

Returns float value from ADC

class FuGProbusVRegisterGroups(*value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None*)

Bases: *hvl_ccb.utils.enum.NameEnum*

An enumeration.

CONFIG = 'K'

INPUT = 'D'

MONITOR_I = 'M1'

MONITOR_V = 'M0'

OUTPUTONCMD = 'BON'

OUTPUTX0 = 'B0'

OUTPUTX1 = 'B1'

OUTPUTX2 = 'B2'

OUTPUTXCMD = 'BX'

SETCURRENT = 'S1'

SETVOLTAGE = 'S0'

class FuGProbusVSetRegisters(*fug, super_register: hvl_ccb.dev.fug.FuGProbusVRegisterGroups*)

Bases: object

Setvalue control acc. 4.2.1 for the voltage and the current output

property actualsetvalue: float

The actual valid set value, which depends on the ramp function.

Returns actual valid set value

property high_resolution: *hvl_ccb.dev.fug.FuGDigitalVal*

Status of the high resolution mode of the output.

Return 0 normal operation

Return 1 High Res. Mode

property rampmode: *hvl_ccb.dev.fug.FuGRampModes*

The set ramp mode to control the setvalue.

Returns the mode of the ramp as instance of FuGRampModes

property ramprate: **float**

The set ramp rate in V/s.

Returns ramp rate in V/s

property rampstate: *hvl_ccb.dev.fug.FuGDigitalVal*

Status of ramp function.

Return 0 if final setvalue is reached

Return 1 if still ramping up

property setvalue: **float**

For the voltage or current output this setvalue was programmed.

Returns the programmed setvalue

class FuGRampModes(*value*)

Bases: `enum.IntEnum`

An enumeration.

FOLLOWRAMP = 1

Follow the ramp up- and downwards

IMMEDIATELY = 0

Standard mode: no ramp

ONLYUPWARDSOFFTOZERO = 4

Follow the ramp up- and downwards, if output is OFF set value is zero

RAMPUPWARDS = 2

Follow the ramp only upwards, downwards immediately

SPECIALRAMPUPWARDS = 3

Follow a special ramp function only upwards

class FuGReadbackChannels(*value*)

Bases: `enum.IntEnum`

An enumeration.

CURRENT = 1

FIRMWARE = 5

RATEDCURRENT = 4

RATEDVOLTAGE = 3

SN = 6

STATUSBYTE = 2

VOLTAGE = 0

class `FuGSerialCommunication(configuration)`

Bases: `hvl_ccb.comm.serial.SerialCommunication`

Specific communication protocol implementation for FuG power supplies. Already predefines device-specific protocol parameters in config.

static `config_cls()`

Return the default configdataclass class.

Returns a reference to the default configdataclass class

query(*command: str*) → str

Send a command to the interface and handle the status message. Eventually raises an exception.

Parameters **command** – Command to send

Raises `FuGError` – if the connection is broken or the error from the power source itself

Returns Answer from the interface or empty string

class `FuGSerialCommunicationConfig(terminator: bytes = b'\n', encoding: str = 'utf-8',
encoding_error_handling: str = 'strict', wait_sec_read_text_nonempty:
Union[int, float] = 0.5, default_n_attempts_read_text_nonempty: int =
10, port: Union[str, NoneType] = None, baudrate: int = 9600, parity:
Union[str, hvl_ccb.comm.serial.SerialCommunicationParity] =
<SerialCommunicationParity.NONE: 'N'>, stopbits: Union[int,
hvl_ccb.comm.serial.SerialCommunicationStopbits] =
<SerialCommunicationStopbits.ONE: 1>, bytesize: Union[int,
hvl_ccb.comm.serial.SerialCommunicationBytesize] =
<SerialCommunicationBytesize.EIGHTBITS: 8>, timeout: Union[int,
float] = 3)`

Bases: `hvl_ccb.comm.serial.SerialCommunicationConfig`

baudrate: int = 9600

Baudrate for FuG power supplies is 9600 baud

bytesize: Union[int, hvl_ccb.comm.serial.SerialCommunicationBytesize] = 8

One byte is eight bits long

default_n_attempts_read_text_nonempty: int = 10

default number of attempts to read a non-empty text

force_value(fieldname, value)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

classmethod `keys()` → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod `optional_defaults()` → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

parity: Union[str, *hvl_ccb.comm.serial.SerialCommunicationParity*] = 'N'

FuG does not use parity

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

stopbits: Union[int, *hvl_ccb.comm.serial.SerialCommunicationStopbits*] = 1

FuG uses one stop bit

terminator: bytes = b'\n'

The terminator is LF

timeout: Union[int, float] = 3

use 3 seconds timeout as default

wait_sec_read_text_nonempty: Union[int, float] = 0.5

default time to wait between attempts of reading a non-empty text

class FuGTerminators(value)

Bases: enum.IntEnum

An enumeration.

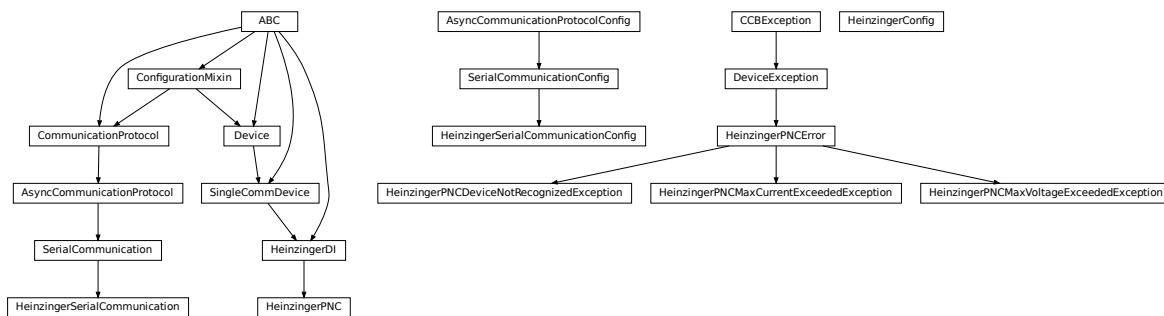
CR = 3

CRLF = 0

LF = 2

LFCR = 1

hvl_ccb.dev.heinzinger



Device classes for Heinzinger Digital Interface I/II and Heinzinger PNC power supply.

The Heinzinger Digital Interface I/II is used for many Heinzinger power units. Manufacturer homepage: <https://www.heinzinger.com/products/accessories-and-more/digital-interfaces/>

The Heinzinger PNC series is a series of high voltage direct current power supplies. The class HeinzingerPNC is tested with two PNChp 60000-1neg and a PNChp 1500-1neg. Check the code carefully before using it with other PNC devices, especially PNC3p or PNCcap. Manufacturer homepage: <https://www.heinzinger.com/products/high-voltage/universal-high-voltage-power-supplies/>

```
class HeinzingerConfig(default_number_of_recordings: Union[int,
                    hvl_ccb.dev.heinzinger.HeinzingerConfig.RecordingsEnum] = 1,
                    number_of_decimals: int = 6, wait_sec_stop_commands: Union[int, float] = 0.5)
```

Bases: object

Device configuration dataclass for Heinzinger power supplies.

```
class RecordingsEnum(value)
```

Bases: enum.IntEnum

An enumeration.

EIGHT = 8

FOUR = 4

ONE = 1

SIXTEEN = 16

TWO = 2

```
clean_values()
```

```
default_number_of_recordings: Union[int,
hvl_ccb.dev.heinzinger.HeinzingerConfig.RecordingsEnum] = 1
```

```
force_value(fieldname, value)
```

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

```
is_configdataclass = True
```

```
classmethod keys() → Sequence[str]
```

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

```
number_of_decimals: int = 6
```

```
classmethod optional_defaults() → Dict[str, object]
```

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

```
classmethod required_keys() → Sequence[str]
```

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

```
wait_sec_stop_commands: Union[int, float] = 0.5
```

Time to wait after subsequent commands during stop (in seconds)

```
class HeinzingerDI(com, dev_config=None)
```

Bases: *hvl_ccb.dev.base.SingleCommDevice*, *abc.ABC*

Heinzinger Digital Interface I/II device class

Sends basic SCPI commands and reads the answer. Only the standard instruction set from the manual is implemented.

class `OutputStatus(value)`

Bases: `enum.IntEnum`

Status of the voltage output

OFF = 0

ON = 1

UNKNOWN = -1

static `config_cls()`

Return the default configdataclass class.

Returns a reference to the default configdataclass class

static `default_com_cls()`

Get the class for the default communication protocol used with this device.

Returns the type of the standard communication protocol for this device

`get_current()` → float

Queries the set current of the Heinzinger PNC (not the measured current!).

Raises `SerialCommunicationIOError` – when communication port is not opened

`get_interface_version()` → str

Queries the version number of the digital interface.

Raises `SerialCommunicationIOError` – when communication port is not opened

`get_number_of_recordings()` → int

Queries the number of recordings the device is using for average value calculation.

Returns int number of recordings

Raises `SerialCommunicationIOError` – when communication port is not opened

`get_serial_number()` → str

Ask the device for its serial number and returns the answer as a string.

Returns string containing the device serial number

Raises `SerialCommunicationIOError` – when communication port is not opened

`get_voltage()` → float

Queries the set voltage of the Heinzinger PNC (not the measured voltage!).

Raises `SerialCommunicationIOError` – when communication port is not opened

`measure_current()` → float

Ask the Device to measure its output current and return the measurement result.

Returns measured current as float

Raises `SerialCommunicationIOError` – when communication port is not opened

`measure_voltage()` → float

Ask the Device to measure its output voltage and return the measurement result.

Returns measured voltage as float

Raises `SerialCommunicationIOError` – when communication port is not opened

output_off() → *None*

Switch DC voltage output off and updates the output status.

Raises **SerialCommunicationIOError** – when communication port is not opened

output_on() → *None*

Switch DC voltage output on and updates the output status.

Raises **SerialCommunicationIOError** – when communication port is not opened

property output_status: *hvl_ccb.dev.heinzinger.HeinzingerDI.OutputStatus*

reset_interface() → *None*

Reset of the digital interface; only Digital Interface I: Power supply is switched to the Local-Mode (Manual operation)

Raises **SerialCommunicationIOError** – when communication port is not opened

set_current(value: Union[int, float]) → *None*

Sets the output current of the Heinzinger PNC to the given value.

Parameters value – current expressed in *self.unit_current*

Raises **SerialCommunicationIOError** – when communication port is not opened

set_number_of_recordings(value: Union[int, *hvl_ccb.dev.heinzinger.HeinzingerConfig.RecordingsEnum*]) → *None*

Sets the number of recordings the device is using for average value calculation. The possible values are 1, 2, 4, 8 and 16.

Raises **SerialCommunicationIOError** – when communication port is not opened

set_voltage(value: Union[int, float]) → *None*

Sets the output voltage of the Heinzinger PNC to the given value.

Parameters value – voltage expressed in *self.unit_voltage*

Raises **SerialCommunicationIOError** – when communication port is not opened

abstract start()

Opens the communication protocol.

Raises **SerialCommunicationIOError** – when communication port cannot be opened.

stop() → *None*

Stop the device. Closes also the communication protocol.

class HeinzingerPNC(com, dev_config=None)

Bases: *hvl_ccb.dev.heinzinger.HeinzingerDI*

Heinzinger PNC power supply device class.

The power supply is controlled over a Heinzinger Digital Interface I/II

class UnitCurrent(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)

Bases: *hvl_ccb.utils.enum.AutoNumberNameEnum*

An enumeration.

A = 3

UNKNOWN = 1

mA = 2

```
class UnitVoltage(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)
    Bases: hvl_ccb.utils.enum.AutoNumberNameEnum

    An enumeration.

    UNKNOWN = 1
    V = 2
    kV = 3

identify_device() → None
    Identify the device nominal voltage and current based on its serial number.

    Raises SerialCommunicationIOError – when communication port is not opened

property max_current: Union[int, float]
property max_current_hardware: Union[int, float]
property max_voltage: Union[int, float]
property max_voltage_hardware: Union[int, float]

set_current(value: Union[int, float]) → None
    Sets the output current of the Heinzinger PNC to the given value.

    Parameters value – current expressed in self.unit_current

    Raises SerialCommunicationIOError – when communication port is not opened

set_voltage(value: Union[int, float]) → None
    Sets the output voltage of the Heinzinger PNC to the given value.

    Parameters value – voltage expressed in self.unit_voltage

    Raises SerialCommunicationIOError – when communication port is not opened

start() → None
    Opens the communication protocol and configures the device.

property unit_current: hvl_ccb.dev.heinzinger.HeinzingerPNC.UnitCurrent
property unit_voltage: hvl_ccb.dev.heinzinger.HeinzingerPNC.UnitVoltage

exception HeinzingerPNCDeviceNotRecognizedException
    Bases: hvl_ccb.dev.heinzinger.HeinzingerPNCError

    Error indicating that the serial number of the device is not recognized.

exception HeinzingerPNCError
    Bases: hvl_ccb.dev.base.DeviceException

    General error with the Heinzinger PNC voltage source.

exception HeinzingerPNCMaxCurrentExceededException
    Bases: hvl_ccb.dev.heinzinger.HeinzingerPNCError

    Error indicating that program attempted to set the current to a value exceeding ‘max_current’.

exception HeinzingerPNCMaxVoltageExceededException
    Bases: hvl_ccb.dev.heinzinger.HeinzingerPNCError

    Error indicating that program attempted to set the voltage to a value exceeding ‘max_voltage’.

class HeinzingerSerialCommunication(configuration)
    Bases: hvl_ccb.comm.serial.SerialCommunication
```

Specific communication protocol implementation for Heinzinger power supplies. Already predefines device-specific protocol parameters in config.

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

```
class HeinzingerSerialCommunicationConfig(terminator: bytes = b'\n', encoding: str = 'utf-8',
                                         encoding_error_handling: str = 'strict',
                                         wait_sec_read_text_nonempty: Union[int, float] = 0.5,
                                         default_n_attempts_read_text_nonempty: int = 40, port:
                                         Union[str, NoneType] = None, baudrate: int = 9600, parity:
                                         Union[str, hvl_ccb.comm.serial.SerialCommunicationParity]
                                         = <SerialCommunicationParity.NONE: 'N'>, stopbits:
                                         Union[int,
                                         hvl_ccb.comm.serial.SerialCommunicationStopbits] =
                                         <SerialCommunicationStopbits.ONE: 1>, bytesize:
                                         Union[int,
                                         hvl_ccb.comm.serial.SerialCommunicationBytesize] =
                                         <SerialCommunicationBytesize.EIGHTBITS: 8>, timeout:
                                         Union[int, float] = 3)
```

Bases: `hvl_ccb.comm.serial.SerialCommunicationConfig`

baudrate: int = 9600

Baudrate for Heinzinger power supplies is 9600 baud

bytesize: Union[int, hvl_ccb.comm.serial.SerialCommunicationBytesize] = 8

One byte is eight bits long

default_n_attempts_read_text_nonempty: int = 40

increased to 40 default number of attempts to read a non-empty text

force_value(fieldname, value)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

parity: Union[str, hvl_ccb.comm.serial.SerialCommunicationParity] = 'N'

Heinzinger does not use parity

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

stopbits: `Union[int, hvl_ccb.comm.serial.SerialCommunicationStopbits] = 1`

Heinzinger uses one stop bit

terminator: `bytes = b'\n'`

The terminator is LF

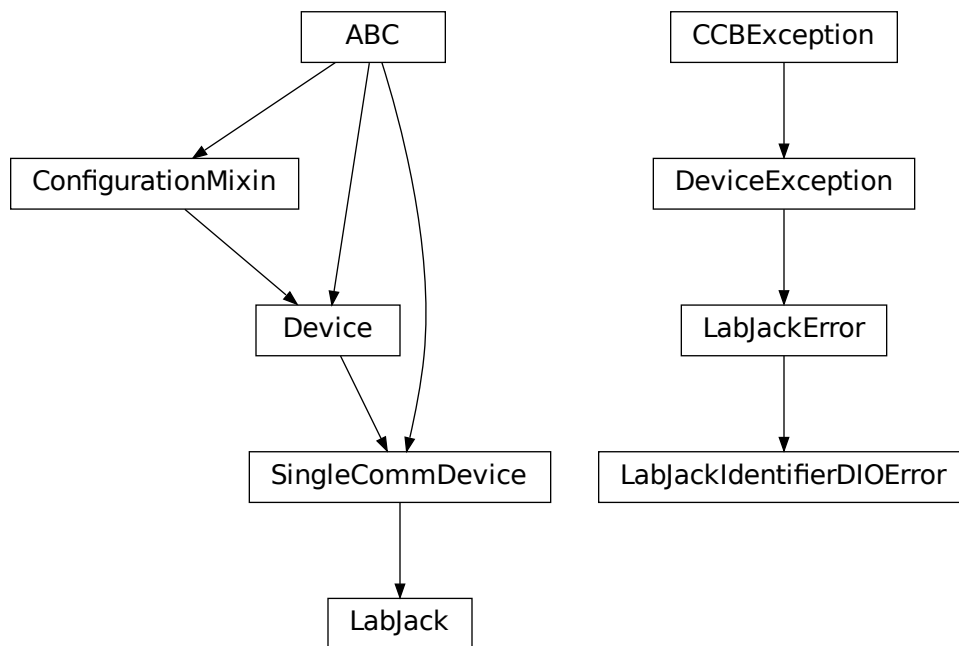
timeout: `Union[int, float] = 3`

use 3 seconds timeout as default

wait_sec_read_text_nonempty: `Union[int, float] = 0.5`

default time to wait between attempts of reading a non-empty text

[hvl_ccb.dev.labjack](#)



LabJack T-series devices wrapper around the LabJack's LJM Library; see <https://labjack.com/ljm> . The wrapper was originally developed and tested for a LabJack T7-PRO device.

A

Extra installation

To use this LabJack T-series devices wrapper:

1. install the `hvl_ccb` package with a `labjack` extra feature:

```
$ pip install "hvl_ccb[labjack]"
```

this will install the Python bindings for the library.

2. install the library - follow instruction in <https://labjack.com/support/software/installers/ljm> .

class LabJack(*com*, *dev_config=None*)

Bases: `hvl_ccb.dev.base.SingleCommDevice`

LabJack Device.

This class is tested with a LabJack T7-Pro and should also work with T4 and T7 devices communicating through the LJM Library. Other or older hardware versions and variants of LabJack devices are not supported.

class AInRange(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `hvl_ccb.utils.enum.StrEnumBase`

An enumeration.

ONE = 1.0

ONE_HUNDREDTH = 0.01

ONE_TENTH = 0.1

TEN = 10.0

property value: float

class BitLimit(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `aenum.IntEnum`

Maximum integer values for clock settings

THIRTY_TWO_BIT = 4294967295

class CalMicroAmpere(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `aenum.Enum`

Pre-defined microampere (uA) values for calibration current source query.

TEN = '10uA'

TWO_HUNDRED = '200uA'

class CjcType(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `hvl_ccb.utils.enum.NameEnum`

CJC slope and offset

internal = (1, 0)

lm34 = (55.56, 255.37)

class ClockFrequency(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `aenum.IntEnum`

Available clock frequencies, in Hz

```
FIVE_MHZ = 5000000
FORTY_MHZ = 40000000
MAXIMUM = 80000000
MINIMUM = 312500
TEN_MHZ = 10000000
TWELVE_HUNDRED_FIFTY_KHZ = 1250000
TWENTY_FIVE_HUNDRED_KHZ = 2500000
TWENTY_MHZ = 20000000
```

DIOChannel

alias of `hvl_ccb._dev.labjack.TSeriesDIOChannel`

```
class DIOStatus(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)
```

Bases: `aenum.IntEnum`

State of a digital I/O channel.

```
HIGH = 1
```

```
LOW = 0
```

```
class DeviceType(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)
```

Bases: `hvl_ccb.utils.enum.AutoNumberNameEnum`

LabJack device types.

Can be also looked up by ambiguous Product ID (*p_id*) or by instance name: ``python LabJackDeviceType(4) is LabJackDeviceType('T4')``

```
ANY = 1
```

```
T4 = 2
```

```
T7 = 3
```

```
T7_PRO = 4
```

```
classmethod get_by_p_id(p_id: int) → Union[hvl_ccb._dev.labjack.DeviceType,
                                             List[hvl_ccb._dev.labjack.DeviceType]]
```

Get LabJack device type instance via LabJack product ID.

Note: Product ID is not unambiguous for LabJack devices.

Parameters *p_id* – Product ID of a LabJack device

Returns Instance or list of instances of *LabJackDeviceType*

Raises **ValueError** – when Product ID is unknown

```
class TemperatureUnit(value=<no_arg>, names=None, module=None, type=None, start=1,
                       boundary=None)
```

Bases: `hvl_ccb.utils.enum.NameEnum`

Temperature unit (to be returned)

```
C = 1
```

```
F = 2
```

```
K = 0
```

```
class ThermocoupleType(value=<no_arg>, names=None, module=None, type=None, start=1,
                        boundary=None)
```

Bases: `hvl_ccb.utils.enum.NameEnum`

Thermocouple type; NONE means disable thermocouple mode.

C = 30

E = 20

J = 21

K = 22

NONE = 0

PT100 = 40

PT1000 = 42

PT500 = 41

R = 23

S = 25

T = 24

```
config_high_pulse(address: Union[str, hvl_ccb._dev.labjack.TSeriesDIOChannel], t_start: Union[int,
float], t_width: Union[int, float], n_pulses: int = 1) → None
```

Configures one FIO channel to send a timed HIGH pulse. Configure multiple channels to send pulses with relative timing accuracy. Times have a maximum resolution of 1e-7 seconds @ 10 MHz. :param address: FIO channel: [T7] FIO0;2;3;4;5. [T4] FIO6;7. :raises LabJackError if address is not supported. :param t_start: pulse start time in seconds. :raises ValueError: if t_start is negative or would exceed the clock period. :param t_width: duration of high pulse, in seconds. :raises ValueError: if t_width is negative or would exceed the clock period. :param n_pulses: number of pulses to be sent; single pulse default. :raises TypeError if n_pulses is not of type int. :raises Value Error if n_pulses is negative or exceeds the 32bit limit.

```
static default_com_cls()
```

Get the class for the default communication protocol used with this device.

Returns the type of the standard communication protocol for this device

```
disable_pulses(*addresses: Optional[Union[str, hvl_ccb._dev.labjack.TSeriesDIOChannel]]) → None
```

Disable previously configured pulse channels. :param addresses: tuple of FIO addresses. All channels disabled if no argument is passed.

```
enable_clock(clock_enabled: bool) → None
```

Enable/disable LabJack clock to configure or send pulses. :param clock_enabled: True -> enable, False -> disable. :raises TypeError: if clock_enabled is not of type bool

```
get_ain(*channels: int) → Union[float, Sequence[float]]
```

Read currently measured value (voltage, resistance, ...) from one or more of analog inputs.

Parameters channels – AIN number or numbers (0..254)

Returns the read value (voltage, resistance, ...) as *float* or *tuple* of them in case multiple channels given

```
get_cal_current_source(name: Union[str, CalMicroAmpere]) → float
```

This function will return the calibration of the chosen current source, this is not a measurement!

The value was stored during fabrication.

Parameters **name** – ‘200uA’ or ‘10uA’ current source

Returns calibration of the chosen current source in ampere

get_clock() → Dict[str, Union[int, float]]

Return clock settings read from LabJack.

get_digital_input(*address: Union[str, hvl_ccb._dev.labjack.TSeriesDIOChannel]*) → *hvl_ccb.dev.labjack.LabJack.DIOStatus*

Get the value of a digital input.

allowed names for T7 (Pro): FIO0 - FIO7, EIO0 - EIO 7, CIO0- CIO3, MIO0 - MIO2 :param address: name of the output -> ‘FIO0’ :return: HIGH when *address* DIO is high, and LOW when *address* DIO is low

get_product_id() → int

This function returns the product ID reported by the connected device.

Attention: returns 7 for both T7 and T7-Pro devices!

Returns integer product ID of the device

get_product_name(*force_query_id=False*) → str

This function will return the product name based on product ID reported by the device.

Attention: returns “T7” for both T7 and T7-Pro devices!

Parameters **force_query_id** – boolean flag to force *get_product_id* query to device instead of using cached device type from previous queries.

Returns device name string, compatible with *LabJack.DeviceType*

get_product_type(*force_query_id: bool = False*) → *hvl_ccb._dev.labjack.DeviceType*

This function will return the device type based on reported device type and in case of unambiguity based on configuration of device’s communication protocol (e.g. for “T7” and “T7_PRO” devices), or, if not available first matching.

Parameters **force_query_id** – boolean flag to force *get_product_id* query to device instead of using cached device type from previous queries.

Returns *DeviceType* instance

Raises *LabJackIdentifierDIOError* – when read Product ID is unknown

get_sbus_rh(*number: int*) → float

Read the relative humidity value from a serial SBUS sensor.

Parameters **number** – port number (0..22)

Returns relative humidity in %RH

get_sbus_temp(*number: int*) → float

Read the temperature value from a serial SBUS sensor.

Parameters **number** – port number (0..22)

Returns temperature in Kelvin

get_serial_number() → int

Returns the serial number of the connected LabJack.

Returns Serial number.

read_resistance(*channel: int*) → float

Read resistance from specified channel.

Parameters `channel` – channel with resistor

Returns resistance value with 2 decimal places

read_thermocouple(*pos_channel: int*) → float

Read the temperature of a connected thermocouple.

Parameters `pos_channel` – is the AIN number of the positive pin

Returns temperature in specified unit

send_pulses(**addresses: Union[str, hvl_ccb._dev.labjack.TSeriesDIOChannel]*) → *None*

Sends pre-configured pulses for specified addresses. :param addresses: tuple of FIO addresses :raises LabJackError if an address has not been configured.

set_ain_differential(*pos_channel: int, differential: bool*) → *None*

Sets an analog input to differential mode or not. T7-specific: For base differential channels, positive must be even channel from 0-12 and negative must be positive+1. For extended channels 16-127, see Mux80 datasheet.

Parameters

- `pos_channel` – is the AIN number (0..12)
- `differential` – True or False

Raises *LabJackError* – if parameters are unsupported

set_ain_range(*channel: int, vrange: Union[Real, AInRange]*) → *None*

Set the range of an analog input port.

Parameters

- `channel` – is the AIN number (0..254)
- `vrange` – is the voltage range to be set

set_ain_resistance(*channel: int, vrange: Union[Real, AInRange], resolution: int*) → *None*

Set the specified channel to resistance mode. It utilized the 200uA current source of the LabJack.

Parameters

- `channel` – channel that should measure the resistance
- `vrange` – voltage range of the channel
- `resolution` – resolution index of the channel T4: 0-5, T7: 0-8, T7-Pro 0-12

set_ain_resolution(*channel: int, resolution: int*) → *None*

Set the resolution index of an analog input port.

Parameters

- `channel` – is the AIN number (0..254)
- `resolution` – is the resolution index within 0...`get_product_type().ain_max_resolution` range; 0 will set the resolution index to default value.

set_ain_thermocouple(*pos_channel: int, thermocouple: Union[None, str, ThermocoupleType], cjc_address: int = 60050, cjc_type: Union[str, CjcType] = CjcType.internal, vrange: Union[Real, AInRange] = AInRange.ONE_HUNDREDTH, resolution: int = 10, unit: Union[str, TemperatureUnit] = TemperatureUnit.K*) → *None*

Set the analog input channel to thermocouple mode.

Parameters

- `pos_channel` – is the analog input channel of the positive part of the differential pair

- **thermocouple** – None to disable thermocouple mode, or string specifying the thermocouple type
- **cjc_address** – modbus register address to read the CJC temperature
- **cjc_type** – determines cjc slope and offset, ‘internal’ or ‘lm34’
- **vrange** – measurement voltage range
- **resolution** – resolution index (T7-Pro: 0-12)
- **unit** – is the temperature unit to be returned (‘K’, ‘C’ or ‘F’)

Raises **LabJackError** – if parameters are unsupported

set_analog_output(*channel: int, value: Union[int, float]*) → *None*

Set the voltage of a analog output port

Parameters

- **channel** – DAC channel number 1/0
- **value** – The output voltage value 0-5 Volts int/float

set_clock(*clock_frequency: Union[Number, ClockFrequency] = 10000000, clock_period: Number = 1*) → *None*

Configure LabJack clock for pulse out feature. :param clock_frequency: clock frequency in Hz; default 10 MHz for base 10. :raises ValueError: if clock_frequency is not allowed (see ClockFrequency). :param clock_period: clock roll time in seconds; default 1s, 0 for max. :raises ValueError: if clock_period exceeds the 32bit tick limit. Clock period determines pulse spacing when using multi-pulse settings. Ensure period exceeds maximum intended pulse end time.

set_digital_output(*address: str, state: Union[int, DIOStatus]*) → *None*

Set the value of a digital output.

Parameters

- **address** – name of the output -> ‘FIO0’
- **state** – state of the output -> *DIOStatus* instance or corresponding *int* value

start() → *None*

Start the Device.

stop() → *None*

Stop the Device.

exception LabJackError

Bases: *hvl_ccb.dev.base.DeviceException*

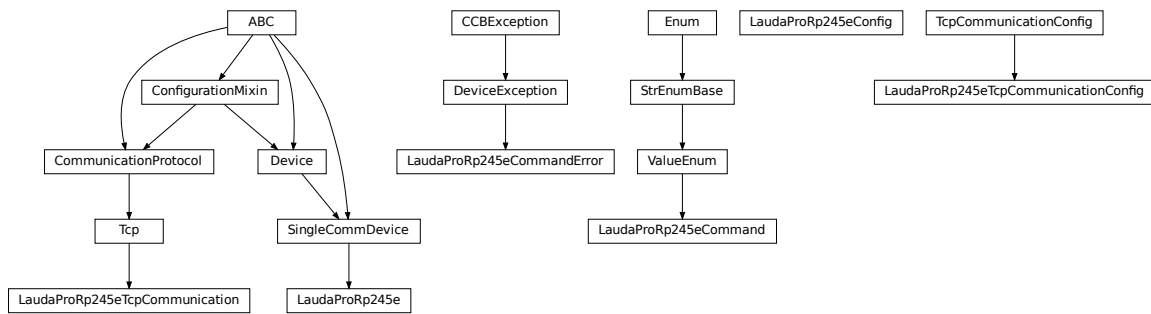
General Exception for the LabJack device.

exception LabJackIdentifierDIOError

Bases: *hvl_ccb.dev.labjack.LabJackError*

Error indicating a wrong DIO identifier

hvl_ccb.dev.lauda



Device class for controlling a Lauda PRO RP245E, circulation chiller over TCP.

class `LaudaProRp245e`(*com, dev_config=None*)

Bases: `hvl_ccb.dev.base.SingleCommDevice`

Lauda RP245E circulation chiller class.

static `config_cls()` → `Type[hvl_ccb.dev.lauda.LaudaProRp245eConfig]`

Return the default configdataclass class.

Returns a reference to the default configdataclass class

continue_ramp() → `str`

Continue current ramp program.

Returns reply of the device to the last call of “query”

static `default_com_cls()` → `Type[hvl_ccb.dev.lauda.LaudaProRp245eTcpCommunication]`

Get the class for the default communication protocol used with this device.

Returns the type of the standard communication protocol for this device

get_bath_temp() → `float`

:return : float value of measured lauda bath temp in °C

get_device_type() → `str`

:return : Connected Lauda device type (for connection/com test)

pause() → `str`

Stop temperature control and pump.

Returns reply of the device to the last call of “query”

pause_ramp() → `str`

Pause current ramp program.

Returns reply of the device to the last call of “query”

reset_ramp() → `str`

Delete all segments from current ramp program.

Returns reply of the device to the last call of “query”

run() → `str`

Start temperature control & pump.

Returns reply of the device to the last call of “query”

set_control_mode(*mod: Union[int, hvl_ccb.dev.lauda.LaudaProRp245eConfig.ExtControlModeEnum]* = *ExtControlModeEnum.INTERNAL*) → str

Define control mode. 0 = INTERNAL (control bath temp), 1 = EXPT100 (pt100 attached to chiller), 2 = ANALOG, 3 = SERIAL, 4 = USB, 5 = ETH (to be used when passing the ext. temp. via ethernet) (temperature then needs to be passed every second, when not using options 3, 4, or 5)

Parameters **mod** – temp control mode (control internal temp or external temp).

Returns reply of the device to the last call of “query” (“OK”, if command was recognized”)

set_external_temp(*external_temp: float = 20.0*) → str

Pass value of external controlled temperature. Should be done every second, when control of external temperature is active. Has to be done right before control of external temperature is activated.

Parameters **external_temp** – current value of external temperature to be controlled.

Returns reply of the device to the last call of “query”

set_pump_level(*pump_level: int = 6*) → str

Set pump level Raises ValueError, if pump level is invalid.

Parameters **pump_level** – pump level.

Returns reply of the device to the last call of “query”

set_ramp_iterations(*num: int = 1*) → str

Define number of ramp program cycles.

Parameters **num** – number of program cycles to be performed.

Returns reply of the device to the last call of “query”

set_ramp_program(*program: int = 1*) → str

Define ramp program for following ramp commands. Raises ValueError if maximum number of ramp programs (5) is exceeded.

Parameters **program** – Number of ramp program to be activated for following commands.

Returns reply of the device to the last call of “query”

set_ramp_segment(*temp: float = 20.0, dur: int = 0, tol: float = 0.0, pump: int = 6*) → str

Define segment of current ramp program - will be attached to current program. Raises ValueError, if pump level is invalid.

Parameters

- **temp** – target temperature of current ramp segment
- **dur** – duration in minutes, in which target temperature should be reached
- **tol** – tolerance at which target temperature should be reached (for 0.00, next segment is started after dur has passed).
- **pump** – pump level to be used for this program segment.

Returns reply of the device to the last call of “query”

set_temp_set_point(*temp_set_point: float = 20.0*) → str

Define temperature set point

Parameters **temp_set_point** – temperature set point.

Returns reply of the device to the last call of “query”

start() → *None*

Start this device.

start_ramp() → str

Start current ramp program.

Returns reply of the device to the last call of “query”

stop() → *None*

Stop this device. Disables access and closes the communication protocol.

stop_ramp() → str

Stop current ramp program.

Returns reply of the device to the last call of “query”

validate_pump_level(*level: int*)

Validates pump level. Raises ValueError, if pump level is incorrect. :param level: pump level, integer

class **LaudaProRp245eCommand**(*value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None*)

Bases: *hvl_ccb.utils.enum.ValueEnum*

Commands for Lauda PRO RP245E Chiller Command strings most often need to be complimented with a parameter (attached as a string) before being sent to the device. Commands implemented as defined in “Lauda Betriebsanleitung fuer PRO Badthermostate und Umwaeltzthermostate” pages 42 - 49

BATH_TEMP = 'IN_PV_00'

Request internal bath temperature

COM_TIME_OUT = 'OUT_SP_08_'

Define communication time out

CONT_MODE = 'OUT_MODE_01_'

Set control mode 1=internal, 2=ext. analog, 3=ext. serial, 4=USB, 5=ethernet

DEVICE_TYPE = 'TYPE'

Request device type

EXTERNAL_TEMP = 'OUT_PV_05_'

Pass on external controlled temperature

LOWER_TEMP = 'OUT_SP_05_'

Define lower temp limit

OPERATION_MODE = 'OUT_SP_02_'

Define operation mode

PUMP_LEVEL = 'OUT_SP_01_'

Define pump level 1-8

RAMP_CONTINUE = 'RMP_CONT'

Continue a paused ramp program

RAMP_DELETE = 'RMP_RESET'

Reset a selected ramp program

RAMP_ITERATIONS = 'RMP_OUT_02_'

Define how often a ramp program should be iterated

RAMP_PAUSE = 'RMP_PAUSE'

Pause a selected ramp program

RAMP_SELECT = 'RMP_SELECT_'

Select a ramp program (target for all further ramp commands)

RAMP_SET = 'RMP_OUT_00_'

Define parameters of a selected ramp program

RAMP_START = 'RMP_START'

Start a selected ramp program

RAMP_STOP = 'RMP_STOP'

Stop a running ramp program

START = 'START'

Start temp control (pump and heating/cooling)

STOP = 'STOP'

Stop temp control (pump and heating/cooling)

TEMP_SET_POINT = 'OUT_SP_00_'

Define temperature set point

UPPER_TEMP = 'OUT_SP_04_'

Define upper temp limit

build_str(*param: str = ''*, *terminator: str = '\n'*)

Build a command string for sending to the device

Parameters

- **param** – Command's parameter given as string
- **terminator** – Command's terminator

Returns Command's string with a parameter and terminator

exception **LaudaProRp245eCommandError**

Bases: [hvl_ccb.dev.base.DeviceException](#)

Exception raised when an error is returned upon a command.

class **LaudaProRp245eConfig**(*temp_set_point_init: Union[int, float] = 20.0*, *pump_init: int = 6*, *upper_temp: Union[int, float] = 80.0*, *lower_temp: Union[int, float] = - 55.0*, *com_time_out: Union[int, float] = 0*, *max_pump_level: int = 8*, *max_pr_number: int = 5*, *operation_mode: Union[int, [hvl_ccb.dev.lauda.LaudaProRp245eConfig.OperationModeEnum](#)] = [OperationModeEnum.AUTO](#)*, *control_mode: Union[int, [hvl_ccb.dev.lauda.LaudaProRp245eConfig.ExtControlModeEnum](#)] = [ExtControlModeEnum.INTERNAL](#)*)

Bases: `object`

Configuration for the Lauda RP245E circulation chiller.

class **ExtControlModeEnum**(*value*)

Bases: `enum.IntEnum`

Source for definition of external, controlled temperature (option 2, 3 and 4 are not available with current configuration of the Lauda RP245E, add-on hardware would required)

ANALOG = 2

ETH = 5

EXPT100 = 1

INTERNAL = 0

SERIAL = 3

```

    USB = 4

class OperationModeEnum(value)
    Bases: enum.IntEnum

    Operation Mode (Cooling OFF/Cooling On/AUTO - set to AUTO)

    AUTO = 2
        Automatically select heating/cooling

    COOLOFF = 0

    COOLON = 1

clean_values() → None

com_time_out: Union[int, float] = 0
    Communication time out (0 = OFF)

control_mode: Union[int, hvl_ccb.dev.lauda.LaudaProRp245eConfig.ExtControlModeEnum]
= 0

force_value(fieldname, value)
    Forces a value to a dataclass field despite the class being frozen.

    NOTE: you can define post_force_value method with same signature as this method to do extra processing
    after value has been forced on fieldname.

    Parameters
        • fieldname – name of the field
        • value – value to assign

is_configdataclass = True

classmethod keys() → Sequence[str]
    Returns a list of all configdataclass fields key-names.

    Returns a list of strings containing all keys.

lower_temp: Union[int, float] = -55.0
    Lower temperature limit (safe for Galdden HT135 cooling liquid)

max_pr_number: int = 5
    Maximum number of ramp programs that can be stored in the memory of the chiller

max_pump_level: int = 8
    Highest pump level of the chiller

operation_mode: Union[int,
hvl_ccb.dev.lauda.LaudaProRp245eConfig.OperationModeEnum] = 2

classmethod optional_defaults() → Dict[str, object]
    Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified
    on instantiation.

    Returns a list of strings containing all optional keys.

pump_init: int = 6
    Default pump Level

classmethod required_keys() → Sequence[str]
    Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on
    instantiation.

```

Returns a list of strings containing all required keys.

temp_set_point_init: `Union[int, float] = 20.0`

Default temperature set point

upper_temp: `Union[int, float] = 80.0`

Upper temperature limit (safe for Galdden HT135 cooling liquid)

class `LaudaProRp245eTcpCommunication(configuration)`

Bases: `hvl_ccb.comm.tcp.Tcp`

Implements the Communication Protocol for Lauda PRO RP245E TCP connection.

close() \rightarrow `None`

Close the Lauda PRO RP245E TCP connection.

static config_cls() \rightarrow `Type[hvl_ccb.dev.lauda.LaudaProRp245eTcpCommunicationConfig]`

Return the default configdataclass class.

Returns a reference to the default configdataclass class

open() \rightarrow `None`

Open the Lauda PRO RP245E TCP connection.

Raises `LaudaProRp245eCommandError` – if the connection fails.

query_command(*command*: `hvl_ccb.dev.lauda.LaudaProRp245eCommand`, *param*: `str = ""`) \rightarrow `str`

Send and receive function. E.g. to be used when setting/changing device setting. :param *command*: first part of command string, defined in `LaudaProRp245eCommand` :param *param*: second part of command string, parameter (by default '') :return: `None`

read() \rightarrow `str`

Receive value function. :return: reply from device as a string, the terminator, as well as the 'OK' stripped from the reply to make it directly useful as a value (e.g. in case the internal bath temperature is requested)

write_command(*command*: `hvl_ccb.dev.lauda.LaudaProRp245eCommand`, *param*: `str = ""`) \rightarrow `None`

Send command function. :param *command*: first part of command string, defined in `LaudaProRp245eCommand` :param *param*: second part of command string, parameter (by default '') :return: `None`

class `LaudaProRp245eTcpCommunicationConfig`(*host*: `Union[str, ipaddress.IPv4Address, ipaddress.IPv6Address]`, *port*: `int = 54321`, *bufsize*: `int = 1024`, *wait_sec_pre_read_or_write*: `Union[int, float] = 0.005`, *terminator*: `str = '\n'`)

Bases: `hvl_ccb.comm.tcp.TcpCommunicationConfig`

Configuration dataclass for `LaudaProRp245eTcpCommunication`.

clean_values() \rightarrow `None`

force_value(*fieldname*, *value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define `post_force_value` method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

classmethod **keys()** \rightarrow `Sequence[str]`

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

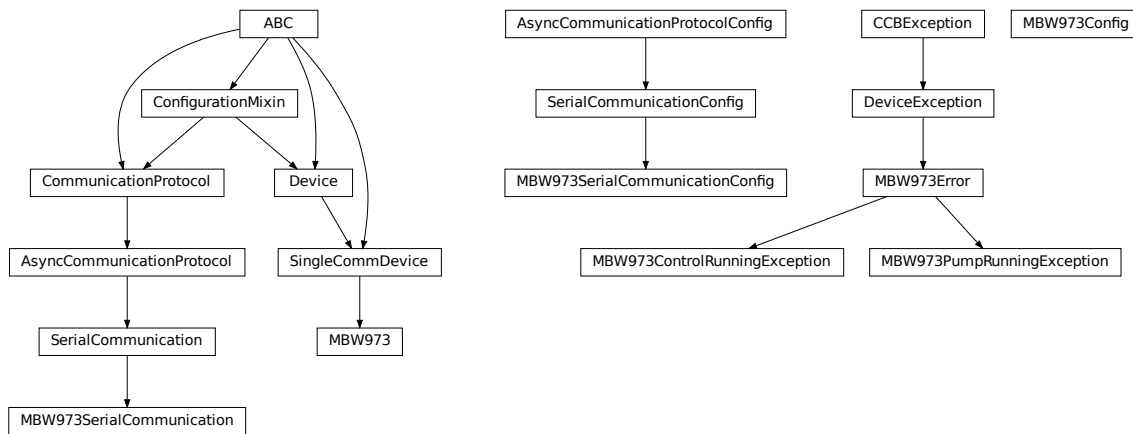
terminator: str = '\r\n'

The terminator character

wait_sec_pre_read_or_write: Union[int, float] = 0.005

Delay time between commands in seconds

hvl_ccb.dev.mbw973



Device class for controlling a MBW 973 SF6 Analyzer over a serial connection.

The MBW 973 is a gas analyzer designed for gas insulated switchgear and measures humidity, SF6 purity and SO2 contamination in one go. Manufacturer homepage: <https://www.mbw.ch/products/sf6-gas-analysis/973-sf6-analyzer/>

class MBW973(com, dev_config=None)

Bases: `hvl_ccb.dev.base.SingleCommDevice`

MBW 973 dew point mirror device class.

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

static default_com_cls()

Get the class for the default communication protocol used with this device.

Returns the type of the standard communication protocol for this device

is_done() → bool

Poll status of the dew point mirror and return True, if all measurements are done.

Returns True, if all measurements are done; False otherwise.

Raises *SerialCommunicationIOError* – when communication port is not opened

read(*cast_type: typing.Type = <class 'str'>*)

Read value from *self.com* and cast to *cast_type*. Raises *ValueError* if read text (*str*) is not convertible to *cast_type*, e.g. to *float* or to *int*.

Returns Read value of *cast_type* type.

read_float() → float

Convenience wrapper for *self.read()*, with typing hint for return value.

Returns Read *float* value.

read_int() → int

Convenience wrapper for *self.read()*, with typing hint for return value.

Returns Read *int* value.

read_measurements() → Dict[str, float]

Read out measurement values and return them as a dictionary.

Returns Dictionary with values.

Raises *SerialCommunicationIOError* – when communication port is not opened

set_measuring_options(*humidity: bool = True, sf6_purity: bool = False*) → *None*

Send measuring options to the dew point mirror.

Parameters

- **humidity** – Perform humidity test or not?
- **sf6_purity** – Perform SF6 purity test or not?

Raises *SerialCommunicationIOError* – when communication port is not opened

start() → *None*

Start this device. Opens the communication protocol and retrieves the set measurement options from the device.

Raises *SerialCommunicationIOError* – when communication port cannot be opened.

start_control() → *None*

Start dew point control to acquire a new value set.

Raises *SerialCommunicationIOError* – when communication port is not opened

stop() → *None*

Stop the device. Closes also the communication protocol.

write(*value*) → *None*

Send *value* to *self.com*.

Parameters **value** – Value to send, converted to *str*.

Raises *SerialCommunicationIOError* – when communication port is not opened

class **MBW973Config**(*polling_interval: Union[int, float] = 2*)

Bases: object

Device configuration dataclass for MBW973.

clean_values()

force_value(*fieldname*, *value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

is_configdataclass = True

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

polling_interval: Union[int, float] = 2

Polling period for *is_done* status queries [in seconds].

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

exception MBW973ControlRunningException

Bases: [hvl_ccb.dev.mbw973.MBW973Error](#)

Error indicating there is still a measurement running, and a new one cannot be started.

exception MBW973Error

Bases: [hvl_ccb.dev.base.DeviceException](#)

General error with the MBW973 dew point mirror device.

exception MBW973PumpRunningException

Bases: [hvl_ccb.dev.mbw973.MBW973Error](#)

Error indicating the pump of the dew point mirror is still recovering gas, unable to start a new measurement.

class MBW973SerialCommunication(*configuration*)

Bases: [hvl_ccb.comm.serial.SerialCommunication](#)

Specific communication protocol implementation for the MBW973 dew point mirror. Already predefines device-specific protocol parameters in config.

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

```
class MBW973SerialCommunicationConfig(terminator: bytes = b'\r', encoding: str = 'utf-8',
                                     encoding_error_handling: str = 'strict',
                                     wait_sec_read_text_nonempty: Union[int, float] = 0.5,
                                     default_n_attempts_read_text_nonempty: int = 10, port: Union[str,
                                     NoneType] = None, baudrate: int = 9600, parity: Union[str,
                                     hvl_ccb.comm.serial.SerialCommunicationParity] =
                                     <SerialCommunicationParity.NONE: 'N'>, stopbits: Union[int,
                                     hvl_ccb.comm.serial.SerialCommunicationStopbits] =
                                     <SerialCommunicationStopbits.ONE: 1>, bytesize: Union[int,
                                     hvl_ccb.comm.serial.SerialCommunicationBytesize] =
                                     <SerialCommunicationBytesize.EIGHTBITS: 8>, timeout:
                                     Union[int, float] = 3)
```

Bases: `hvl_ccb.comm.serial.SerialCommunicationConfig`

baudrate: `int = 9600`

Baudrate for MBW973 is 9600 baud

bytesize: `Union[int, hvl_ccb.comm.serial.SerialCommunicationBytesize] = 8`

One byte is eight bits long

force_value(*fieldname*, *value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

parity: `Union[str, hvl_ccb.comm.serial.SerialCommunicationParity] = 'N'`

MBW973 does not use parity

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

stopbits: `Union[int, hvl_ccb.comm.serial.SerialCommunicationStopbits] = 1`

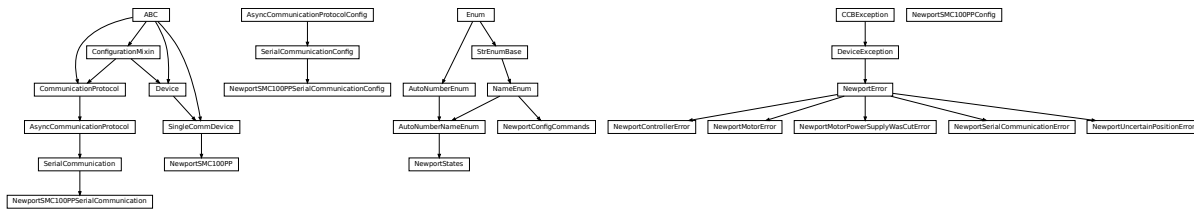
MBW973 does use one stop bit

terminator: `bytes = b'\r'`

The terminator is only CR

timeout: `Union[int, float] = 3`

use 3 seconds timeout as default

hvl_ccb.dev.newport

Device class for Newport SMC100PP stepper motor controller with serial communication.

The SMC100PP is a single axis motion controller/driver for stepper motors up to 48 VDC at 1.5 A rms. Up to 31 controllers can be networked through the internal RS-485 communication link.

Manufacturer homepage: <https://www.newport.com/f/smc100-single-axis-dc-or-stepper-motion-controller>

class NewportConfigCommands(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)

Bases: [hvl_ccb.utils.enum.NameEnum](#)

Commands predefined by the communication protocol of the SMC100PP

AC = 'acceleration'

BA = 'backlash_compensation'

BH = 'hysteresis_compensation'

FRM = 'micro_step_per_full_step_factor'

FRS = 'motion_distance_per_full_step'

HT = 'home_search_type'

JR = 'jerk_time'

OH = 'home_search_velocity'

OT = 'home_search_timeout'

QIL = 'peak_output_current_limit'

SA = 'rs485_address'

SL = 'negative_software_limit'

SR = 'positive_software_limit'

VA = 'velocity'

VB = 'base_velocity'

ZX = 'stage_configuration'

exception NewportControllerError

Bases: [hvl_ccb.dev.newport.NewportError](#)

Error with the Newport controller.

exception NewportError

Bases: [hvl_ccb.dev.base.DeviceException](#)

General Exception for Newport Device

exception NewportMotorErrorBases: [hvl_ccb.dev.newport.NewportError](#)

Error with the Newport motor.

exception NewportMotorPowerSupplyWasCutErrorBases: [hvl_ccb.dev.newport.NewportError](#)

Error with the Newport motor after the power supply was cut and then restored, without interrupting the communication with the controller.

class NewportSMC100PP(*com, dev_config=None*)Bases: [hvl_ccb.dev.base.SingleCommDevice](#)

Device class of the Newport motor controller SMC100PP

class MotorErrors(*value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None*)Bases: `aenum.Enum`Possible motor errors reported by the motor during `get_state()`.`DC_VOLTAGE_TOO_LOW = 3``FOLLOWING_ERROR = 6``HOMING_TIMEOUT = 5``NED_END_OF_TURN = 11``OUTPUT_POWER_EXCEEDED = 2``PEAK_CURRENT_LIMIT = 9``POS_END_OF_TURN = 10``RMS_CURRENT_LIMIT = 8``SHORT_CIRCUIT = 7``WRONG_ESP_STAGE = 4`**class StateMessages**(*value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None*)Bases: `aenum.Enum`Possible messages returned by the controller on `get_state()` query.`CONFIG = '14'``DISABLE_FROM_JOGGING = '3E'``DISABLE_FROM_MOVING = '3D'``DISABLE_FROM_READY = '3C'``HOMING_FROM_RS232 = '1E'``HOMING_FROM_SMC = '1F'``JOGGING_FROM_DISABLE = '47'``JOGGING_FROM_READY = '46'``MOVING = '28'``NO_REF_ESP_STAGE_ERROR = '10'``NO_REF_FROM_CONFIG = '0C'`

```

NO_REF_FROM_DISABLED = '0D'
NO_REF_FROM_HOMING = '0B'
NO_REF_FROM_JOGGING = '11'
NO_REF_FROM_MOVING = '0F'
NO_REF_FROM_READY = '0E'
NO_REF_FROM_RESET = '0A'
READY_FROM_DISABLE = '34'
READY_FROM_HOMING = '32'
READY_FROM_JOGGING = '35'
READY_FROM_MOVING = '33'

```

States

alias of `hvl_ccb.dev.newport.NewportStates`

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

static default_com_cls()

Get the class for the default communication protocol used with this device.

Returns the type of the standard communication protocol for this device

exit_configuration(add: Optional[int] = None) → None

Exit the CONFIGURATION state and go back to the NOT REFERENCED state. All configuration parameters are saved to the device's memory.

Parameters `add` – controller address (1 to 31)

Raises

- `SerialCommunicationIOError` – if the com is closed
- `NewportSerialCommunicationError` – if an unexpected answer is obtained
- `NewportControllerError` – if the controller reports an error

get_acceleration(add: Optional[int] = None) → Union[int, float]

Leave the configuration state. The configuration parameters are saved to the device's memory.

Parameters `add` – controller address (1 to 31)

Returns acceleration (preset units/s²), value between 1e-6 and 1e12

Raises

- `SerialCommunicationIOError` – if the com is closed
- `NewportSerialCommunicationError` – if an unexpected answer is obtained
- `NewportControllerError` – if the controller reports an error

get_controller_information(add: Optional[int] = None) → str

Get information on the controller name and driver version

Parameters `add` – controller address (1 to 31)

Returns controller information

Raises

- `SerialCommunicationIOError` – if the com is closed
- `NewportSerialCommunicationError` – if an unexpected answer is obtained
- `NewportControllerError` – if the controller reports an error

get_motor_configuration(*add: Optional[int] = None*) → Dict[str, float]

Query the motor configuration and returns it in a dictionary.

Parameters *add* – controller address (1 to 31)

Returns dictionary containing the motor's configuration

Raises

- `SerialCommunicationIOError` – if the com is closed
- `NewportSerialCommunicationError` – if an unexpected answer is obtained
- `NewportControllerError` – if the controller reports an error

get_move_duration(*dist: Union[int, float], add: Optional[int] = None*) → float

Estimate the time necessary to move the motor of the specified distance.

Parameters

- *dist* – distance to travel
- *add* – controller address (1 to 31), defaults to self.address

Raises

- `SerialCommunicationIOError` – if the com is closed
- `NewportSerialCommunicationError` – if an unexpected answer is obtained
- `NewportControllerError` – if the controller reports an error

get_negative_software_limit(*add: Optional[int] = None*) → Union[int, float]

Get the negative software limit (the maximum position that the motor is allowed to travel to towards the left).

Parameters *add* – controller address (1 to 31)

Returns negative software limit (preset units), value between -1e12 and 0

Raises

- `SerialCommunicationIOError` – if the com is closed
- `NewportSerialCommunicationError` – if an unexpected answer is obtained
- `NewportControllerError` – if the controller reports an error

get_position(*add: Optional[int] = None*) → float

Returns the value of the current position.

Parameters *add* – controller address (1 to 31)

Raises

- `SerialCommunicationIOError` – if the com is closed
- `NewportSerialCommunicationError` – if an unexpected answer is obtained
- `NewportControllerError` – if the controller reports an error
- `NewportUncertainPositionError` – if the position is ambiguous

get_positive_software_limit(*add: Optional[int] = None*) → Union[int, float]

Get the positive software limit (the maximum position that the motor is allowed to travel to towards the right).

Parameters *add* – controller address (1 to 31)

Returns positive software limit (preset units), value between 0 and 1e12

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

get_state(*add: int = None*) → *StateMessages*

Check on the motor errors and the controller state

Parameters *add* – controller address (1 to 31)

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error
- *NewportMotorError* – if the motor reports an error

Returns state message from the device (member of StateMessages)

go_home(*add: Optional[int] = None*) → *None*

Move the motor to its home position.

Parameters *add* – controller address (1 to 31), defaults to self.address

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

go_to_configuration(*add: Optional[int] = None*) → *None*

This method is executed during start(). It can also be executed after a reset(). The controller is put in CONFIG state, where configuration parameters can be changed.

Parameters *add* – controller address (1 to 31)

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

initialize(*add: Optional[int] = None*) → *None*

Puts the controller from the NOT_REF state to the READY state. Sends the motor to its “home” position.

Parameters *add* – controller address (1 to 31)

Raises

- *SerialCommunicationIOError* – if the com is closed

- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

move_to_absolute_position(*pos*: Union[int, float], *add*: Optional[int] = None) → None

Move the motor to the specified position.

Parameters

- **pos** – target absolute position (affected by the configured offset)
- **add** – controller address (1 to 31), defaults to self.address

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

move_to_relative_position(*pos*: Union[int, float], *add*: Optional[int] = None) → None

Move the motor of the specified distance.

Parameters

- **pos** – distance to travel (the sign gives the direction)
- **add** – controller address (1 to 31), defaults to self.address

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

reset(*add*: Optional[int] = None) → None

Resets the controller, equivalent to a power-up. This puts the controller back to NOT REFERENCED state, which is necessary for configuring the controller.

Parameters **add** – controller address (1 to 31)

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

set_acceleration(*acc*: Union[int, float], *add*: Optional[int] = None) → None

Leave the configuration state. The configuration parameters are saved to the device's memory.

Parameters

- **acc** – acceleration (preset units/s²), value between 1e-6 and 1e12
- **add** – controller address (1 to 31)

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

set_motor_configuration(*add: Optional[int] = None, config: Optional[dict] = None*) → *None*

Set the motor configuration. The motor must be in CONFIG state.

Parameters

- **add** – controller address (1 to 31)
- **config** – dictionary containing the motor's configuration

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

set_negative_software_limit(*lim: Union[int, float], add: Optional[int] = None*) → *None*

Set the negative software limit (the maximum position that the motor is allowed to travel to towards the left).

Parameters

- **lim** – negative software limit (preset units), value between -1e12 and 0
- **add** – controller address (1 to 31)

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

set_positive_software_limit(*lim: Union[int, float], add: Optional[int] = None*) → *None*

Set the positive software limit (the maximum position that the motor is allowed to travel to towards the right).

Parameters

- **lim** – positive software limit (preset units), value between 0 and 1e12
- **add** – controller address (1 to 31)

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

start()

Opens the communication protocol and applies the config.

Raises *SerialCommunicationIOError* – when communication port cannot be opened

stop() → *None*

Stop the device. Close the communication protocol.

stop_motion(*add: Optional[int] = None*) → *None*

Stop a move in progress by decelerating the positioner immediately with the configured acceleration until it stops. If a controller address is provided, stops a move in progress on this controller, else stops the moves on all controllers.

Parameters **add** – controller address (1 to 31)

Raises

- [`SerialCommunicationIOError`](#) – if the com is closed
- [`NewportSerialCommunicationError`](#) – if an unexpected answer is obtained
- [`NewportControllerError`](#) – if the controller reports an error

wait_until_motor_initialized(*add: Optional[int] = None*) → *None*

Wait until the motor leaves the HOMING state (at which point it should have arrived to the home position).

Parameters *add* – controller address (1 to 31)

Raises

- [`SerialCommunicationIOError`](#) – if the com is closed
- [`NewportSerialCommunicationError`](#) – if an unexpected answer is obtained
- [`NewportControllerError`](#) – if the controller reports an error

```
class NewportSMC100PPConfig(address: int = 1, user_position_offset: Union[int, float] = 23.987,  
                           screw_scaling: Union[int, float] = 1, exit_configuration_wait_sec: Union[int,  
                           float] = 5, move_wait_sec: Union[int, float] = 1, acceleration: Union[int, float]  
                           = 10, backlash_compensation: Union[int, float] = 0, hysteresis_compensation:  
                           Union[int, float] = 0.015, micro_step_per_full_step_factor: int = 100,  
                           motion_distance_per_full_step: Union[int, float] = 0.01, home_search_type:  
                           Union[int, hvl_ccb.dev.newport.NewportSMC100PPConfig.HomeSearch] =  
                           HomeSearch.HomeSwitch, jerk_time: Union[int, float] = 0.04,  
                           home_search_velocity: Union[int, float] = 4, home_search_timeout: Union[int,  
                           float] = 27.5, home_search_polling_interval: Union[int, float] = 1,  
                           peak_output_current_limit: Union[int, float] = 0.4, rs485_address: int = 2,  
                           negative_software_limit: Union[int, float] = - 23.5, positive_software_limit:  
                           Union[int, float] = 25, velocity: Union[int, float] = 4, base_velocity: Union[int,  
                           float] = 0, stage_configuration: Union[int,  
                           hvl_ccb.dev.newport.NewportSMC100PPConfig.EspStageConfig] =  
                           EspStageConfig.EnableEspStageCheck)
```

Bases: `object`

Configuration dataclass for the Newport motor controller SMC100PP.

```
class EspStageConfig(value=<no_arg>, names=None, module=None, type=None, start=1,  
                   boundary=None)
```

Bases: `aenum.IntEnum`

Different configurations to check or not the motor configuration upon power-up.

DisableEspStageCheck = 1

EnableEspStageCheck = 3

UpdateEspStageInfo = 2

```
class HomeSearch(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)
```

Bases: `aenum.IntEnum`

Different methods for the motor to search its home position during initialization.

CurrentPosition = 1

EndOfRunSwitch = 4

EndOfRunSwitch_and_Index = 3

HomeSwitch = 2


```

    HomeSwitch_and_Index = 0
acceleration: Union[int, float] = 10
address: int = 1
backlash_compensation: Union[int, float] = 0
base_velocity: Union[int, float] = 0
clean_values()
exit_configuration_wait_sec: Union[int, float] = 5
force_value(fieldname, value)
    Forces a value to a dataclass field despite the class being frozen.

    NOTE: you can define post_force_value method with same signature as this method to do extra processing
    after value has been forced on fieldname.

    Parameters
        • fieldname – name of the field
        • value – value to assign

home_search_polling_interval: Union[int, float] = 1
home_search_timeout: Union[int, float] = 27.5
home_search_type: Union[int, hvl\_ccb.dev.newport.NewportSMC100PPConfig.HomeSearch]
= 2
home_search_velocity: Union[int, float] = 4
hysteresis_compensation: Union[int, float] = 0.015
is_configdataclass = True
jerk_time: Union[int, float] = 0.04
classmethod keys() → Sequence[str]
    Returns a list of all configdataclass fields key-names.

    Returns a list of strings containing all keys.

micro_step_per_full_step_factor: int = 100
motion_distance_per_full_step: Union[int, float] = 0.01
property motor_config: Dict[str, float]
    Gather the configuration parameters of the motor into a dictionary.

    Returns dict containing the configuration parameters of the motor

move_wait_sec: Union[int, float] = 1
negative_software_limit: Union[int, float] = -23.5
classmethod optional_defaults() → Dict[str, object]
    Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified
    on instantiation.

    Returns a list of strings containing all optional keys.

peak_output_current_limit: Union[int, float] = 0.4
positive_software_limit: Union[int, float] = 25

```

`post_force_value(fieldname, value)`

classmethod `required_keys()` → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

`rs485_address: int = 2`

`screw_scaling: Union[int, float] = 1`

`stage_configuration: Union[int, hvl_ccb.dev.newport.NewportSMC100PPConfig.EspStageConfig] = 3`

`user_position_offset: Union[int, float] = 23.987`

`velocity: Union[int, float] = 4`

class `NewportSMC100PPSerialCommunication(configuration)`

Bases: [hvl_ccb.comm.serial.SerialCommunication](#)

Specific communication protocol implementation for NewportSMC100 controller. Already predefines device-specific protocol parameters in config.

class `ControllerErrors(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)`

Bases: `aenum.Enum`

Possible controller errors with values as returned by the device in response to sent commands.

`ADDR_INCORRECT = 'B'`

`CMD_EXEC_ERROR = 'V'`

`CMD_NOT_ALLOWED = 'D'`

`CMD_NOT_ALLOWED_CC = 'X'`

`CMD_NOT_ALLOWED_CONFIGURATION = 'I'`

`CMD_NOT_ALLOWED_DISABLE = 'J'`

`CMD_NOT_ALLOWED_HOMING = 'L'`

`CMD_NOT_ALLOWED_MOVING = 'M'`

`CMD_NOT_ALLOWED_NOT_REFERENCED = 'H'`

`CMD_NOT_ALLOWED_PP = 'W'`

`CMD_NOT_ALLOWED_READY = 'K'`

`CODE_OR_ADDR_INVALID = 'A'`

`COM_TIMEOUT = 'S'`

`DISPLACEMENT_OUT_OF_LIMIT = 'G'`

`EEPROM_ACCESS_ERROR = 'U'`

`ESP_STAGE_NAME_INVALID = 'F'`

`HOME_STARTED = 'E'`

`NO_ERROR = '@'`

`PARAM_MISSING_OR_INVALID = 'C'`

`POSITION_OUT_OF_LIMIT = 'N'`

`check_for_error(add: int) → None`

Ask the Newport controller for the last error it recorded.

This method is called after every command or query.

Parameters `add` – controller address (1 to 31)

Raises

- `SerialCommunicationIOError` – if the com is closed
- `NewportSerialCommunicationError` – if an unexpected answer is obtained
- `NewportControllerError` – if the controller reports an error

`static config_cls()`

Return the default configdataclass class.

Returns a reference to the default configdataclass class

`query(add: int, cmd: str, param: Optional[Union[int, float, str]] = None) → str`

Send a query to the controller, read the answer, and check for errors. The prefix add+cmd is removed from the answer.

Parameters

- `add` – the controller address (1 to 31)
- `cmd` – the command to be sent
- `param` – optional parameter (int/float/str) appended to the command

Returns the answer from the device without the prefix

Raises

- `SerialCommunicationIOError` – if the com is closed
- `NewportSerialCommunicationError` – if an unexpected answer is obtained
- `NewportControllerError` – if the controller reports an error

`query_multiple(add: int, cmd: str, prefixes: List[str]) → List[str]`

Send a query to the controller, read the answers, and check for errors. The prefixes are removed from the answers.

Parameters

- `add` – the controller address (1 to 31)
- `cmd` – the command to be sent
- `prefixes` – prefixes of each line expected in the answer

Returns list of answers from the device without prefix

Raises

- `SerialCommunicationIOError` – if the com is closed
- `NewportSerialCommunicationError` – if an unexpected answer is obtained
- `NewportControllerError` – if the controller reports an error

read_text() → str

Read one line of text from the serial port, and check for presence of a null char which indicates that the motor power supply was cut and then restored. The input buffer may hold additional data afterwards, since only one line is read.

This method uses *self.access_lock* to ensure thread-safety.

Returns String read from the serial port; '' if there was nothing to read.

Raises

- *SerialCommunicationIOError* – when communication port is not opened
- *NewportMotorPowerSupplyWasCutError* – if a null char is read

send_command(add: int, cmd: str, param: Optional[Union[int, float, str]] = None) → None

Send a command to the controller, and check for errors.

Parameters

- **add** – the controller address (1 to 31)
- **cmd** – the command to be sent
- **param** – optional parameter (int/float/str) appended to the command

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

send_stop(add: int) → None

Send the general stop ST command to the controller, and check for errors.

Parameters **add** – the controller address (1 to 31)

Returns ControllerErrors reported by Newport Controller

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained

```
class NewportSMC100PPSerialCommunicationConfig(terminator: bytes = b'\n', encoding: str = 'ascii',
                                                encoding_error_handling: str = 'replace',
                                                wait_sec_read_text_nonempty: Union[int, float] = 0.5,
                                                default_n_attempts_read_text_nonempty: int = 10,
                                                port: Union[str, NoneType] = None, baudrate: int =
                                                57600, parity: Union[str,
                                                hvl_ccb.comm.serial.SerialCommunicationParity] =
                                                <SerialCommunicationParity.NONE: 'N'>, stopbits:
                                                Union[int,
                                                hvl_ccb.comm.serial.SerialCommunicationStopbits] =
                                                <SerialCommunicationStopbits.ONE: 1>, bytesize:
                                                Union[int,
                                                hvl_ccb.comm.serial.SerialCommunicationBytesize] =
                                                <SerialCommunicationBytesize.EIGHTBITS: 8>,
                                                timeout: Union[int, float] = 10)
```

Bases: *hvl_ccb.comm.serial.SerialCommunicationConfig*

baudrate: `int = 57600`

Baudrate for NewportSMC100 controller is 57600 baud

bytesize: `Union[int, hvl_ccb.comm.serial.SerialCommunicationBytesize] = 8`

NewportSMC100 controller uses 8 bits for one data byte

encoding: `str = 'ascii'`

use ASCII as de-/encoding, cf. the manual

encoding_error_handling: `str = 'replace'`

replace bytes with instead of raising utf-8 exception when decoding fails

force_value(*fieldname*, *value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

classmethod **keys**() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod **optional_defaults**() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

parity: `Union[str, hvl_ccb.comm.serial.SerialCommunicationParity] = 'N'`

NewportSMC100 controller does not use parity

classmethod **required_keys**() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

stopbits: `Union[int, hvl_ccb.comm.serial.SerialCommunicationStopbits] = 1`

NewportSMC100 controller uses one stop bit

terminator: `bytes = b'\r\n'`

The terminator is CR/LF

timeout: `Union[int, float] = 10`

use 10 seconds timeout as default

exception **NewportSerialCommunicationError**

Bases: [hvl_ccb.dev.newport.NewportError](#)

Communication error with the Newport controller.

class **NewportStates**(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)

Bases: [hvl_ccb.utils.enum.AutoNumberNameEnum](#)

States of the Newport controller. Certain commands are allowed only in certain states.

CONFIG = 3

DISABLE = 6

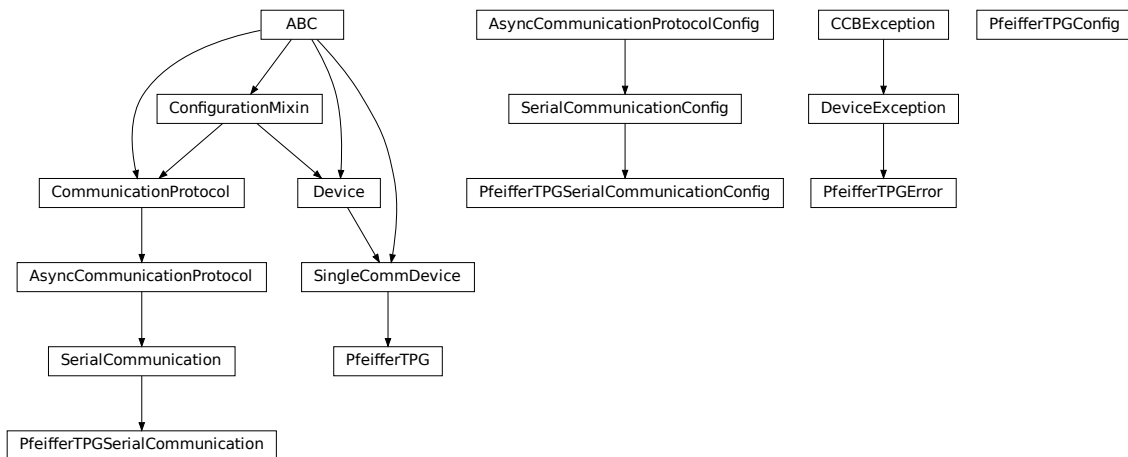
HOMING = 2
JOGGING = 7
MOVING = 5
NO_REF = 1
READY = 4

exception NewportUncertainPositionError

Bases: [hvl_ccb.dev.newport.NewportError](#)

Error with the position of the Newport motor.

[hvl_ccb.dev.pfeiffer_tpg](#)



Device class for Pfeiffer TPG controllers.

The Pfeiffer TPG control units are used to control Pfeiffer Compact Gauges. Models: TPG 251 A, TPG 252 A, TPG 256A, TPG 261, TPG 262, TPG 361, TPG 362 and TPG 366.

Manufacturer homepage: <https://www.pfeiffer-vacuum.com/en/products/measurement-analysis/measurement/activeline/controllers/>

class PfeifferTPG(*com*, *dev_config=None*)

Bases: [hvl_ccb.dev.base.SingleCommDevice](#)

Pfeiffer TPG control unit device class

class PressureUnits(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)

Bases: [hvl_ccb.utils.enum.NameEnum](#)

Enum of available pressure units for the digital display. “0” corresponds either to bar or to mbar depending on the TPG model. In case of doubt, the unit is visible on the digital display.

Micron = 3

Pascal = 2

Torr = 1

```
Volt = 5
bar = 0
hPascal = 4
mbar = 0
class SensorStatus(value)
    Bases: enum.IntEnum
    An enumeration.
    Identification_error = 6
    No_sensor = 5
    Ok = 0
    Overrange = 2
    Sensor_error = 3
    Sensor_off = 4
    Underrange = 1
class SensorTypes(value)
    Bases: enum.Enum
    An enumeration.
    CMR = 4
    IKR = 2
    IKR11 = 2
    IKR9 = 2
    IMR = 5
    None = 7
    PBR = 6
    PKR = 3
    TPR = 1
    noSENSOR = 7
    noSen = 7
static config_cls()
    Return the default configdataclass class.
    Returns a reference to the default configdataclass class
static default_com_cls()
    Get the class for the default communication protocol used with this device.
    Returns the type of the standard communication protocol for this device
get_full_scale_mbar() → List[Union[int, float]]
    Get the full scale range of the attached sensors
    Returns full scale range values in mbar, like [0.01, 1, 0.1, 1000, 50000, 10]
    Raises
```

- *SerialCommunicationIOError* – when communication port is not opened
- *PfeifferTPGError* – if command fails

get_full_scale_unitless() → List[int]

Get the full scale range of the attached sensors. See lookup table between command and corresponding pressure in the device user manual.

Returns list of full scale range values, like [0, 1, 3, 3, 2, 0]

Raises

- *SerialCommunicationIOError* – when communication port is not opened
- *PfeifferTPGError* – if command fails

identify_sensors() → None

Send identification request TID to sensors on all channels.

Raises

- *SerialCommunicationIOError* – when communication port is not opened
- *PfeifferTPGError* – if command fails

measure(channel: int) → Tuple[str, float]

Get the status and measurement of one sensor

Parameters **channel** – int channel on which the sensor is connected, with 1 <= channel <= number_of_sensors

Returns measured value as float if measurement successful, sensor status as string if not

Raises

- *SerialCommunicationIOError* – when communication port is not opened
- *PfeifferTPGError* – if command fails

measure_all() → List[Tuple[str, float]]

Get the status and measurement of all sensors (this command is not available on all models)

Returns list of measured values as float if measurements successful, and or sensor status as strings if not

Raises

- *SerialCommunicationIOError* – when communication port is not opened
- *PfeifferTPGError* – if command fails

property number_of_sensors

set_display_unit(unit: Union[str, hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.PressureUnits]) → None

Set the unit in which the measurements are shown on the display.

Raises

- *SerialCommunicationIOError* – when communication port is not opened
- *PfeifferTPGError* – if command fails

set_full_scale_mbar(fsr: List[Union[int, float]]) → None

Set the full scale range of the attached sensors (in unit mbar)

Parameters **fsr** – full scale range values in mbar, for example [0.01, 1000]

Raises

- **`SerialCommunicationIOError`** – when communication port is not opened
- **`PfeifferTPGError`** – if command fails

set_full_scale_unitless(*fsr*: *List[int]*) → *None*

Set the full scale range of the attached sensors. See lookup table between command and corresponding pressure in the device user manual.

Parameters *fsr* – list of full scale range values, like *[0, 1, 3, 3, 2, 0]*

Raises

- **`SerialCommunicationIOError`** – when communication port is not opened
- **`PfeifferTPGError`** – if command fails

start() → *None*

Start this device. Opens the communication protocol, and identify the sensors.

Raises **`SerialCommunicationIOError`** – when communication port cannot be opened

stop() → *None*

Stop the device. Closes also the communication protocol.

property unit

The pressure unit of readings is always mbar, regardless of the display unit.

class PfeifferTPGConfig(*model*: *Union[str, hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGConfig.Model]* = *Model.TPG25xA*)

Bases: *object*

Device configuration dataclass for Pfeiffer TPG controllers.

class Model(*value*=<*no_arg*>, *names*=*None*, *module*=*None*, *type*=*None*, *start*=*1*, *boundary*=*None*)

Bases: *hvl_ccb.utils.enum.NameEnum*

An enumeration.

TPG25xA = {**0.1**: 8, **1**: 0, **10**: 1, **100**: 2, **1000**: 3, **2000**: 4, **5000**: 5, **10000**: 6, **50000**: 7}

TPGx6x = {**0.01**: 0, **0.1**: 1, **1**: 2, **10**: 3, **100**: 4, **1000**: 5, **2000**: 6, **5000**: 7, **10000**: 8, **50000**: 9}

is_valid_scale_range_reversed_str(*v*: *str*) → *bool*

Check if given string represents a valid reversed scale range of a model.

Parameters *v* – Reversed scale range string.

Returns *True* if valid, *False* otherwise.

clean_values()

force_value(*fieldname*, *value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

is_configdataclass = *True*

classmethod `keys()` → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

model: Union[str, [hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGConfig.Model](#)] = {0.1: 8, 1: 0, 10: 1, 100: 2, 1000: 3, 2000: 4, 5000: 5, 10000: 6, 50000: 7}

classmethod `optional_defaults()` → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

classmethod `required_keys()` → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

exception `PfeifferTPGError`

Bases: [hvl_ccb.dev.base.DeviceException](#)

Error with the Pfeiffer TPG Controller.

class `PfeifferTPGSerialCommunication(configuration)`

Bases: [hvl_ccb.comm.serial.SerialCommunication](#)

Specific communication protocol implementation for Pfeiffer TPG controllers. Already predefines device-specific protocol parameters in config.

static `config_cls()`

Return the default configdataclass class.

Returns a reference to the default configdataclass class

query(*cmd: str*) → str

Send a query, then read and returns the first line from the com port.

Parameters *cmd* – query message to send to the device

Returns first line read on the com

Raises

- [SerialCommunicationIOError](#) – when communication port is not opened
- [PfeifferTPGError](#) – if the device does not acknowledge the command or if the answer from the device is empty

send_command(*cmd: str*) → *None*

Send a command to the device and check for acknowledgement.

Parameters *cmd* – command to send to the device

Raises

- [SerialCommunicationIOError](#) – when communication port is not opened
- [PfeifferTPGError](#) – if the answer from the device differs from the expected acknowledgement character 'chr(6)'.

```
class PfeifferTPGSerialCommunicationConfig(terminator: bytes = b'\r\n', encoding: str = 'utf-8',
                                           encoding_error_handling: str = 'strict',
                                           wait_sec_read_text_nonempty: Union[int, float] = 0.5,
                                           default_n_attempts_read_text_nonempty: int = 10, port:
                                           Union[str, NoneType] = None, baudrate: int = 9600, parity:
                                           Union[str,
                                           hvl_ccb.comm.serial.SerialCommunicationParity] =
                                           <SerialCommunicationParity.NONE: 'N'>, stopbits:
                                           Union[int,
                                           hvl_ccb.comm.serial.SerialCommunicationStopbits] =
                                           <SerialCommunicationStopbits.ONE: 1>, bytesize:
                                           Union[int,
                                           hvl_ccb.comm.serial.SerialCommunicationBytesize] =
                                           <SerialCommunicationBytesize.EIGHTBITS: 8>, timeout:
                                           Union[int, float] = 3)
```

Bases: `hvl_ccb.comm.serial.SerialCommunicationConfig`

baudrate: `int = 9600`

Baudrate for Pfeiffer TPG controllers is 9600 baud

bytesize: `Union[int, hvl_ccb.comm.serial.SerialCommunicationBytesize] = 8`

One byte is eight bits long

force_value(*fieldname*, *value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

parity: `Union[str, hvl_ccb.comm.serial.SerialCommunicationParity] = 'N'`

Pfeiffer TPG controllers do not use parity

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

stopbits: `Union[int, hvl_ccb.comm.serial.SerialCommunicationStopbits] = 1`

Pfeiffer TPG controllers use one stop bit

terminator: `bytes = b'\r\n'`

The terminator is <CR><LF>

```
timeout: Union[int, float] = 3
    use 3 seconds timeout as default
```

hvl_ccb.dev.picotech_pt104

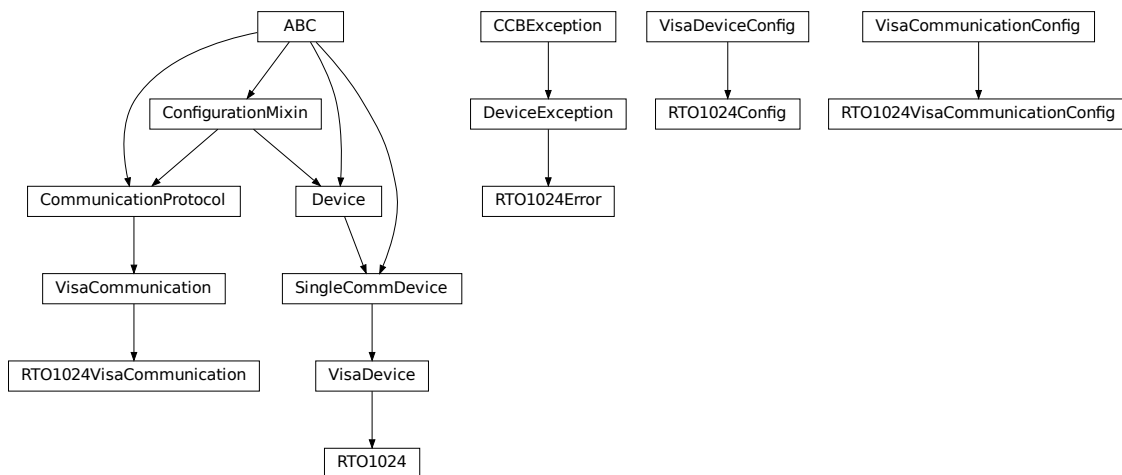
NOTE: [PicoSDK Python wrappers](#) already on import attempt to load the [PicoSDK library](#); thus, the API docs can only be generated in a system with the latter installed and are by default disabled.

To build the API docs for this submodule locally edit the `docs/hvl_ccb.dev.picotech_pt104.rst` file to remove the `.. code-block::` directive preceding the following directives:

```
.. inheritance-diagram:: hvl_ccb.dev.picotech_pt104
    :parts: 1

.. automodule:: hvl_ccb.dev.picotech_pt104
    :members:
    :undoc-members:
    :show-inheritance:
```

hvl_ccb.dev.rs_rto1024



Python module for the Rhode & Schwarz RTO 1024 oscilloscope. The communication to the device is through VISA, type TCP/IP / INSTR.

```
class RTO1024(com: Union[hvl_ccb.dev.rs_rto1024.RTO1024VisaCommunication,
                        hvl_ccb.dev.rs_rto1024.RTO1024VisaCommunicationConfig, dict], dev_config:
    Union[hvl_ccb.dev.rs_rto1024.RTO1024Config, dict])
```

Bases: `hvl_ccb.dev.visa.VisaDevice`

Device class for the Rhode & Schwarz RTO 1024 oscilloscope.

```
class TriggerModes(value=<no_arg>, names=None, module=None, type=None, start=1,
                    boundary=None)
```

Bases: `hvl_ccb.utils.enum.AutoNumberNameEnum`

Enumeration for the three available trigger modes.

AUTO = 1

FREERUN = 3

NORMAL = 2

classmethod names()

Returns a list of the available trigger modes. :return: list of strings

activate_measurements(*meas_n: int, source: str, measurements: List[str], category: str = 'AMPTIME'*)

Activate the list of 'measurements' of the waveform 'source' in the measurement box number 'meas_n'. The list 'measurements' starts with the main measurement and continues with additional measurements of the same 'category'.

Parameters

- **meas_n** – measurement number 1..8
- **source** – measurement source, for example C1W1
- **measurements** – list of measurements, the first one will be the main measurement.
- **category** – the category of measurements, by default AMPTIME

backup_waveform(*filename: str*) → *None*

Backup a waveform file from the standard directory specified in the device configuration to the standard backup destination specified in the device configuration. The filename has to be specified without .bin or path.

Parameters filename – The waveform filename without extension and path

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

static default_com_cls()

Return the default communication protocol for this device type, which is VisaCommunication.

Returns the VisaCommunication class

file_copy(*source: str, destination: str*) → *None*

Copy a file from one destination to another on the oscilloscope drive. If the destination file already exists, it is overwritten without notice.

Parameters

- **source** – absolute path to the source file on the DSO filesystem
- **destination** – absolute path to the destination file on the DSO filesystem

Raises *RT01024Error* – if the operation did not complete

get_acquire_length() → float

Gets the time of one acquisition, that is the time across the 10 divisions of the diagram.

- Range: 250E-12 ... 500 [s]
- Increment: 1E-12 [s]

Returns the time for one acquisition. Range: 250e-12 ... 500 [s]

get_channel_offset(*channel: int*) → float

Gets the voltage offset of the indicated channel.

Parameters `channel` – is the channel number (1..4)

Returns channel offset voltage in V (value between -1 and 1)

get_channel_position(*channel: int*) → float

Gets the vertical position of the indicated channel.

Parameters `channel` – is the channel number (1..4)

Returns channel position in div (value between -5 and 5)

get_channel_range(*channel: int*) → float

Queries the channel range in V.

Parameters `channel` – is the input channel (1..4)

Returns channel range in V

get_channel_scale(*channel: int*) → float

Queries the channel scale in V/div.

Parameters `channel` – is the input channel (1..4)

Returns channel scale in V/div

get_channel_state(*channel: int*) → bool

Queries if the channel is active or not.

Parameters `channel` – is the input channel (1..4)

Returns True if active, else False

get_reference_point() → int

Gets the reference point of the time scale in % of the display. If the “Trigger offset” is zero, the trigger point matches the reference point. ReferencePoint = zero pint of the time scale

- Range: 0 ... 100 [%]
- Increment: 1 [%]

Returns the reference in %

get_repetitions() → int

Get the number of acquired waveforms with RUN Nx SINGLE. Also defines the number of waveforms used to calculate the average waveform.

- Range: 1 ... 16777215
- Increment: 10
- *RST = 1

Returns the number of waveforms to acquire

get_timestamps() → List[float]

Gets the timestamps of all recorded frames in the history and returns them as a list of floats.

Returns list of timestamps in [s]

Raises [RT01024Error](#) – if the timestamps are invalid

list_directory(*path: str*) → List[Tuple[str, str, int]]

List the contents of a given directory on the oscilloscope filesystem.

Parameters `path` – is the path to a folder

Returns a list of filenames in the given folder

load_configuration(filename: str) → None

Load current settings from a configuration file. The filename has to be specified without base directory and '.dff' extension.

Information from the manual *ReCaLI* calls up the instrument settings from an intermediate memory identified by the specified number. The instrument settings can be stored to this memory using the command *SAV with the associated number. It also activates the instrument settings which are stored in a file and loaded using *MMEMory:LOAD:STATe* .

Parameters **filename** – is the name of the settings file without path and extension

local_display(state: bool) → None

Enable or disable local display of the scope.

Parameters **state** – is the desired local display state

prepare_ultra_segmentation() → None

Make ready for a new acquisition in ultra segmentation mode. This function does one acquisition without ultra segmentation to clear the history and prepare for a new measurement.

read_measurement(meas_n: int, name: str) → float

Parameters

- **meas_n** – measurement number 1..8
- **name** – measurement name, for example “MAX”

Returns measured value

run_continuous_acquisition() → None

Start acquiring continuously.

run_single_acquisition() → None

Start a single or Nx acquisition.

save_configuration(filename: str) → None

Save the current oscilloscope settings to a file. The filename has to be specified without path and '.dff' extension, the file will be saved to the configured settings directory.

Information from the manual *SAVe* stores the current instrument settings under the specified number in an intermediate memory. The settings can be recalled using the command *RCL with the associated number. To transfer the stored instrument settings to a file, use *MMEMory:STORe:STATe* .

Parameters **filename** – is the name of the settings file without path and extension

save_waveform_history(filename: str, channel: int, waveform: int = 1) → None

Save the history of one channel and one waveform to a .bin file. This function is used after an acquisition using sequence trigger mode (with or without ultra segmentation) was performed.

Parameters

- **filename** – is the name (without extension) of the file
- **channel** – is the channel number
- **waveform** – is the waveform number (typically 1)

Raises *RT01024Error* – if storing waveform times out

set_acquire_length(timerange: float) → None

Defines the time of one acquisition, that is the time across the 10 divisions of the diagram.

- Range: 250E-12 ... 500 [s]
- Increment: 1E-12 [s]
- *RST = 0.5 [s]

Parameters timerange – is the time for one acquisition. Range: 250e-12 ... 500 [s]

set_channel_offset(*channel: int, offset: float*) → *None*

Sets the voltage offset of the indicated channel.

- Range: Dependent on the channel scale and coupling [V]
- Increment: Minimum 0.001 [V], may be higher depending on the channel scale and coupling
- *RST = 0

Parameters

- **channel** – is the channel number (1..4)
- **offset** – Offset voltage. Positive values move the waveform down, negative values move it up.

set_channel_position(*channel: int, position: float*) → *None*

Sets the vertical position of the indicated channel as a graphical value.

- Range: -5.0 ... 5.0 [div]
- Increment: 0.02
- *RST = 0

Parameters

- **channel** – is the channel number (1..4)
- **position** – is the position. Positive values move the waveform up, negative values move it down.

set_channel_range(*channel: int, v_range: float*) → *None*

Sets the voltage range across the 10 vertical divisions of the diagram. Use the command alternatively instead of set_channel_scale.

- Range for range: Depends on attenuation factors and coupling. With 1:1 probe and external attenuations and 50 input coupling, the range is 10 mV to 10 V. For 1 M input coupling, it is 10 mV to 100 V. If the probe and/or external attenuation is changed, multiply the range values by the attenuation factors.
- Increment: 0.01
- *RST = 0.5

Parameters

- **channel** – is the channel number (1..4)
- **v_range** – is the vertical range [V]

set_channel_scale(*channel: int, scale: float*) → *None*

Sets the vertical scale for the indicated channel. The scale value is given in volts per division.

- Range for scale: depends on attenuation factor and coupling. With 1:1 probe and external attenuations and 50 input coupling, the vertical scale (input sensitivity) is 1 mV/div to 1 V/div. For 1 M input coupling, it is 1 mV/div to 10 V/div. If the probe and/or external attenuation is changed, multiply the values by the attenuation factors to get the actual scale range.
- Increment: 1e-3
- *RST = 0.05

See also: `set_channel_range`

Parameters

- **channel** – is the channel number (1..4)
- **scale** – is the vertical scaling [V/div]

set_channel_state(*channel: int, state: bool*) → *None*

Switches the channel signal on or off.

Parameters

- **channel** – is the input channel (1..4)
- **state** – is True for on, False for off

set_reference_point(*percentage: int*) → *None*

Sets the reference point of the time scale in % of the display. If the “Trigger offset” is zero, the trigger point matches the reference point. ReferencePoint = zero pint of the time scale

- Range: 0 ... 100 [%]
- Increment: 1 [%]
- *RST = 50 [%]

Parameters percentage – is the reference in %

set_repetitions(*number: int*) → *None*

Set the number of acquired waveforms with RUN Nx SINGLE. Also defines the number of waveforms used to calculate the average waveform.

- Range: 1 ... 16777215
- Increment: 10
- *RST = 1

Parameters number – is the number of waveforms to acquire

set_trigger_level(*channel: int, level: float, event_type: int = 1*) → *None*

Sets the trigger level for the specified event and source.

- Range: -10 to 10 V
- Increment: 1e-3 V
- *RST = 0 V

Parameters

- **channel** – indicates the trigger source.
 - 1..4 = channel 1 to 4, available for all event types 1..3

- 5 = external trigger input on the rear panel for analog signals, available for A-event type = 1
- 6..9 = not available

- **level** – is the voltage for the trigger level in [V].
- **event_type** – is the event type. 1: A-Event, 2: B-Event, 3: R-Event

set_trigger_mode(mode: Union[str, hvl_ccb.dev.rs_rto1024.RTO1024.TriggerModes]) → None

Sets the trigger mode which determines the behavior of the instrument if no trigger occurs.

Parameters mode – is either auto, normal, or freerun.

Raises **RT01024Error** – if an invalid triggermode is selected

set_trigger_source(channel: int, event_type: int = 1) → None

Set the trigger (Event A) source channel.

Parameters

- **channel** – is the channel number (1..4)
- **event_type** – is the event type. 1: A-Event, 2: B-Event, 3: R-Event

start() → None

Start the RTO1024 oscilloscope and bring it into a defined state and remote mode.

stop() → None

Stop the RTO1024 oscilloscope, reset events and close communication. Brings back the device to a state where local operation is possible.

stop_acquisition() → None

Stop any acquisition.

class RT01024Config(waveforms_path: str, settings_path: str, backup_path: str, spoll_interval: Union[int, float] = 0.5, spoll_start_delay: Union[int, float] = 2, command_timeout_seconds: Union[int, float] = 60, wait_sec_short_pause: Union[int, float] = 0.1, wait_sec_enable_history: Union[int, float] = 1, wait_sec_post_acquisition_start: Union[int, float] = 2)

Bases: [hvl_ccb.dev.visa.VisaDeviceConfig](#), [hvl_ccb.dev.rs_rto1024._RT01024ConfigDefaultsBase](#), [hvl_ccb.dev.rs_rto1024._RT01024ConfigBase](#)

Configdataclass for the RTO1024 device.

force_value(fieldname, value)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

exception RT01024Error

Bases: [hvl_ccb.dev.base.DeviceException](#)

class RT01024VisaCommunication(configuration)

Bases: [hvl_ccb.comm.visa.VisaCommunication](#)

Specialization of VisaCommunication for the RTO1024 oscilloscope

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

class RT01024VisaCommunicationConfig(*host: Union[str, ipaddress.IPv4Address, ipaddress.IPv6Address], interface_type: Union[str, [hvl_ccb.comm.visa.VisaCommunicationConfig.InterfaceType](#)] = InterfaceType.TCPIP_INSTR, board: int = 0, port: int = 5025, timeout: int = 5000, chunk_size: int = 204800, open_timeout: int = 1000, write_termination: str = '\n', read_termination: str = '\n', visa_backend: str = ''*)

Bases: [hvl_ccb.comm.visa.VisaCommunicationConfig](#)

Configuration dataclass for VisaCommunication with specifications for the RTO1024 device class.

force_value(fieldname, value)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

interface_type: Union[str, [hvl_ccb.comm.visa.VisaCommunicationConfig.InterfaceType](#)] = 2

Interface type of the VISA connection, being one of InterfaceType.

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

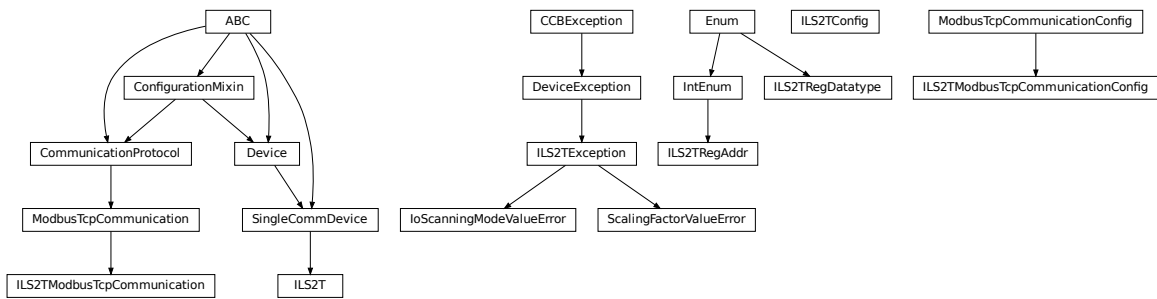
Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

hvl_ccb.dev.se_ils2t

Device class for controlling a Schneider Electric ILS2T stepper drive over modbus TCP.

class ILS2T(*com*, *dev_config=None*)

Bases: `hvl_ccb.dev.base.SingleCommDevice`

Schneider Electric ILS2T stepper drive class.

ACTION_JOG_VALUE = 0

The single action value for *ILS2T.Mode.JOG*

class ActionsPtp(*value*)

Bases: `enum.IntEnum`

Allowed actions in the point to point mode (*ILS2T.Mode.PTP*).

ABSOLUTE_POSITION = 0

RELATIVE_POSITION_MOTOR = 2

RELATIVE_POSITION_TARGET = 1

DEFAULT_IO_SCANNING_CONTROL_VALUES = {'action': 2, 'continue_after_stop_cu': 0, 'disable_driver_di': 0, 'enable_driver_en': 0, 'execute_stop_sh': 0, 'fault_reset_fr': 0, 'mode': 3, 'quick_stop_qs': 0, 'ref_16': 1500, 'ref_32': 0, 'reset_stop_ch': 0}

Default IO Scanning control mode values

class Mode(*value*)

Bases: `enum.IntEnum`

ILS2T device modes

JOG = 1

PTP = 3

class Ref16Jog(*value*)

Bases: `enum.Flag`

Allowed values for ILS2T ref_16 register (the shown values are the integer representation of the bits), all in Jog mode = 1

FAST = 4

NEG = 2

NEG_FAST = 6

NONE = 0

POS = 1

POS_FAST = 5

RegAddr

Modbus Register Addresses

alias of *hvl_ccb.dev.se_ils2t.ILS2TRegAddr*

RegDatatype

Modbus Register Datatypes

alias of *hvl_ccb.dev.se_ils2t.ILS2TRegDatatype*

class State(value)

Bases: `enum.IntEnum`

State machine status values

ON = 6

QUICKSTOP = 7

READY = 4

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

static default_com_cls()

Get the class for the default communication protocol used with this device.

Returns the type of the standard communication protocol for this device

disable(log_warn: bool = True, wait_sec_max: Optional[int] = None) → bool

Disable the driver of the stepper motor and enable the brake.

Note: the driver cannot be disabled if the motor is still running.

Parameters

- **log_warn** – if log a warning in case the motor cannot be disabled.
- **wait_sec_max** – maximal wait time for the motor to stop running and to disable it; by default, with *None*, use a config value

Returns *True* if disable request could and was sent, *False* otherwise.

do_ioscanning_write(kwargs: int) → None**

Perform a write operation using IO Scanning mode.

Parameters **kwargs** – Keyword-argument list with options to send, remaining are taken from the defaults.

enable() → None

Enable the driver of the stepper motor and disable the brake.

execute_absolute_position(position: int) → bool

Execute a absolute position change, i.e. enable motor, perform absolute position change, wait until done and disable motor afterwards.

Check position at the end if wrong do not raise error; instead just log and return check result.

Parameters **position** – absolute position of motor in user defined steps.

Returns *True* if actual position is as expected, *False* otherwise.

execute_relative_step(steps: int) → bool

Execute a relative step, i.e. enable motor, perform relative steps, wait until done and disable motor afterwards.

Check position at the end if wrong do not raise error; instead just log and return check result.

Parameters **steps** – Number of steps.

Returns *True* if actual position is as expected, *False* otherwise.

get_dc_volt() → float

Read the DC supply voltage of the motor.

Returns DC input voltage.

get_error_code() → Dict[int, Dict[str, Any]]

Read all messages in fault memory. Will read the full error message and return the decoded values. At the end the fault memory of the motor will be deleted. In addition, `reset_error` is called to re-enable the motor for operation.

Returns Dictionary with all information

get_position() → int

Read the position of the drive and store into status.

Returns Position step value

get_status() → Dict[str, int]

Perform an IO Scanning read and return the status of the motor.

Returns dict with status information.

get_temperature() → int

Read the temperature of the motor.

Returns Temperature in degrees Celsius.

jog_run(direction: bool = True, fast: bool = False) → *None*

Slowly turn the motor in positive direction.

jog_stop() → *None*

Stop turning the motor in Jog mode.

quickstop() → *None*

Stops the motor with high deceleration rate and falls into error state. Reset with `reset_error` to recover into normal state.

reset_error() → *None*

Resets the motor into normal state after quick stop or another error occurred.

set_jog_speed(slow: int = 60, fast: int = 180) → *None*

Set the speed for jog mode. Default values correspond to startup values of the motor.

Parameters

- **slow** – RPM for slow jog mode.
- **fast** – RPM for fast jog mode.

set_max_acceleration(rpm_minute: int) → *None*

Set the maximum acceleration of the motor.

Parameters **rpm_minute** – revolution per minute per minute

set_max_deceleration(rpm_minute: int) → *None*

Set the maximum deceleration of the motor.

Parameters `rpm_minute` – revolution per minute per minute

`set_max_rpm(rpm: int) → None`

Set the maximum RPM.

Parameters `rpm` – revolution per minute ($0 < rpm \leq \text{RPM_MAX}$)

Raises `ILS2TException` – if RPM is out of range

`set_ramp_type(ramp_type: int = -1) → None`

Set the ramp type. There are two options available: 0: linear ramp -1: motor optimized ramp

Parameters `ramp_type` – 0: linear ramp | -1: motor optimized ramp

`start() → None`

Start this device.

`stop() → None`

Stop this device. Disables the motor (applies brake), disables access and closes the communication protocol.

`user_steps(steps: int = 16384, revolutions: int = 1) → None`

Define steps per revolution. Default is 16384 steps per revolution. Maximum precision is 32768 steps per revolution.

Parameters

- **steps** – number of steps in *revolutions*.
- **revolutions** – number of revolutions corresponding to *steps*.

`write_absolute_position(position: int) → None`

Write instruction to turn the motor until it reaches the absolute position. This function does not enable or disable the motor automatically.

Parameters `position` – absolute position of motor in user defined steps.

`write_relative_step(steps: int) → None`

Write instruction to turn the motor the relative amount of steps. This function does not enable or disable the motor automatically.

Parameters `steps` – Number of steps to turn the motor.

```
class ILS2TConfig(rpm_max_init: numbers.Integral = 1500, wait_sec_post_enable: Union[int, float] = 1,
                  wait_sec_max_disable: Union[int, float] = 10, wait_sec_post_cannot_disable: Union[int,
                  float] = 1, wait_sec_post_relative_step: Union[int, float] = 2,
                  wait_sec_post_absolute_position: Union[int, float] = 2)
```

Bases: object

Configuration for the ILS2T stepper motor device.

`clean_values()`

`force_value(fieldname, value)`

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define `post_force_value` method with same signature as this method to do extra processing after `value` has been forced on `fieldname`.

Parameters

- **fieldname** – name of the field

- **value** – value to assign

is_configdataclass = True

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

rpm_max_init: numbers.Integral = 1500

initial maximum RPM for the motor, can be set up to 3000 RPM. The user is allowed to set a new max RPM at runtime using [ILS2T.set_max_rpm\(\)](#), but the value must never exceed this configuration setting.

wait_sec_max_disable: Union[int, float] = 10

wait_sec_post_absolute_position: Union[int, float] = 2

wait_sec_post_cannot_disable: Union[int, float] = 1

wait_sec_post_enable: Union[int, float] = 1

wait_sec_post_relative_step: Union[int, float] = 2

exception ILS2TException

Bases: [hvl_ccb.dev.base.DeviceException](#)

Exception to indicate problems with the SE ILS2T stepper motor.

class ILS2TModbusTcpCommunication(configuration)

Bases: [hvl_ccb.comm.modbus_tcp.ModbusTcpCommunication](#)

Specific implementation of Modbus/TCP for the Schneider Electric ILS2T stepper motor.

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

class ILS2TModbusTcpCommunicationConfig(host: Union[str, ipaddress.IPv4Address, ipaddress.IPv6Address], unit: int = 255, port: int = 502)

Bases: [hvl_ccb.comm.modbus_tcp.ModbusTcpCommunicationConfig](#)

Configuration dataclass for Modbus/TCP communication specific for the Schneider Electric ILS2T stepper motor.

force_value(fieldname, value)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field

- **value** – value to assign

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

unit: int = 255

The unit has to be 255 such that IO scanning mode works.

class ILS2TRegAddr(*value*)

Bases: enum.IntEnum

Modbus Register Adresses for for Schneider Electric ILS2T stepper drive.

ACCESS_ENABLE = 282

FLT_INFO = 15362

FLT_MEM_DEL = 15112

FLT_MEM_RESET = 15114

IO_SCANNING = 6922

JOGN_FAST = 10506

JOGN_SLOW = 10504

POSITION = 7706

RAMP_ACC = 1556

RAMP_DECEL = 1558

RAMP_N_MAX = 1554

RAMP_TYPE = 1574

SCALE = 1550

TEMP = 7200

VOLT = 7198

class ILS2TRegDatatype(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)

Bases: aenum.Enum

Modbus Register Datatypes for Schneider Electric ILS2T stepper drive.

From the manual of the drive:

datatype	byte	min	max
INT8	1 Byte	-128	127
UINT8	1 Byte	0	255
INT16	2 Byte	-32_768	32_767
UINT16	2 Byte	0	65_535
INT32	4 Byte	-2_147_483_648	2_147_483_647
UINT32	4 Byte	0	4_294_967_295
BITS	just 32bits	N/A	N/A

INT32 = (-2147483648, 2147483647)

is_in_range(value: int) → bool

exception IoScanningModeValueError

Bases: *hvl_ccb.dev.se_ils2t.ILS2TException*

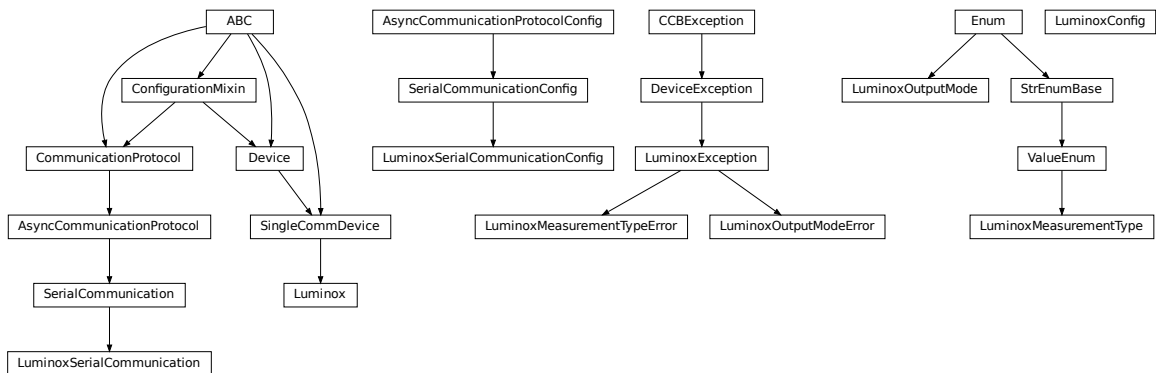
Exception to indicate that the selected IO scanning mode is invalid.

exception ScalingFactorValueError

Bases: *hvl_ccb.dev.se_ils2t.ILS2TException*

Exception to indicate that a scaling factor value is invalid.

hvl_ccb.dev.sst_luminox



Device class for a SST Luminox Oxygen sensor. This device can measure the oxygen concentration between 0 % and 25 %.

Furthermore, it measures the barometric pressure and internal temperature. The device supports two operating modes: in streaming mode the device measures all parameters every second, in polling mode the device measures only after a query.

Technical specification and documentation for the device can be found at the manufacturer's page: <https://www.sstsensing.com/product/luminox-optical-oxygen-sensors-2/>

class Luminox(com, dev_config=None)

Bases: *hvl_ccb.dev.base.SingleCommDevice*

Luminox oxygen sensor device class.

activate_output(mode: *hvl_ccb.dev.sst_luminox.LuminoxOutputMode*) → None

activate the selected output mode of the Luminox Sensor. :param mode: polling or streaming

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

static default_com_cls()

Get the class for the default communication protocol used with this device.

Returns the type of the standard communication protocol for this device

query_polling(*measurement*: Union[str, [hvl_ccb.dev.sst_luminex.LuminexMeasurementType](#)]) → Union[Dict[Union[str, [hvl_ccb.dev.sst_luminex.LuminexMeasurementType](#)], Union[float, int, str]], float, int, str]

Query a value or values of Luminex measurements in the polling mode, according to a given measurement type.

Parameters *measurement* – type of measurement

Returns value of requested measurement

Raises

- **ValueError** – when a wrong key for [LuminexMeasurementType](#) is provided
- **LuminexOutputModeError** – when polling mode is not activated
- **LuminexMeasurementTypeError** – when expected measurement value is not read

read_streaming() → Dict[Union[str, [hvl_ccb.dev.sst_luminex.LuminexMeasurementType](#)], Union[float, int, str]]

Read values of Luminex in the streaming mode. Convert the single string into separate values.

Returns dictionary with [LuminexMeasurementType.all_measurements_types\(\)](#) keys and accordingly type-parsed values.

Raises

- **LuminexOutputModeError** – when streaming mode is not activated
- **LuminexMeasurementTypeError** – when any of expected measurement values is not read

start() → *None*

Start this device. Opens the communication protocol.

stop() → *None*

Stop the device. Closes also the communication protocol.

class LuminexConfig(*wait_sec_post_activate*: Union[int, float] = 0.5, *wait_sec_trials_activate*: Union[int, float] = 0.1, *nr_trials_activate*: int = 5)

Bases: object

Configuration for the SST Luminex oxygen sensor.

clean_values()

force_value(*fieldname*, *value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field

- **value** – value to assign

is_configdataclass = True

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

nr_trials_activate: int = 5

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

wait_sec_post_activate: Union[int, float] = 0.5

wait_sec_trials_activate: Union[int, float] = 0.1

exception LuminoxException

Bases: [hvl_ccb.dev.base.DeviceException](#)

General Exception for Luminox Device.

class LuminoxMeasurementType(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)

Bases: [hvl_ccb.utils.enum.ValueEnum](#)

Measurement types for *LuminoxOutputMode.polling*.

The *all_measurements* type will read values for the actual measurement types as given in *LuminoxOutputMode.all_measurements_types()*; it parses multiple single values using regexp's for other measurement types, therefore, no regexp is defined for this measurement type.

all_measurements = 'A'

classmethod all_measurements_types() → Tuple[[hvl_ccb.dev.sst_luminox.LuminoxMeasurementType](#), ...]

A tuple of *LuminoxMeasurementType* enum instances which are actual measurements, i.e. not date of manufacture or software revision.

barometric_pressure = 'P'

property command: str

date_of_manufacture = '# 0'

parse_read_measurement_value(read_txt: str) → Union[Dict[Union[str, [hvl_ccb.dev.sst_luminox.LuminoxMeasurementType](#)], Union[float, int, str]], float, int, str]

partial_pressure_o2 = 'O'

percent_o2 = '%'

sensor_status = 'e'

serial_number = '# 1'

```
software_revision = '# 2'
```

```
temperature_sensor = 'T'
```

LuminoxMeasurementTypeDict

A typing hint for a dictionary holding LuminoxMeasurementType values. Keys are allowed as strings because *LuminoxMeasurementType* is of a *StrEnumBase* type.

alias of Dict[Union[str, *LuminoxMeasurementType*], Union[float, int, str]]

exception LuminoxMeasurementTypeError

Bases: *hvl_ccb.dev.sst_luminox.LuminoxException*

Wrong measurement type for requested data

LuminoxMeasurementTypeValue

A typing hint for all possible LuminoxMeasurementType values as read in either streaming mode or in a polling mode with *LuminoxMeasurementType.all_measurements*.

Beware: has to be manually kept in sync with *LuminoxMeasurementType* instances *cast_type* attribute values.

alias of Union[float, int, str]

class LuminoxOutputMode(value)

Bases: enum.Enum

output mode.

```
polling = 1
```

```
streaming = 0
```

exception LuminoxOutputModeError

Bases: *hvl_ccb.dev.sst_luminox.LuminoxException*

Wrong output mode for requested data

class LuminoxSerialCommunication(configuration)

Bases: *hvl_ccb.comm.serial.SerialCommunication*

Specific communication protocol implementation for the SST Luminox oxygen sensor. Already predefines device-specific protocol parameters in config.

```
static config_cls()
```

Return the default configdataclass class.

Returns a reference to the default configdataclass class

```
class LuminoxSerialCommunicationConfig(terminator: bytes = b'\r\n', encoding: str = 'utf-8',
                                       encoding_error_handling: str = 'strict',
                                       wait_sec_read_text_nonempty: Union[int, float] = 0.5,
                                       default_n_attempts_read_text_nonempty: int = 10, port:
                                       Union[str, NoneType] = None, baudrate: int = 9600, parity:
                                       Union[str, hvl_ccb.comm.serial.SerialCommunicationParity] =
                                       <SerialCommunicationParity.NONE: 'N'>, stopbits: Union[int,
                                       hvl_ccb.comm.serial.SerialCommunicationStopbits] =
                                       <SerialCommunicationStopbits.ONE: 1>, bytesize: Union[int,
                                       hvl_ccb.comm.serial.SerialCommunicationBytesize] =
                                       <SerialCommunicationBytesize.EIGHTBITS: 8>, timeout:
                                       Union[int, float] = 3)
```

Bases: *hvl_ccb.comm.serial.SerialCommunicationConfig*

```
baudrate: int = 9600
```

Baudrate for SST Luminox is 9600 baud

bytesize: `Union[int, hvl_ccb.comm.serial.SerialCommunicationBytesize] = 8`

One byte is eight bits long

force_value(*fieldname*, *value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

parity: `Union[str, hvl_ccb.comm.serial.SerialCommunicationParity] = 'N'`

SST Luminos does not use parity

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

stopbits: `Union[int, hvl_ccb.comm.serial.SerialCommunicationStopbits] = 1`

SST Luminos does use one stop bit

terminator: `bytes = b'\r\n'`

The terminator is CR LF

timeout: `Union[int, float] = 3`

use 3 seconds timeout as default

[hvl_ccb.dev.utils](#)

Poller

class Poller(*poll_handler*: Callable, *polling_delay_sec*: Union[int, float] = 0, *polling_interval_sec*: Union[int, float] = 1, *polling_timeout_sec*: Optional[Union[int, float]] = None)

Bases: object

Poller class wrapping *concurrent.futures.ThreadPoolExecutor* which enables passing of results and errors out of the polling thread.

is_polling() → bool

Check if device status is being polled.

Returns *True* when polling thread is set and alive

start_polling() → bool

Start polling.

Returns *True* if was not polling before, *False* otherwise

stop_polling() → bool

Stop polling.

Wait for until polling function returns a result as well as any exception that might have been raised within a thread.

Returns *True* if was polling before, *False* otherwise, and last result of the polling function call.

Raises polling function exceptions

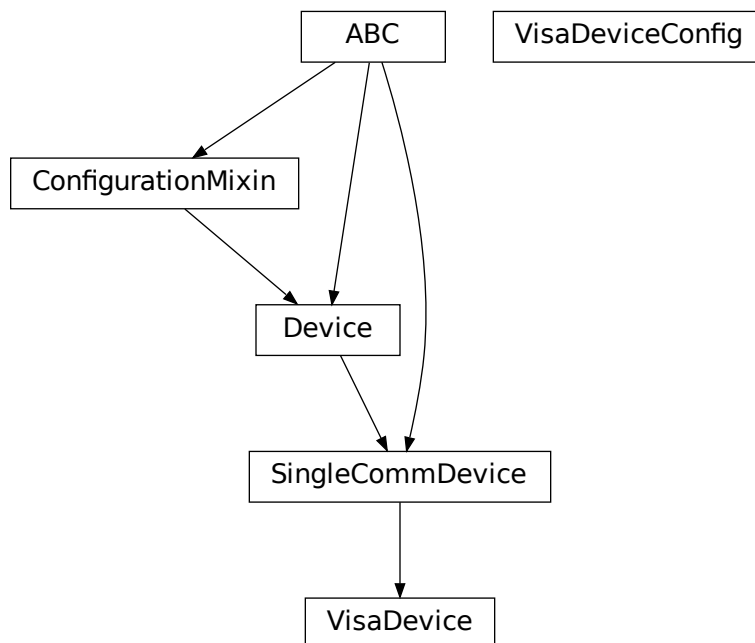
wait_for_polling_result()

Wait for until polling function returns a result as well as any exception that might have been raised within a thread.

Returns polling function result

Raises polling function errors

[hvl_ccb.dev.visa](#)



```
class VisaDevice(com: Union[hvl_ccb.comm.visa.VisaCommunication,  
                           hvl_ccb.comm.visa.VisaCommunicationConfig, dict], dev_config:  
                           Optional[Union[hvl_ccb.dev.visa.VisaDeviceConfig, dict]] = None)
```

Bases: *hvl_ccb.dev.base.SingleCommDevice*

Device communicating over the VISA protocol using VisaCommunication.

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

static default_com_cls() → Type[hvl_ccb.comm.visa.VisaCommunication]

Return the default communication protocol for this device type, which is VisaCommunication.

Returns the VisaCommunication class

get_error_queue() → str

Read out error queue and logs the error.

Returns Error string

get_identification() → str

Queries “*IDN?” and returns the identification string of the connected device.

Returns the identification string of the connected device

reset() → *None*

Send “*RST” and “*CLS” to the device. Typically sets a defined state.

spoll_handler()

Reads the status byte and decodes it. The status byte STB is defined in IEEE 488.2. It provides a rough overview of the instrument status.

Returns

start() → *None*

Start the VisaDevice. Sets up the status poller and starts it.

Returns

stop() → *None*

Stop the VisaDevice. Stops the polling thread and closes the communication protocol.

Returns

wait_operation_complete(*timeout*: Optional[float] = None) → bool

Waits for a operation complete event. Returns after timeout [s] has expired or the operation complete event has been caught.

Parameters **timeout** – Time in seconds to wait for the event; *None* for no timeout.

Returns True, if OPC event is caught, False if timeout expired

```
class VisaDeviceConfig(spoll_interval: Union[int, float] = 0.5, spoll_start_delay: Union[int, float] = 2)  
Bases: hvl_ccb.dev.visa._VisaDeviceConfigDefaultsBase, hvl_ccb.dev.visa.  
_VisaDeviceConfigBase
```

Configdataclass for a VISA device.

force_value(*fieldname*, *value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

classmethod **keys()** → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod **optional_defaults()** → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

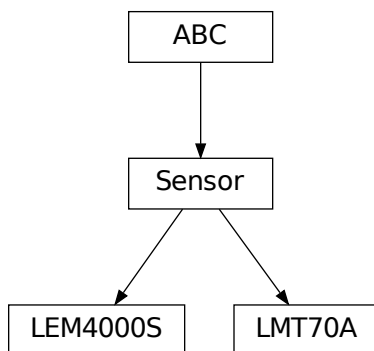
classmethod **required_keys()** → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

Module contents

Devices subpackage.

hvl_ccb.utils**Submodules****hvl_ccb.utils.conversion_sensor**

Sensors that are used by the devices implemented in the CCB

class **LEM4000S**(*shunt: float = 1.2, calibration_factor: float = 1*)

Bases: [hvl_ccb.utils.conversion_sensor.Sensor](#)

Converts the output voltage (V) to the measured current (A) when using a LEM Current transducer LT 4000-S

```
CONVERSION: ClassVar[int] = 5000
```

```
calibration_factor: float = 1
```

```
convert(value, **kwargs)
```

```
shunt: float = 1.2
```

```
class LMT70A(temperature_unit: hvl_ccb.utils.conversion_unit.Temperature = Temperature.C)
```

Bases: [hvl_ccb.utils.conversion_sensor.Sensor](#)

Converts the output voltage (V) to the measured temperature (default °C) when using a TI Precision Analog Temperature Sensor LMT70(A)

```
LUT: ClassVar[numpy.ndarray[Any,
numpy.dtype[numpy.typing._generic_alias.ScalarType]]] = array([[ -55. ,  1.375219],
[ -50. ,  1.350441], [ -40. ,  1.300593], [ -30. ,  1.250398], [ -20. ,  1.199884], [ -10. ,
  1.14907 ], [  0. ,  1.097987], [ 10. ,  1.046647], [ 20. ,  0.99505 ], [ 30. ,
  0.943227], [ 40. ,  0.891178], [ 50. ,  0.838882], [ 60. ,  0.78636 ], [ 70. ,
  0.733608], [ 80. ,  0.680654], [ 90. ,  0.62749 ], [100. ,  0.574117], [110. ,
  0.520551], [120. ,  0.46676 ], [130. ,  0.412739], [140. ,  0.358164], [150. ,
  0.302785]])
```

```
convert(value, **kwargs)
```

```
temperature_unit: hvl\_ccb.utils.conversion\_unit.Temperature = 'C'
```

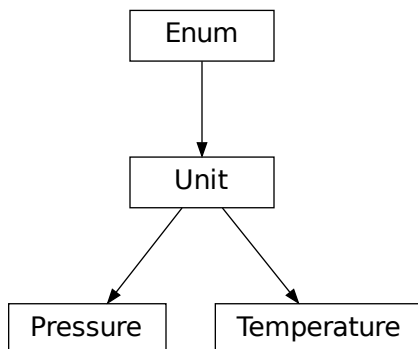
```
class Sensor
```

Bases: [abc.ABC](#)

The BaseClass ‘Sensor’ is designed as a parent for all Sensors. Each attribute must be added to ‘__setattr__’, so that the value is verified each time the value is changed. It is important to mark attributes that should be constant with ‘typing.ClassVar[...]’. Together with ‘super().__setattr__(name, value)’, this guarantees that the values are protected and cannot be altered by the user.

```
abstract convert(value, **kwargs)
```

[hvl_ccb.utils.conversion_unit](#)



example Kelvin <-> Celsius

Unit conversion, within in the same group of units, for

class Pressure(*value*)

Bases: *hvl_ccb.utils.conversion_unit.Unit*

An enumeration.

ATM = 'atm'

ATMOSPHERE = 'atm'

BAR = 'bar'

MILLIMETER_MERCURY = 'mmHg'

MMHG = 'mmHg'

PA = 'Pa'

PASCAL = 'Pa'

POUNDS_PER_SQUARE_INCH = 'psi'

PSI = 'psi'

TORR = 'torr'

class Temperature(*value*)

Bases: *hvl_ccb.utils.conversion_unit.Unit*

An enumeration.

C = 'C'

CELSIUS = 'C'

F = 'F'

FAHRENHEIT = 'F'

K = 'K'

KELVIN = 'K'

class Unit(*value*)

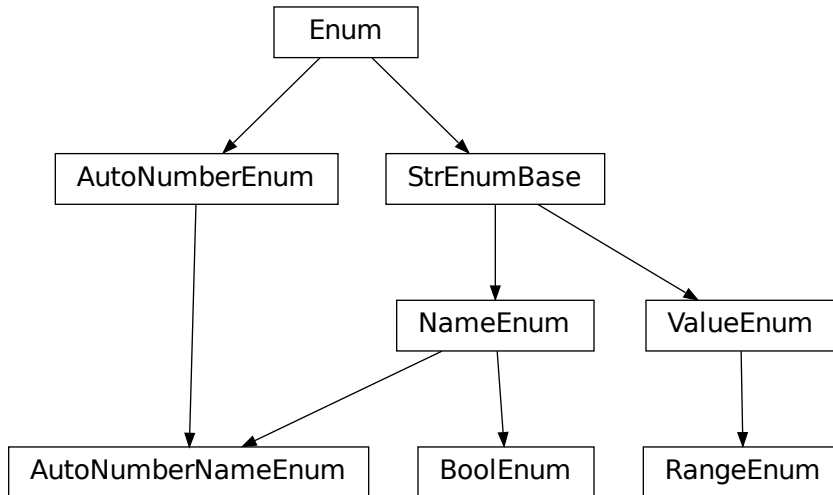
Bases: *enum.Enum*

An enumeration.

abstract classmethod convert(*value*, ***kwargs*)

preserve_type(*func*)

This wrapper preserves the first order type of the input. Upto now the type of the data stored in a list, tuple, array or dict is not preserved. Integer will be converted to float!

hvl_ccb.utils.enum

```
class AutoNumberNameEnum(value=<no_arg>, names=None, module=None, type=None, start=1,
                          boundary=None)
```

Bases: `hvl_ccb.utils.enum.NameEnum`, `aenum.AutoNumberEnum`

Auto-numbered enum with names used as string representation, and with lookup and equality based on this representation.

```
class BoolEnum(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)
```

Bases: `hvl_ccb.utils.enum.NameEnum`

BoolEnum inherits from NameEnum and the type of the first value is enforced to be 'boolean'. For `bool()`-operation the `__bool__` is redefined here.

```
class NameEnum(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)
```

Bases: `hvl_ccb.utils.enum.StrEnumBase`

Enum with names used as string representation, and with lookup and equality based on this representation.

```
class RangeEnum(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)
```

Bases: `float`, `hvl_ccb.utils.enum.ValueEnum`

Range enumeration inherit from `ValueEnum`, find suitable voltage/current/resistance input range for devices such as multimeter and oscilloscope

```
abstract classmethod unit() → str
```

Returns the Unit of the values in the enumeration. :return: the unit of the values in the enumeration in string format

```
class StrEnumBase(value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None)
```

Bases: `aenum.Enum`

String representation-based equality and lookup.

class ValueEnum(*value=<no_arg>, names=None, module=None, type=None, start=1, boundary=None*)

Bases: `hvl_ccb.utils.enum.StrEnumBase`

Enum with string representation of values used as string representation, and with lookup and equality based on this representation.

Attention: to avoid errors, best use together with *unique* enum decorator.

hvl_ccb.utils.typing

Additional Python typing module utilities

ConvertibleTypes

Typing hint for data type that can be used in conversion

alias of `Union[int, float, List[Union[int, float]], Tuple[Union[int, float]], Dict[str, Union[int, float]], numpy.ndarray]`

Number

Typing hint auxiliary for a Python base number types: *int* or *float*.

alias of `Union[int, float]`

check_generic_type(*value, type_, name='instance'*)

Check if *value* is of a generic type *type_*. Raises *TypeError* if it's not.

Parameters

- **name** – name to report in case of an error
- **value** – value to check
- **type** – generic type to check against

is_generic_type_hint(*type_*)

Check if class is a generic type, for example *Union[int, float]*, *List[int]*, or *List*.

Parameters **type** – type to check

hvl_ccb.utils.validation

validate_and_resolve_host(*host: Union[str, ipaddress.IPv4Address, ipaddress.IPv6Address], logger: Optional[logging.Logger] = None*) → str

validate_bool(*x_name: str, x: object, logger: Optional[logging.Logger] = None*) → *None*

Validate if given input *x* is a *bool*.

Parameters

- **x_name** – string name of the validate input, use for the error message
- **x** – an input object to validate as boolean
- **logger** – logger of the calling submodule

Raises **TypeError** – when the validated input does not have boolean type

validate_number(*x_name*: str, *x*: object, *limits*: typing.Optional[typing.Tuple] = (None, None), *number_type*: typing.Union[typing.Type[typing.Union[int, float]], typing.Tuple[typing.Type[typing.Union[int, float]], ...]] = (<class 'int'>, <class 'float'>), *logger*: typing.Optional[logging.Logger] = None) → None

Validate if given input *x* is a number of given *number_type* type, with value between given *limits*[0] and *limits*[1] (inclusive), if not *None*. For array-like objects (npt.NDArray, List, Tuple, Dict) it is checked if all elements are within the limits and have the correct type.

Parameters

- **x_name** – string name of the validate input, use for the error message
- **x** – an input object to validate as number of given type within given range
- **logger** – logger of the calling submodule
- **limits** – [lower, upper] limit, with *None* denoting no limit: [-inf, +inf]
- **number_type** – expected type or tuple of types of a number, by default (*int*, *float*)

Raises

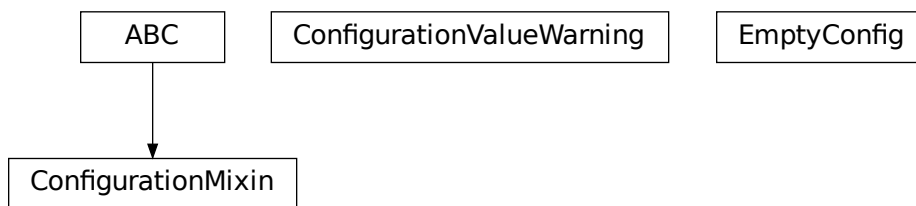
- **TypeError** – when the validated input does not have expected type
- **ValueError** – when the validated input has correct number type but is not within given range

validate_tcp_port(*port*: int, *logger*: Optional[logging.Logger] = None)

Module contents

4.1.2 Submodules

hvl_ccb.configuration



Facilities

providing classes for handling configuration for communication protocols and devices.

class ConfigurationMixin(*configuration*)

Bases: `abc.ABC`

Mixin providing configuration to a class.

property config

ConfigDataclass property.

Returns the configuration

abstract static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

configuration_save_json(*path: str*) → *None*

Save current configuration as JSON file.

Parameters **path** – path to the JSON file.

classmethod from_json(*filename: str*)

Instantiate communication protocol using configuration from a JSON file.

Parameters **filename** – Path and filename to the JSON configuration

exception ConfigurationValueWarning

Bases: UserWarning

User warnings category for values of *@configdataclass* fields.

class EmptyConfig

Bases: object

Empty configuration dataclass.

clean_values()

Cleans and enforces configuration values. Does nothing by default, but may be overridden to add custom configuration value checks.

force_value(*fieldname, value*)

Forces a value to a dataclass field despite the class being frozen.

NOTE: you can define *post_force_value* method with same signature as this method to do extra processing after *value* has been forced on *fieldname*.

Parameters

- **fieldname** – name of the field
- **value** – value to assign

is_configdataclass = True

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

Returns a list of strings containing all optional keys.

classmethod required_keys() → Sequence[str]

Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns a list of strings containing all required keys.

configdataclass(*direct_decoration=None, frozen=True*)

Decorator to make a class a configdataclass. Types in these dataclasses are enforced. Implement a function *clean_values(self)* to do additional checking on value ranges etc.

It is possible to inherit from a configdataclass and re-decorate it with `@configdataclass`. In a subclass, default values can be added to existing fields. Note: adding additional non-default fields is prone to errors, since the order has to be respected through the whole chain (first non-default fields, only then default-fields).

Parameters **frozen** – defaults to True. False allows to later change configuration values. Attention: if configdataclass is not frozen and a value is changed, typing is not enforced anymore!

`hvl_ccb.exception`

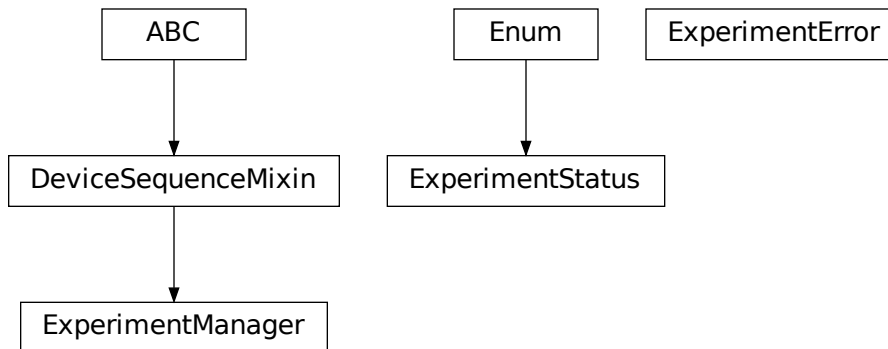
CCBException

Introduce a common base exception for the CCB

exception **CCBException**

Bases: Exception

`hvl_ccb.experiment_manager`



Main module containing the top level `ExperimentManager` class. Inherit from this class to implement your own experiment functionality in another project and it will help you start, stop and manage your devices.

exception **ExperimentError**

Bases: Exception

Exception to indicate that the current status of the experiment manager is on ERROR and thus no operations can be made until reset.

class **ExperimentManager**(*args, **kwargs)

Bases: `hvl_ccb.dev.base.DeviceSequenceMixin`

Experiment Manager can start and stop communication protocols and devices. It provides methods to queue commands to devices and collect results.

add_device(*name*: str, *device*: [hvl_ccb.dev.base.Device](#)) → *None*

Add a new device to the manager. If the experiment is running, automatically start the device. If a device with this name already exists, raise an exception.

Parameters

- **name** – is the name of the device.
- **device** – is the instantiated Device object.

Raises [DeviceExistingException](#) –

devices_failed_start: Dict[str, [hvl_ccb.dev.base.Device](#)]

Dictionary of named device instances from the sequence for which the most recent *start()* attempt failed.

Empty if *stop()* was called last; cf. *devices_failed_stop*.

devices_failed_stop: Dict[str, [hvl_ccb.dev.base.Device](#)]

Dictionary of named device instances from the sequence for which the most recent *stop()* attempt failed.

Empty if *start()* was called last; cf. *devices_failed_start*.

finish() → *None*

Stop experimental setup, stop all devices.

is_error() → bool

Returns true, if the status of the experiment manager is *error*.

Returns True if on error, false otherwise

is_finished() → bool

Returns true, if the status of the experiment manager is *finished*.

Returns True if finished, false otherwise

is_running() → bool

Returns true, if the status of the experiment manager is *running*.

Returns True if running, false otherwise

run() → *None*

Start experimental setup, start all devices.

start() → *None*

Alias for ExperimentManager.run()

property status: [hvl_ccb.experiment_manager.ExperimentStatus](#)

Get experiment status.

Returns experiment status enum code.

stop() → *None*

Alias for ExperimentManager.finish()

class ExperimentStatus(*value*)

Bases: `enum.Enum`

Enumeration for the experiment status

ERROR = 5

FINISHED = 4

FINISHING = 3

INITIALIZED = 0

INITIALIZING = -1

RUNNING = 2

STARTING = 1

4.1.3 Module contents

Top-level package for HVL Common Code Base.

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given. You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://gitlab.com/ethz_hvl/hvl_ccb/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitLab issues for bugs. Anything tagged with “bug” and “help wanted” is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitLab issues for features. Anything tagged with “enhancement” and “help wanted” is open to whoever wants to implement it.

5.1.4 Write Documentation

HVL Common Code Base could always use more documentation, whether as part of the official HVL Common Code Base docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://gitlab.com/ethz_hvl/hvl_ccb/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *hvl_ccb* for local development.

1. Clone *hvl_ccb* repo from GitLab.

```
$ git clone git@gitlab.com:ethz_hvl/hvl_ccb.git
```

2. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv hvl_ccb
$ cd hvl_ccb/
$ pip install -e .[all]
$ pip install -r requirements_dev.txt
```

3. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

4. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 hvl_ccb tests
$ python setup.py test or py.test
$ tox
```

To get flake8 and tox, just pip install them into your virtualenv. You can also use the provided make-like shell script to run flake8 and tests:

```
$ ./make.sh lint
$ ./make.sh test
```

5. Commit your changes and push your branch to GitLab:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

6. Submit a merge request through the GitLab website.

5.3 Merge Request Guidelines

Before you submit a merge request, check that it meets these guidelines:

1. The merge request should include tests.
2. If the merge request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The merge request should work for Python 3.7. Check https://gitlab.com/ethz_hvl/hvl_ccb/merge_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

- To run tests from a single file:

```
$ py.test tests/test_hvl_ccb.py
```

or a single test function:

```
$ py.test tests/test_hvl_ccb.py::test_command_line_interface
```

- If your tests are slow, profile them using the pytest-profiling plugin:

```
$ py.test tests/test_hvl_ccb.py --profile
```

or for a graphical overview (you need a SVG image viewer):

```
$ py.test tests/test_hvl_ccb.py --profile-svg
$ open prof/combined.svg
```

- To add dependency, edit appropriate `*requirements` variable in the `setup.py` file and re-run:

```
$ python setup.py develop
```

- To generate a PDF version of the Sphinx documentation instead of HTML use:

```
$ rm -rf docs/hvl_ccb.rst docs/modules.rst docs/_build && sphinx-apidoc -o docs/hvl_
↪ccb && python -msphinx -M latexpdf docs/ docs/_build
```

This command can also be run through the make-like shell script:

```
$ ./make.sh docs-pdf
```

This requires a local installation of a LaTeX distribution, e.g. MikTeX.

5.5 Deploying

A reminder for the maintainers on how to deploy. Create `release-N.M.K` branch. Make sure all your changes are committed. Update or create entry in `HISTORY.rst` file, and, if applicable, update `AUTHORS.rst` file, update features tables in `README.rst` file, and update API docs:

```
$ make docs
```

Commit all of the above, except for the `docs/hvl_ccb.dev.picotech_pt104.rst`, and then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Go to <https://readthedocs.org/projects/hvl-ccb/builds/> and check if RTD docs build for the pushed tag passed.

Wait for the CI pipeline to finish successfully; afterwards, run a release check:

```
$ make release-check
```

Finally, prepare and push a release:

```
$ make release
```

Merge the release branch into master and devel branches with `--no-ff` flag and delete the release branch:

```
$ git switch master
$ git merge --no-ff release-N.M.K
$ git push
$ git switch devel
$ git merge --no-ff release-N.M.K
$ git push
$ git push --delete origin release-N.M.K
$ git branch --delete release-N.M.K
```

Finally, prepare GitLab release and cleanup the corresponding milestone:

1. go to https://gitlab.com/ethz_hvl/hvl_ccb/-/tags/, select the latest release tag, press “Edit release notes” and add the release notes (copy a corresponding entry from `HISTORY.rst` file with formatting adjusted from ReStructuredText to Markdown); press “Save changes”;
2. go to https://gitlab.com/ethz_hvl/hvl_ccb/-/releases, select the latest release, press “Edit this release” and under “Milestones” select the corresponding milestone; press “Save changes”;
3. go to https://gitlab.com/ethz_hvl/hvl_ccb/-/milestones, make sure that it is 100% complete (otherwise, create a next patch-level milestone and assign it to the ongoing Issues and Merge Requests therein); press “Close Milestone”.

CREDITS

6.1 Maintainers

- Mikołaj Rybiński <mikolaj.rybinski@id.ethz.ch>
- Henrik Menne <henrik.menne@eeh.ee.ethz.ch>
- Henning Janssen <janssen@eeh.ee.ethz.ch>
- Maria Del <maria.del@id.ethz.ch>

6.2 Authors

- Mikołaj Rybiński <mikolaj.rybinski@id.ethz.ch>
- David Graber <dev@davidgraber.ch>
- Henrik Menne <henrik.menne@eeh.ee.ethz.ch>
- Alise Chachereau <chachereau@eeh.ee.ethz.ch>
- Henning Janssen <janssen@eeh.ee.ethz.ch>
- David Taylor <dtaylor@ethz.ch>
- Joseph Engelbrecht <engelbrecht@eeh.ee.ethz.ch>

6.3 Contributors

- Luca Nembrini <lucane@student.ethz.ch>
- Maria Del <maria.del@id.ethz.ch>
- Raphael Faerber <raphael.ferber@eeh.ee.ethz.ch>
- Ruben Stadler <rstadler@student.ethz.ch>
- Hanut Vemulapalli <vemulapalli@eeh.ee.ethz.ch>
- Chi-Ching Hsu <hsu@ibi.baug.ethz.ch>

HISTORY

7.1 0.10.1 (2022-01-24)

- **several improvements and fixes for device cube:**
 - privatize Alarms and AlarmsOverview
 - fix list of cube alarms
 - improve docs
 - fix bugs with earthing sticks
 - fix bug in config dataclass of cube
- introduction of BoolEnum
- introduction of RangeEnum
- bumpversion -> bump2version

7.2 0.10.0 (2022-01-17)

- Reimplementation of the Cube (before known as Supercube)
- **new names:**
 - Supercube Typ B -> BaseCube
 - Supercube Typ A -> PICube (power inverter Cube)
- **new import:**
 - `from hvl_ccb.dev.supercube import SupercubeB -> from hvl_ccb.dev.cube import BaseCube`
- **new programming style:**
 - getter / setter methods -> properties
 - e.g. get: `cube.get_support_output(port=1, contact=1) -> cube.support_1.output_1`
 - e.g. set: `cube.get_support_output(port=1, contact=1, state=True) -> cube.support_1.output_1 = True`
- unify Exceptions of Cube
- implement Fast Switch-Off of Cube

- remove method `support_output_impulse`
- all active alarms can now be queried `cube.active_alarms()`
- alarms will now result in different logging levels depending on the seriousness of the alarm.
- introduction of limits for slope and safety limit for RedReady
- during the startup the CCB will update the time of the cube.
- verification of inputs
- polarity of DC voltage
- Switch from `python-opcua` to `opcua-asyncio` (former package is no longer maintained)

7.3 0.9.0 (2022-01-07)

- New device: Highland T560 digital delay and pulse generator over Telnet.
- **Rework of the Technix Capacitor Charger.**
 - Moved into a separate sub-package
 - NEW import over `import hvl_ccb.dev.technix as XXX`
 - Slightly adapted behaviour
- Add `validate_tcp_port` to validate port number.
- **Add `validate_and_resolve_host` to validate and resolve host names and IPs.**
 - Remove requirement IPy
- Add a unified CCB Exception schema for all devices and communication protocols.
- Add data conversion functions to README.
- Update CI and devel images from Debian 10 buster to Debian 11 bullseye.
- Fix typing due to numpy update.
- Fix incorrect overloading of `clean_values()` in classes of type `XCommunicationConfig`.

7.4 0.8.5 (2021-11-05)

- Added arbitrary waveform for TiePie signal generation, configurable via `dev.tiepie.generator.TiePieGeneratorConfig.waveform` property.
- In `utils.conversion_sensor`: improvements for class constants; removed SciPy dependency.
- Added Python 3.10 support.

7.5 0.8.4 (2021-10-22)

- `utils.validation.validate_number` extension to handle NumPy arrays and array-like objects.
- `utils.conversion_unit` utility classes handle correctly `NamedTuple` instances.
- `utils.conversion_sensor` and `utils.conversion_unit` code simplification (no `transfer_function_order` attribute) and cleanups.
- Fixed incorrect error logging in `configuration.configdataclass`.
- `comm.telnet.TelnetCommunication` tests fixes for local run errors.

7.6 0.8.3 (2021-09-27)

- New data conversion functions in `utils.conversion_sensor` and `utils.conversion_unit` modules. Note: to use these functions you must install `hvl_ccb` with extra requirement, either `hvl_ccb[conversion]` or `hvl_ccb[all]`.
- Improved documentation with respect to installation of external libraries.

7.7 0.8.2 (2021-08-27)

- **New functionality in `dev.labjack.LabJack`:**
 - configure clock and send timed pulse sequences
 - set DAC/analog output voltage
- Bugfix: ignore random bits sent by to `dev.newport.NewportSMC100PP` controller during start-up/powering-up.

7.8 0.8.1 (2021-08-13)

- Add Python version check (min version error; max version warning).
- Daily checks for upstream dependencies compatibility and devel environment improvements.

7.9 0.8.0 (2021-07-02)

- TCP communication protocol.
- Lauda PRO RP 245 E circulation thermostat device over TCP.
- Pico Technology PT-104 Platinum Resistance Data Logger device as a wrapper of the Python bindings for the PicoSDK.
- In `com.visa.VisaCommunication`: periodic status polling when VISA/TCP keep alive connection is not supported by a host.

7.10 0.7.1 (2021-06-04)

- New `utils.validation` submodule with `validate_bool` and `validate_number` utilities extracted from internal use within a `dev.tiepie` subpackage.
- In `comm.serial.SerialCommunication`:
 - strict encoding errors handling strategy for subclasses,
 - user warning for a low communication timeout value.

7.11 0.7.0 (2021-05-25)

- The `dev.tiepie` module was splitted into a subpackage with, in particular, submodules for each of the device types – `oscilloscope`, `generator`, and `i2c` – and with backward-incompatible direct imports from the submodules.
- In `dev.technix`:
 - fixed communication crash on nested status byte query;
 - added enums for GET and SET register commands.
- Further minor logging improvements: added missing module level logger and removed some error logs in `except` blocks used for a flow control.
- In `examples/` folder renamed consistently all the examples.
- In API documentation: fix incorrect links mapping on inheritance diagrams.

7.12 0.6.1 (2021-05-08)

- In `dev.tiepie`:
 - dynamically set `oscilloscope`'s channel limits in `OscilloscopeChannelParameterLimits`: `input_range` and `trigger_level_abs`, incl. update of latter on each change of `input_range` value of a `TiePieOscilloscopeChannelConfig` instances;
 - quick fix for opening of combined instruments by disabling `OscilloscopeParameterLimits.trigger_delay` (an advanced feature);
 - enable automatic devices detection to be able to find network devices with `TiePieOscilloscope.list_devices()`.
- Fix `examples/example_labjack.py`.
- Improved logging: consistently use module level loggers, and always log exception tracebacks.
- Improve API documentation: separate pages per modules, each with an inheritance diagram as an overview.

7.13 0.6.0 (2021-04-23)

- Technix capacitor charger using either serial connection or Telnet protocol.
- **Extensions, improvements and fixes in existing devices:**
 - In `dev.tiepie.TiePieOscilloscope`:
 - * redesigned measurement start and data collection API, incl. time out argument, with no/infinite time out option;
 - * trigger allows now a no/infinite time out;
 - * record length and trigger level were fixed to accept, respectively, floating point and integer numbers;
 - * fixed resolution validation bug;
 - `dev.heinzinger.HeinzingerDI` and `dev.rs_rto1024.RTO1024` instances are now resilient to multiple `stop()` calls.
 - In `dev.crylas.CryLasLaser`: default configuration timeout and polling period were adjusted;
 - Fixed PSI9080 example script.
- **Package and source code improvements:**
 - Update to backward-incompatible `pyvisa-py` $\geq 0.5.2$. Developers, do update your local development environments!
 - External libraries, like LibTiePie SDK or LJM Library, are now not installed by default; they are now extra installation options.
 - Added Python 3.9 support.
 - Improved number formatting in logs.
 - Typing improvements and fixes for `mypy` ≥ 0.800 .

7.14 0.5.0 (2020-11-11)

- TiePie USB oscilloscope, generator and I2C host devices, as a wrapper of the Python bindings for the LibTiePie SDK.
- a FuG Elektronik Power Supply (e.g. Capacitor Charger HCK) using the built-in ADDAT controller with the Probus V protocol over a serial connection
- All devices polling status or measurements use now a `dev.utils.Poller` utility class.
- **Extensions and improvements in existing devices:**
 - In `dev.rs_rto1024.RTO1024`: added Channel state, scale, range, position and offset accessors, and measurements activation and read methods.
 - In `dev.sst_luminox.Luminox`: added querying for all measurements in polling mode, and made output mode activation more robust.
 - In `dev.newport.NewportSMC100PP`: an error-prone `wait_until_move_finished` method of replaced by a fixed waiting time, device operations are now robust to a power supply cut, and device restart is not required to apply a start configuration.
- **Other minor improvements:**

- Single failure-safe starting and stopping of devices sequenced via `dev.base.DeviceSequenceMixin`.
- Moved `read_text_nonempty` up to `comm.serial.SerialCommunication`.
- Added development Dockerfile.
- Updated package and development dependencies: `pymodbus`, `pytest-mock`.

7.15 0.4.0 (2020-07-16)

- **Significantly improved new Supercube device controller:**
 - more robust error-handling,
 - status polling with generic `Poller` helper,
 - messages and status boards.
 - tested with a physical device,
- Improved OPC UA client wrapper, with better error handling, incl. re-tries on `concurrent.futures.TimeoutError`.
- SST Luminex Oxygen sensor device controller.
- **Backward-incompatible changes:**
 - `CommunicationProtocol.access_lock` has changed type from `threading.Lock` to `threading.RLock`.
 - `ILS2T.relative_step` and `ILS2T.absolute_position` are now called, respectively, `ILS2T.write_relative_step` and `ILS2T.write_absolute_position`.
- **Minor bugfixes and improvements:**
 - fix use of max resolution in `Labjack.set_ain_resolution()`,
 - resolve ILS2T devices relative and absolute position setters race condition,
 - added acoustic horn function in the 2015 Supercube.
- **Toolchain changes:**
 - add Python 3.8 support,
 - drop `pytest-runner` support,
 - ensure compatibility with `labjack_ljm` 2019 version library.

7.16 0.3.5 (2020-02-18)

- Fix issue with reading integers from LabJack LJM Library (device's product ID, serial number etc.)
- Fix development requirements specification (tox version).

7.17 0.3.4 (2019-12-20)

- **New devices using serial connection:**
 - Heinzinger Digital Interface I/II and a Heinzinger PNC power supply
 - Q-switched Pulsed Laser and a laser attenuator from CryLas
 - Newport SMC100PP single axis motion controller for 2-phase stepper motors
 - Pfeiffer TPG controller (TPG 25x, TPG 26x and TPG 36x) for Compact pressure Gauges
- PEP 561 compatibility and related corrections for static type checking (now in CI)
- **Refactorings:**
 - Protected non-thread safe read and write in communication protocols
 - Device sequence mixin: start/stop, add/rm and lookup
 - *.format()* to f-strings
 - more enumerations and a quite some improvements of existing code
- Improved error docstrings (`:raises:` annotations) and extended tests for errors.

7.18 0.3.3 (2019-05-08)

- Use PyPI labjack-ljm (no external dependencies)

7.19 0.3.2 (2019-05-08)

- INSTALLATION.rst with LJMPython prerequisite info

7.20 0.3.1 (2019-05-02)

- readthedocs.org support

7.21 0.3 (2019-05-02)

- Prevent an automatic close of VISA connection when not used.
- Rhode & Schwarz RTO 1024 oscilloscope using VISA interface over `TCP::INSTR`.
- Extended tests incl. messages sent to devices.
- Added Supercube device using an OPC UA client
- Added Supercube 2015 device using an OPC UA client (for interfacing with old system version)

7.22 0.2.1 (2019-04-01)

- Fix issue with LJMPython not being installed automatically with setuptools.

7.23 0.2.0 (2019-03-31)

- LabJack LJM Library communication wrapper and LabJack device.
- Modbus TCP communication protocol.
- Schneider Electric ILS2T stepper motor drive device.
- Elektro-Automatik PSI9000 current source device and VISA communication wrapper.
- Separate configuration classes for communication protocols and devices.
- Simple experiment manager class.

7.24 0.1.0 (2019-02-06)

- Communication protocol base and serial communication implementation.
- Device base and MBW973 implementation.

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

h

- [hvl_ccb](#), 182
- [hvl_ccb.comm](#), 33
 - [hvl_ccb.comm.base](#), 11
 - [hvl_ccb.comm.labjack_ljm](#), 15
 - [hvl_ccb.comm.modbus_tcp](#), 18
 - [hvl_ccb.comm.opc](#), 20
 - [hvl_ccb.comm.serial](#), 23
 - [hvl_ccb.comm.tcp](#), 26
 - [hvl_ccb.comm.telnet](#), 28
 - [hvl_ccb.comm.visa](#), 30
- [hvl_ccb.configuration](#), 178
- [hvl_ccb.dev](#), 173
 - [hvl_ccb.dev.base](#), 83
 - [hvl_ccb.dev.crylas](#), 85
 - [hvl_ccb.dev.cube](#), 44
 - [hvl_ccb.dev.cube.base](#), 33
 - [hvl_ccb.dev.cube.constants](#), 38
 - [hvl_ccb.dev.cube.picube](#), 41
 - [hvl_ccb.dev.ea_psi9000](#), 95
 - [hvl_ccb.dev.fug](#), 99
 - [hvl_ccb.dev.heinzinger](#), 110
 - [hvl_ccb.dev.highland_t560](#), 49
 - [hvl_ccb.dev.highland_t560.base](#), 44
 - [hvl_ccb.dev.highland_t560.channel](#), 46
 - [hvl_ccb.dev.highland_t560.device](#), 47
 - [hvl_ccb.dev.labjack](#), 116
 - [hvl_ccb.dev.lauda](#), 123
 - [hvl_ccb.dev.mbw973](#), 129
 - [hvl_ccb.dev.newport](#), 133
 - [hvl_ccb.dev.pfeiffer_tpg](#), 146
 - [hvl_ccb.dev.rs_rto1024](#), 152
 - [hvl_ccb.dev.se_ils2t](#), 160
 - [hvl_ccb.dev.sst_luminos](#), 166
 - [hvl_ccb.dev.supercube2015](#), 64
 - [hvl_ccb.dev.supercube2015.base](#), 49
 - [hvl_ccb.dev.supercube2015.constants](#), 55
 - [hvl_ccb.dev.supercube2015.typ_a](#), 62
 - [hvl_ccb.dev.technix](#), 70
 - [hvl_ccb.dev.technix.base](#), 65
 - [hvl_ccb.dev.technix.device](#), 68
 - [hvl_ccb.dev.tiepie](#), 82
 - [hvl_ccb.dev.tiepie.base](#), 71
 - [hvl_ccb.dev.tiepie.channel](#), 73
 - [hvl_ccb.dev.tiepie.device](#), 75
 - [hvl_ccb.dev.tiepie.generator](#), 76
 - [hvl_ccb.dev.tiepie.i2c](#), 78
 - [hvl_ccb.dev.tiepie.oscilloscope](#), 79
 - [hvl_ccb.dev.tiepie.utils](#), 81
 - [hvl_ccb.dev.utils](#), 170
 - [hvl_ccb.dev.visa](#), 171
 - [hvl_ccb.exception](#), 180
 - [hvl_ccb.experiment_manager](#), 180
 - [hvl_ccb.utils](#), 178
 - [hvl_ccb.utils.conversion_sensor](#), 173
 - [hvl_ccb.utils.conversion_unit](#), 174
 - [hvl_ccb.utils.enum](#), 176
 - [hvl_ccb.utils.typing](#), 177
 - [hvl_ccb.utils.validation](#), 177

A

- A (*HeinzingerPNC.UnitCurrent* attribute), 113
- A (*SupercubeOpcEndpoint* attribute), 62
- ABSOLUTE (*TiePieOscilloscopeTriggerLevelMode* attribute), 75
- ABSOLUTE_POSITION (*ILS2T.ActionsPtp* attribute), 160
- AC (*NewportConfigCommands* attribute), 133
- AC_100KV (*PowerSetup* attribute), 40
- AC_150KV (*PowerSetup* attribute), 40
- AC_200KV (*PowerSetup* attribute), 40
- AC_50KV (*PowerSetup* attribute), 40
- AC_DoubleStage_150kV (*PowerSetup* attribute), 60
- AC_DoubleStage_200kV (*PowerSetup* attribute), 60
- AC_SingleStage_100kV (*PowerSetup* attribute), 60
- AC_SingleStage_50kV (*PowerSetup* attribute), 60
- ACA (*TiePieOscilloscopeChannelCoupling* attribute), 74
- acceleration (*NewportSMC100PPConfig* attribute), 141
- ACCESS_ENABLE (*ILS2TRegAddr* attribute), 165
- access_lock (*CommunicationProtocol* attribute), 14
- ACTION_JOG_VALUE (*ILS2T* attribute), 160
- activate_clock_output() (*T560* method), 47
- activate_measurements() (*RTO1024* method), 153
- activate_output() (*Luminex* method), 166
- activated (*BreakdownDetection* attribute), 56
- ACTIVE (*CryLasLaser.AnswersStatus* attribute), 88
- active_alarms() (*BaseCube* method), 33
- ACTIVE_HIGH (*Polarity* attribute), 44
- ACTIVE_LOW (*Polarity* attribute), 45
- actualsetvalue (*FuGProbusVSetRegisters* property), 107
- ACV (*TiePieOscilloscopeChannelCoupling* attribute), 74
- adc_mode (*FuGProbusVMonitorRegisters* property), 107
- add_device() (*DeviceSequenceMixin* method), 83
- add_device() (*ExperimentManager* method), 180
- ADDR_INCORRECT (*NewportSMC100PPSerialCommunication.ControllerErrors* attribute), 142
- address (*NewportSMC100PPConfig* attribute), 141
- address (*VisaCommunicationConfig* property), 31
- address() (*VisaCommunicationConfig.InterfaceType* method), 31
- ADMODE (*FuGProbusIVCommands* attribute), 104
- Alarm0 (*AlarmText* attribute), 55
- Alarm1 (*AlarmText* attribute), 55
- Alarm10 (*AlarmText* attribute), 55
- Alarm11 (*AlarmText* attribute), 55
- Alarm12 (*AlarmText* attribute), 55
- Alarm13 (*AlarmText* attribute), 55
- Alarm14 (*AlarmText* attribute), 55
- Alarm17 (*AlarmText* attribute), 55
- Alarm19 (*AlarmText* attribute), 55
- Alarm2 (*AlarmText* attribute), 55
- Alarm20 (*AlarmText* attribute), 55
- Alarm21 (*AlarmText* attribute), 55
- Alarm22 (*AlarmText* attribute), 55
- Alarm3 (*AlarmText* attribute), 55
- Alarm4 (*AlarmText* attribute), 55
- Alarm5 (*AlarmText* attribute), 55
- Alarm6 (*AlarmText* attribute), 55
- Alarm7 (*AlarmText* attribute), 55
- Alarm8 (*AlarmText* attribute), 55
- Alarm9 (*AlarmText* attribute), 55
- AlarmText (*class* in *hvl_ccb.dev.supercube2015.constants*), 55
- ALL (*TiePieOscilloscopeAutoResolutionModes* attribute), 80
- all_measurements (*LuminexMeasurementType* attribute), 168
- all_measurements_types() (*LuminexMeasurementType* class method), 168
- amplitude (*TiePieGeneratorConfig* property), 76
- ANALOG (*LaudaProRp245eConfig.ExtControlModeEnum* attribute), 126
- analog_control (*FuGProbusVDIRegisters* property), 106
- ANY (*LabJack.DeviceType* attribute), 118
- ANY (*LJMCommunicationConfig.ConnectionType* attribute), 16
- ANY (*LJMCommunicationConfig.DeviceType* attribute), 16
- ANY (*TiePieOscilloscopeTriggerKind* attribute), 75
- ARBITRARY (*TiePieGeneratorSignalType* attribute), 77
- AsyncCommunicationProtocol (*class* in

hvl_ccb.comm.base), 11
 AsyncCommunicationProtocolConfig (class in *hvl_ccb.comm.base*), 13
 ATM (*Pressure* attribute), 175
 ATMOSPHERE (*Pressure* attribute), 175
 attenuation (*CryLasAttenuator* property), 86
 AUTO (*EarthingStickOperatingStatus* attribute), 39
 AUTO (*LaudaProRp245eConfig.OperationModeEnum* attribute), 127
 AUTO (*RTO1024.TriggerModes* attribute), 153
 auto_install_mode (*T560* property), 47
 auto_install_mode (*T560Config* attribute), 48
 auto_laser_on (*CryLasLaserConfig* attribute), 91
 auto_resolution_mode (*TiePieOscilloscopeConfig* property), 80
 AutoInstallMode (class in *hvl_ccb.dev.highland_t560.base*), 44
 AutoNumberNameEnum (class in *hvl_ccb.utils.enum*), 176

B

B (*SupercubeOpcEndpoint* attribute), 62
 BA (*NewportConfigCommands* attribute), 133
 backlash_compensation (*NewportSMC100PPConfig* attribute), 141
 backup_waveform() (*RTO1024* method), 153
 bar (*PfeifferTPG.PressureUnits* attribute), 147
 BAR (*Pressure* attribute), 175
 barometric_pressure (*LuminoxMeasurementType* attribute), 168
 base_velocity (*NewportSMC100PPConfig* attribute), 141
 BaseCube (class in *hvl_ccb.dev.cube.base*), 33
 BaseCubeConfiguration (class in *hvl_ccb.dev.cube.base*), 36
 BaseCubeOpcUaCommunication (class in *hvl_ccb.dev.cube.base*), 37
 BaseCubeOpcUaCommunicationConfig (class in *hvl_ccb.dev.cube.base*), 37
 BATH_TEMP (*LaudaProRp245eCommand* attribute), 125
 baudrate (*CryLasAttenuatorSerialCommunicationConfig* attribute), 87
 baudrate (*CryLasLaserSerialCommunicationConfig* attribute), 93
 baudrate (*FuGSerialCommunicationConfig* attribute), 109
 baudrate (*HeinzingerSerialCommunicationConfig* attribute), 115
 baudrate (*LuminoxSerialCommunicationConfig* attribute), 169
 baudrate (*MBW973SerialCommunicationConfig* attribute), 132
 baudrate (*NewportSMC100PPSerialCommunicationConfig* attribute), 144

baudrate (*PfeifferTPGSerialCommunicationConfig* attribute), 151
 baudrate (*SerialCommunicationConfig* attribute), 24
 BH (*NewportConfigCommands* attribute), 133
 board (*VisaCommunicationConfig* attribute), 31
 BoolEnum (class in *hvl_ccb.utils.enum*), 176
 breakdown_detection_active (*BaseCube* property), 33
 breakdown_detection_reset() (*BaseCube* method), 34
 breakdown_detection_triggered (*BaseCube* property), 34
 BreakdownDetection (class in *hvl_ccb.dev.supercube2015.constants*), 56
 bufsize (*TcpCommunicationConfig* attribute), 27
 build_str() (*LaudaProRp245eCommand* method), 126
 bytesize (*CryLasAttenuatorSerialCommunicationConfig* attribute), 87
 bytesize (*CryLasLaserSerialCommunicationConfig* attribute), 94
 bytesize (*FuGSerialCommunicationConfig* attribute), 109
 bytesize (*HeinzingerSerialCommunicationConfig* attribute), 115
 bytesize (*LuminoxSerialCommunicationConfig* attribute), 169
 bytesize (*MBW973SerialCommunicationConfig* attribute), 132
 bytesize (*NewportSMC100PPSerialCommunicationConfig* attribute), 145
 bytesize (*PfeifferTPGSerialCommunicationConfig* attribute), 151
 Bytesize (*SerialCommunicationConfig* attribute), 24
 bytesize (*SerialCommunicationConfig* attribute), 24

C

C (*LabJack.TemperatureUnit* attribute), 118
 C (*LabJack.ThermocoupleType* attribute), 119
 C (*Temperature* attribute), 175
 calibration_factor (*CryLasLaserConfig* attribute), 91
 calibration_factor (*LEM4000S* attribute), 174
 calibration_mode (*FuGProbusVDIRegisters* property), 106
 cc_mode (*FuGProbusVDIRegisters* property), 106
 CCBException, 180
 cee16 (*GeneralSockets* attribute), 58
 cee16_socket (*BaseCube* property), 34
 CELSIUS (*Temperature* attribute), 175
 ch_a (*T560* property), 47
 ch_b (*T560* property), 47
 ch_c (*T560* property), 47
 ch_d (*T560* property), 47
 channels_enabled (*TiePieOscilloscope* property), 79

`check_for_error()` (*NewportSMC100PPSerialCommunication method*), 143
`check_generic_type()` (in module *hvl_ccb.utils.typing*), 177
`check_master_slave_config()` (*PSI9000 method*), 95
`chunk_size` (*VisaCommunicationConfig attribute*), 31
`clean_amplitude()` (*TiePieGeneratorConfig method*), 76
`clean_auto_resolution_mode()` (*TiePieOscilloscopeConfig static method*), 80
`clean_coupling()` (*TiePieOscilloscopeChannelConfig static method*), 73
`clean_enabled()` (*TiePieGeneratorConfig static method*), 76
`clean_enabled()` (*TiePieOscilloscopeChannelConfig static method*), 73
`clean_frequency()` (*TiePieGeneratorConfig method*), 76
`clean_input_range()` (*TiePieOscilloscopeChannelConfig method*), 73
`clean_offset()` (*TiePieGeneratorConfig method*), 76
`clean_pre_sample_ratio()` (*TiePieOscilloscopeConfig method*), 81
`clean_probe_offset()` (*TiePieOscilloscopeChannelConfig method*), 73
`clean_record_length()` (*TiePieOscilloscopeConfig method*), 81
`clean_resolution()` (*TiePieOscilloscopeConfig static method*), 81
`clean_sample_frequency()` (*TiePieOscilloscopeConfig method*), 81
`clean_signal_type()` (*TiePieGeneratorConfig static method*), 76
`clean_trigger_enabled()` (*TiePieOscilloscopeChannelConfig static method*), 73
`clean_trigger_hysteresis()` (*TiePieOscilloscopeChannelConfig method*), 73
`clean_trigger_kind()` (*TiePieOscilloscopeChannelConfig static method*), 73
`clean_trigger_level()` (*TiePieOscilloscopeChannelConfig method*), 73
`clean_trigger_level_mode()` (*TiePieOscilloscopeChannelConfig static method*), 73
`clean_trigger_timeout()` (*TiePieOscilloscopeConfig method*), 81
`clean_values()` (*AsyncCommunicationProtocolConfig method*), 13
`clean_values()` (*BaseCubeConfiguration method*), 36
`clean_values()` (*CryLasAttenuatorConfig method*), 86
`clean_values()` (*CryLasLaserConfig method*), 91
`clean_values()` (*EmptyConfig method*), 84, 179
`clean_values()` (*FuGConfig method*), 101
`clean_values()` (*HeinzingerConfig method*), 111
`clean_values()` (*ILS2TConfig method*), 163
`clean_values()` (*LaudaProRp245eConfig method*), 127
`clean_values()` (*LaudaProRp245eTcpCommunicationConfig method*), 128
`clean_values()` (*LJMCommunicationConfig method*), 17
`clean_values()` (*LuminoxConfig method*), 167
`clean_values()` (*MBW973Config method*), 130
`clean_values()` (*ModbusTcpCommunicationConfig method*), 19
`clean_values()` (*NewportSMC100PPConfig method*), 141
`clean_values()` (*OpcUaCommunicationConfig method*), 21
`clean_values()` (*PfeifferTPGConfig method*), 149
`clean_values()` (*PICubeConfiguration method*), 42
`clean_values()` (*PSI9000Config method*), 97
`clean_values()` (*SerialCommunicationConfig method*), 24
`clean_values()` (*SupercubeConfiguration method*), 53
`clean_values()` (*T560Config method*), 48
`clean_values()` (*TcpCommunicationConfig method*), 27
`clean_values()` (*TechnixConfig method*), 69
`clean_values()` (*TelnetCommunicationConfig method*), 29
`clean_values()` (*TiePieDeviceConfig method*), 71
`clean_values()` (*VisaCommunicationConfig method*), 31
`clean_waveform()` (*TiePieGeneratorConfig method*), 76
`Client` (*class in hvl_ccb.comm.opc*), 20
`CLOSE` (*EarthingStickOperation attribute*), 39
`close()` (*CommunicationProtocol method*), 14
`close()` (*LaudaProRp245eTcpCommunication method*), 128
`close()` (*LJMCommunication method*), 15
`close()` (*ModbusTcpCommunication method*), 18
`close()` (*NullCommunicationProtocol method*), 14
`close()` (*OpcUaCommunication method*), 20
`close()` (*SerialCommunication method*), 23
`close()` (*Tcp method*), 26
`close()` (*TelnetCommunication method*), 28
`close()` (*VisaCommunication method*), 30
`close_shutter()` (*CryLasLaser method*), 89
`CLOSED` (*CryLasLaser.AnswersShutter attribute*), 88
`CLOSED` (*CryLasLaserShutterStatus attribute*), 94
`CLOSED` (*DoorStatus attribute*), 38
`closed` (*DoorStatus attribute*), 56
`CLOSED` (*EarthingStickStatus attribute*), 39
`closed` (*EarthingStickStatus attribute*), 57

CMD_EXEC_ERROR (New- `config_cls()` (`AsyncCommunicationProtocol` static method), 11
portSMC100PPSerialCommunication.ControllerErrors attribute), 142 `config_cls()` (`BaseCube` static method), 34
CMD_NOT_ALLOWED (New- `config_cls()` (`BaseCubeOpcUaCommunication` static method), 37
portSMC100PPSerialCommunication.ControllerErrors attribute), 142 `config_cls()` (`ConfigurationMixin` static method), 178
CMD_NOT_ALLOWED_CC (New- `config_cls()` (`CryLasAttenuator` static method), 86
portSMC100PPSerialCommunication.ControllerErrors attribute), 142 `config_cls()` (`CryLasAttenuatorSerialCommunication` static method), 87
CMD_NOT_ALLOWED_CONFIGURATION (New- `config_cls()` (`CryLasLaser` static method), 89
portSMC100PPSerialCommunication.ControllerErrors attribute), 142 `config_cls()` (`CryLasLaserSerialCommunication` static method), 93
CMD_NOT_ALLOWED_DISABLE (New- `config_cls()` (`Device` static method), 83
portSMC100PPSerialCommunication.ControllerErrors attribute), 142 `config_cls()` (`FuGProbusIV` static method), 104
CMD_NOT_ALLOWED_HOMING (New- `config_cls()` (`FuGSerialCommunication` static method), 109
portSMC100PPSerialCommunication.ControllerErrors attribute), 142 `config_cls()` (`HeinzingerDI` static method), 112
CMD_NOT_ALLOWED_MOVING (New- `config_cls()` (`HeinzingerSerialCommunication` static method), 115
portSMC100PPSerialCommunication.ControllerErrors attribute), 142 `config_cls()` (`ILS2T` static method), 161
CMD_NOT_ALLOWED_NOT_REFERENCED (New- `config_cls()` (`ILS2TModbusTcpCommunication` static method), 164
portSMC100PPSerialCommunication.ControllerErrors attribute), 142 `config_cls()` (`LaudaProRp245e` static method), 123
CMD_NOT_ALLOWED_PP (New- `config_cls()` (`LaudaProRp245eTcpCommunication` static method), 128
portSMC100PPSerialCommunication.ControllerErrors attribute), 142 `config_cls()` (`LJMCommunication` static method), 15
CMD_NOT_ALLOWED_READY (New- `config_cls()` (`Luminox` static method), 166
portSMC100PPSerialCommunication.ControllerErrors attribute), 142 `config_cls()` (`LuminoxSerialCommunication` static method), 169
CMR (`PfeifferTPG.SensorTypes` attribute), 147 `config_cls()` (`MBW973` static method), 129
CODE_OR_ADDR_INVALID (New- `config_cls()` (`MBW973SerialCommunication` static method), 131
portSMC100PPSerialCommunication.ControllerErrors attribute), 142 `config_cls()` (`ModbusTcpCommunication` static method), 18
collect_measurement_data() (`TiePieOscilloscope` `config_cls()` (`NewportSMC100PP` static method), 135
method), 79 `config_cls()` (`NewportSMC100PPSerialCommunication` static method), 143
com (`SingleCommDevice` property), 85
COM_TIME_OUT (`LaudaProRp245eCommand` attribute), 125 `config_cls()` (`NullCommunicationProtocol` static method), 14
com_time_out (`LaudaProRp245eConfig` attribute), 127 `config_cls()` (`OpcUaCommunication` static method),
COM_TIMEOUT (`NewportSMC100PPSerialCommunication.ControllerErrors` attribute), 142 `config_cls()` (`PfeifferTPG` static method), 147
command (`LuminoxMeasurementType` property), 168 `config_cls()` (`PfeifferTPGSerialCommunication` static method), 150
COMMAND (`TriggerMode` attribute), 46 `config_cls()` (`PICube` static method), 41
command() (`FuGProbusIV` method), 104 `config_cls()` (`PICubeOpcUaCommunication` static method), 43
communication_channel (`TechnixConfig` attribute), 69
CommunicationException, 14
CommunicationProtocol (class in `config_cls()` (`PSI9000` static method), 95
hvl_ccb.comm.base), 14 `config_cls()` (`PSI9000VisaCommunication` static method), 98
config (`ConfigurationMixin` property), 178 `config_cls()` (`RTO1024` static method), 153
CONFIG (`FuGProbusVRegisterGroups` attribute), 107 `config_cls()` (`RTO1024VisaCommunication` static method), 159
CONFIG (`NewportSMC100PP.StateMessages` attribute), 134
CONFIG (`NewportStates` attribute), 145 `config_cls()` (`SerialCommunication` static method), 23

- `config_cls()` (*Supercube2015Base* static method), 49
`config_cls()` (*SupercubeAOpcUaCommunication* static method), 63
`config_cls()` (*SupercubeOpcUaCommunication* static method), 53
`config_cls()` (*SyncCommunicationProtocol* static method), 14
`config_cls()` (*T560* static method), 47
`config_cls()` (*T560Communication* static method), 45
`config_cls()` (*Tcp* static method), 26
`config_cls()` (*Technix* static method), 68
`config_cls()` (*TechnixSerialCommunication* static method), 65
`config_cls()` (*TechnixTelnetCommunication* static method), 66
`config_cls()` (*TelnetCommunication* static method), 28
`config_cls()` (*TiePieOscilloscope* static method), 79
`config_cls()` (*VisaCommunication* static method), 30
`config_cls()` (*VisaDevice* static method), 172
`config_gen` (*TiePieGeneratorMixin* attribute), 77
`config_high_pulse()` (*LabJack* method), 119
`config_i2c` (*TiePieI2CHostMixin* attribute), 78
`config_osc` (*TiePieOscilloscope* attribute), 79
`config_osc_channel_dict` (*TiePieOscilloscope* attribute), 79
`config_status` (*FuG* property), 99
`configdataclass()` (in module *hvl_ccb.configuration*), 179
`configuration_save_json()` (*ConfigurationMixin* method), 179
`ConfigurationMixin` (class in *hvl_ccb.configuration*), 178
`ConfigurationValueWarning`, 179
`connection_type` (*LJMCommunicationConfig* attribute), 17
`CONT_MODE` (*LaudaProRp245eCommand* attribute), 125
`continue_ramp()` (*LaudaProRp245e* method), 123
`control_mode` (*LaudaProRp245eConfig* attribute), 127
`CONVERSION` (*LEM4000S* attribute), 173
`convert()` (*LEM4000S* method), 174
`convert()` (*LMT70A* method), 174
`convert()` (*Sensor* method), 174
`convert()` (*Unit* class method), 175
`ConvertibleTypes` (in module *hvl_ccb.utils.typing*), 177
`COOLOFF` (*LaudaProRp245eConfig.OperationModeEnum* attribute), 127
`COOLON` (*LaudaProRp245eConfig.OperationModeEnum* attribute), 127
`coupling` (*TiePieOscilloscopeChannelConfig* property), 73
`CR` (*FuGTerminators* attribute), 110
`create_serial_port()` (*SerialCommunicationConfig* method), 24
`create_telnet()` (*TelnetCommunicationConfig* method), 29
`CRLF` (*FuGTerminators* attribute), 110
`CryLasAttenuator` (class in *hvl_ccb.dev.crylas*), 86
`CryLasAttenuatorConfig` (class in *hvl_ccb.dev.crylas*), 86
`CryLasAttenuatorError`, 87
`CryLasAttenuatorSerialCommunication` (class in *hvl_ccb.dev.crylas*), 87
`CryLasAttenuatorSerialCommunicationConfig` (class in *hvl_ccb.dev.crylas*), 87
`CryLasLaser` (class in *hvl_ccb.dev.crylas*), 88
`CryLasLaser.AnswersShutter` (class in *hvl_ccb.dev.crylas*), 88
`CryLasLaser.AnswersStatus` (class in *hvl_ccb.dev.crylas*), 88
`CryLasLaser.LaserStatus` (class in *hvl_ccb.dev.crylas*), 89
`CryLasLaser.RepetitionRates` (class in *hvl_ccb.dev.crylas*), 89
`CryLasLaserConfig` (class in *hvl_ccb.dev.crylas*), 91
`CryLasLaserError`, 92
`CryLasLaserNotReadyError`, 92
`CryLasLaserPoller` (class in *hvl_ccb.dev.crylas*), 92
`CryLasLaserSerialCommunication` (class in *hvl_ccb.dev.crylas*), 92
`CryLasLaserSerialCommunicationConfig` (class in *hvl_ccb.dev.crylas*), 93
`CryLasLaserShutterStatus` (class in *hvl_ccb.dev.crylas*), 94
`CubeEarthingStickOperationError`, 38
`CubeError`, 38
`CubeRemoteControlError`, 38
`CubeStatusChangeError`, 38
`CubeStopError`, 38
`current` (*FuG* property), 99
`CURRENT` (*FuGProbusIVCommands* attribute), 104
`CURRENT` (*FuGReadbackChannels* attribute), 108
`current` (*Technix* property), 68
`current_lower_limit` (*PSI9000Config* attribute), 97
`current_monitor` (*FuG* property), 100
`current_primary` (*PICube* property), 41
`current_primary` (*Power* attribute), 60
`current_upper_limit` (*PSI9000Config* attribute), 97
`CurrentPosition` (*New-portSMC100PPConfig.HomeSearch* attribute), 140
`cv_mode` (*FuGProbusVDIRegisters* property), 106
- ## D
- `datachange_notification()` (*OpcUaSubHandler* method), 22
`datachange_notification()` (*SupercubeSubscriptionHandler* method), 54

date_of_manufacture (*LuminosMeasurementType* attribute), 168
 datetime_to_opc() (*BaseCube* class method), 34
 DC (*TiePieGeneratorSignalType* attribute), 77
 DC_140KV (*PowerSetup* attribute), 40
 DC_280KV (*PowerSetup* attribute), 40
 DC_DoubleStage_280kV (*PowerSetup* attribute), 61
 DC_SingleStage_140kV (*PowerSetup* attribute), 61
 DC_VOLTAGE_TOO_LOW (*NewportSMC100PP.MotorErrors* attribute), 134
 DCA (*TiePieOscilloscopeChannelCoupling* attribute), 74
 DCV (*TiePieOscilloscopeChannelCoupling* attribute), 74
 default_com_cls() (*BaseCube* static method), 34
 default_com_cls() (*CryLasAttenuator* static method), 86
 default_com_cls() (*CryLasLaser* static method), 89
 default_com_cls() (*FuGProbusIV* static method), 104
 default_com_cls() (*HeinzingerDI* static method), 112
 default_com_cls() (*ILS2T* static method), 161
 default_com_cls() (*LabJack* static method), 119
 default_com_cls() (*LaudaProRp245e* static method), 123
 default_com_cls() (*Luminos* static method), 167
 default_com_cls() (*MBW973* static method), 129
 default_com_cls() (*NewportSMC100PP* static method), 135
 default_com_cls() (*PfeifferTPG* static method), 147
 default_com_cls() (*PICube* static method), 41
 default_com_cls() (*PSI9000* static method), 95
 default_com_cls() (*RTO1024* static method), 153
 default_com_cls() (*SingleCommDevice* static method), 85
 default_com_cls() (*Supercube2015Base* static method), 50
 default_com_cls() (*Supercube2015WithFU* static method), 62
 default_com_cls() (*T560* static method), 47
 default_com_cls() (*Technix* method), 68
 default_com_cls() (*TiePieOscilloscope* static method), 79
 default_com_cls() (*VisaDevice* static method), 172
 DEFAULT_IO_SCANNING_CONTROL_VALUES (*ILS2T* attribute), 160
 default_n_attempts_read_text_nonempty (*AsyncCommunicationProtocolConfig* attribute), 13
 default_n_attempts_read_text_nonempty (*FuGSerialCommunicationConfig* attribute), 109
 default_n_attempts_read_text_nonempty (*HeinzingerSerialCommunicationConfig* attribute), 115
 default_number_of_recordings (*HeinzingerConfig* attribute), 111
 Device (class in *hvl_ccb.dev.base*), 83
 DEVICE_TYPE (*LaudaProRp245eCommand* attribute), 125
 device_type (*LJMCommunicationConfig* attribute), 17
 DeviceException, 83
 DeviceExistingException, 83
 DeviceFailuresException, 83
 devices_failed_start (*DeviceSequenceMixin* attribute), 84
 devices_failed_start (*ExperimentManager* attribute), 181
 devices_failed_stop (*DeviceSequenceMixin* attribute), 84
 devices_failed_stop (*ExperimentManager* attribute), 181
 DeviceSequenceMixin (class in *hvl_ccb.dev.base*), 83
 di (*FuG* property), 100
 digital_control (*FuGProbusVDIRegisters* property), 106
 DIOChannel (*LabJack* attribute), 118
 DISABLE (*NewportStates* attribute), 145
 disable() (*ILS2T* method), 161
 DISABLE_FROM_JOGGING (*NewportSMC100PP.StateMessages* attribute), 134
 DISABLE_FROM_MOVING (*NewportSMC100PP.StateMessages* attribute), 134
 DISABLE_FROM_READY (*NewportSMC100PP.StateMessages* attribute), 134
 disable_pulses() (*LabJack* method), 119
 DISABLED (*TiePieOscilloscopeAutoResolutionModes* attribute), 80
 DisableEspStageCheck (*NewportSMC100PPConfig.EspStageConfig* attribute), 140
 disarm_trigger() (*T560* method), 48
 disconnect() (*Client* method), 20
 DISPLACEMENT_OUT_OF_LIMIT (*NewportSMC100PPSerialCommunication.ControllerErrors* attribute), 142
 display_message_board() (*BaseCube* method), 34
 display_status_board() (*BaseCube* method), 34
 do_ioscanning_write() (*ILS2T* method), 161
 door_1_status (*BaseCube* attribute), 34
 door_2_status (*BaseCube* attribute), 34
 door_3_status (*BaseCube* attribute), 34
 DoorStatus (class in *hvl_ccb.dev.cube.constants*), 38
 DoorStatus (class in *hvl_ccb.dev.supercube2015.constants*), 56

E

E (*LabJack.ThermocoupleType* attribute), 119
 E0 (*FuGErrorcodes* attribute), 102

- E1 (*FuErrorcodes* attribute), 102
- E10 (*FuErrorcodes* attribute), 102
- E100 (*FuErrorcodes* attribute), 102
- E106 (*FuErrorcodes* attribute), 102
- E11 (*FuErrorcodes* attribute), 102
- E115 (*FuErrorcodes* attribute), 102
- E12 (*FuErrorcodes* attribute), 102
- E125 (*FuErrorcodes* attribute), 102
- E13 (*FuErrorcodes* attribute), 102
- E135 (*FuErrorcodes* attribute), 102
- E14 (*FuErrorcodes* attribute), 102
- E145 (*FuErrorcodes* attribute), 102
- E15 (*FuErrorcodes* attribute), 102
- E155 (*FuErrorcodes* attribute), 102
- E16 (*FuErrorcodes* attribute), 103
- E165 (*FuErrorcodes* attribute), 103
- E2 (*FuErrorcodes* attribute), 103
- E206 (*FuErrorcodes* attribute), 103
- E306 (*FuErrorcodes* attribute), 103
- E4 (*FuErrorcodes* attribute), 103
- E5 (*FuErrorcodes* attribute), 103
- E504 (*FuErrorcodes* attribute), 103
- E505 (*FuErrorcodes* attribute), 103
- E6 (*FuErrorcodes* attribute), 103
- E666 (*FuErrorcodes* attribute), 103
- E7 (*FuErrorcodes* attribute), 103
- E8 (*FuErrorcodes* attribute), 103
- E9 (*FuErrorcodes* attribute), 103
- earthing_rod_1_status (*BaseCube* attribute), 34
- earthing_rod_2_status (*BaseCube* attribute), 34
- earthing_rod_3_status (*BaseCube* attribute), 34
- EarthingRodStatus (class *hvl_ccb.dev.cube.constants*), 39
- EarthingStick (class *hvl_ccb.dev.supercube2015.constants*), 56
- EarthingStickOperatingStatus (class *hvl_ccb.dev.cube.constants*), 39
- EarthingStickOperation (class *hvl_ccb.dev.cube.constants*), 39
- EarthingStickStatus (class *hvl_ccb.dev.cube.constants*), 39
- EarthingStickStatus (class *hvl_ccb.dev.supercube2015.constants*), 57
- EEPROM_ACCESS_ERROR (NewportSMC100PPSerialCommunication.ControllerErrors attribute), 142
- EIGHT (*HeinzingerConfig.RecordingsEnum* attribute), 111
- EIGHT_BIT (*TiePieOscilloscopeResolution* attribute), 81
- EIGHT_HUNDRED_MILLI_VOLT (*TiePieOscilloscopeRange* attribute), 74
- EIGHT_VOLT (*TiePieOscilloscopeRange* attribute), 74
- EIGHTBITS (*SerialCommunicationBytesize* attribute), 24
- EIGHTY_VOLT (*TiePieOscilloscopeRange* attribute), 74
- EmptyConfig (class in *hvl_ccb.configuration*), 179
- EmptyConfig (class in *hvl_ccb.dev.base*), 84
- enable() (*ILS2T* method), 161
- enable_clock() (*LabJack* method), 119
- enabled (*TiePieGeneratorConfig* property), 76
- enabled (*TiePieOscilloscopeChannelConfig* property), 73
- EnableEspStageCheck (NewportSMC100PPConfig.EspStageConfig attribute), 140
- encoding (*AsyncCommunicationProtocolConfig* attribute), 13
- encoding (*NewportSMC100PPSerialCommunicationConfig* attribute), 145
- encoding_error_handling (*AsyncCommunicationProtocolConfig* attribute), 13
- encoding_error_handling (NewportSMC100PPSerialCommunicationConfig attribute), 145
- EndOfRunSwitch (NewportSMC100PPConfig.HomeSearch attribute), 140
- EndOfRunSwitch_and_Index (NewportSMC100PPConfig.HomeSearch attribute), 140
- endpoint_name (*BaseCubeOpcUaCommunicationConfig* attribute), 37
- endpoint_name (*OpcUaCommunicationConfig* attribute), 21
- endpoint_name (*PICubeOpcUaCommunicationConfig* attribute), 43
- in endpoint_name (*SupercubeAOpcUaConfiguration* attribute), 64
- in ERROR (*DoorStatus* attribute), 38
- error (*DoorStatus* attribute), 56
- in ERROR (*EarthingStickStatus* attribute), 39
- error (*EarthingStickStatus* attribute), 58
- in ERROR (*ExperimentStatus* attribute), 181
- ERROR (*SafetyStatus* attribute), 40
- in Error (*SafetyStatus* attribute), 61
- errorcode (*FuError* attribute), 102
- in Errors (class in *hvl_ccb.dev.supercube2015.constants*), 58
- ESP_STAGE_NAME_INVALID (NewportSMC100PPSerialCommunication.ControllerErrors attribute), 142
- ETH (*LaudaProRp245eConfig.ExtControlModeEnum* attribute), 126
- ETHERNET (*LJMCommunicationConfig.ConnectionType* attribute), 16
- EVEN (*SerialCommunicationParity* attribute), 25
- event_notification() (*OpcUaSubHandler* method), 22
- EXECUTE (*FuGProbusIVCommands* attribute), 104

`execute_absolute_position()` (*ILS2T method*), 161
`execute_on_x` (*FuGProbusVConfigRegisters property*), 105
`execute_relative_step()` (*ILS2T method*), 161
`EXECUTEONX` (*FuGProbusIVCommands attribute*), 104
`exit_configuration()` (*NewportSMC100PP method*), 135
`exit_configuration_wait_sec` (*NewportSMC100PPConfig attribute*), 141
`EXPERIMENT_BLOCKED` (*EarthingRodStatus attribute*), 39
`EXPERIMENT_READY` (*EarthingRodStatus attribute*), 39
`ExperimentError`, 180
`ExperimentManager` (class in *hvl_ccb.experiment_manager*), 180
`ExperimentStatus` (class in *hvl_ccb.experiment_manager*), 181
`EXPT100` (*LaudaProRp245eConfig.ExtControlModeEnum attribute*), 126
`EXT_FALLING_EDGE` (*TriggerMode attribute*), 46
`EXT_RISING_EDGE` (*TriggerMode attribute*), 46
`External` (*PowerSetup attribute*), 61
`EXTERNAL_SOURCE` (*PowerSetup attribute*), 40
`EXTERNAL_TEMP` (*LaudaProRp245eCommand attribute*), 125

F

`F` (*LabJack.TemperatureUnit attribute*), 118
`F` (*Temperature attribute*), 175
`FAHRENHEIT` (*Temperature attribute*), 175
`failures` (*DeviceFailuresException attribute*), 83
`FALLING` (*TiePieOscilloscopeTriggerKind attribute*), 75
`FAST` (*ILS2T.Ref16Jog attribute*), 160
`file_copy()` (*RTO1024 method*), 153
`finish()` (*ExperimentManager method*), 181
`FINISHED` (*ExperimentStatus attribute*), 181
`FINISHING` (*ExperimentStatus attribute*), 181
`fire_trigger()` (*T560 method*), 48
`FIRMWARE` (*FuGReadbackChannels attribute*), 108
`FIVE_MHZ` (*LabJack.ClockFrequency attribute*), 117
`FIVEBITS` (*SerialCommunicationBytesize attribute*), 24
`FLT_INFO` (*ILS2TRegAddr attribute*), 165
`FLT_MEM_DEL` (*ILS2TRegAddr attribute*), 165
`FLT_MEM_RESET` (*ILS2TRegAddr attribute*), 165
`FOLLOWING_ERROR` (*NewportSMC100PP.MotorErrors attribute*), 134
`FOLLOWRAMP` (*FuGRampModes attribute*), 108
`force_trigger()` (*TiePieOscilloscope method*), 80
`force_value()` (*AsyncCommunicationProtocolConfig method*), 13
`force_value()` (*BaseCubeConfiguration method*), 36
`force_value()` (*BaseCubeOpcUaCommunicationConfig method*), 37
`force_value()` (*CryLasAttenuatorConfig method*), 86
`force_value()` (*CryLasAttenuatorSerialCommunicationConfig method*), 87
`force_value()` (*CryLasLaserConfig method*), 91
`force_value()` (*CryLasLaserSerialCommunicationConfig method*), 94
`force_value()` (*EmptyConfig method*), 84, 179
`force_value()` (*FuGConfig method*), 101
`force_value()` (*FuGSerialCommunicationConfig method*), 109
`force_value()` (*HeinzingerConfig method*), 111
`force_value()` (*HeinzingerSerialCommunicationConfig method*), 115
`force_value()` (*ILS2TConfig method*), 163
`force_value()` (*ILS2TModbusTcpCommunicationConfig method*), 164
`force_value()` (*LaudaProRp245eConfig method*), 127
`force_value()` (*LaudaProRp245eTcpCommunicationConfig method*), 128
`force_value()` (*LJMCommunicationConfig method*), 17
`force_value()` (*LuminoxConfig method*), 167
`force_value()` (*LuminoxSerialCommunicationConfig method*), 170
`force_value()` (*MBW973Config method*), 131
`force_value()` (*MBW973SerialCommunicationConfig method*), 132
`force_value()` (*ModbusTcpCommunicationConfig method*), 19
`force_value()` (*NewportSMC100PPConfig method*), 141
`force_value()` (*NewportSMC100PPSerialCommunicationConfig method*), 145
`force_value()` (*OpcUaCommunicationConfig method*), 21
`force_value()` (*PfeifferTPGConfig method*), 149
`force_value()` (*PfeifferTPGSerialCommunicationConfig method*), 151
`force_value()` (*PICubeConfiguration method*), 42
`force_value()` (*PICubeOpcUaCommunicationConfig method*), 43
`force_value()` (*PSI9000Config method*), 97
`force_value()` (*PSI9000VisaCommunicationConfig method*), 98
`force_value()` (*RTO1024Config method*), 158
`force_value()` (*RTO1024VisaCommunicationConfig method*), 159
`force_value()` (*SerialCommunicationConfig method*), 24
`force_value()` (*SupercubeAOpcUaConfiguration method*), 64
`force_value()` (*SupercubeConfiguration method*), 53
`force_value()` (*SupercubeOpcUaCommunicationConfig method*), 54

- force_value() (*T560CommunicationConfig* method), 45
 force_value() (*T560Config* method), 48
 force_value() (*TcpCommunicationConfig* method), 27
 force_value() (*TechnixConfig* method), 69
 force_value() (*TechnixSerialCommunicationConfig* method), 66
 force_value() (*TechnixTelnetCommunicationConfig* method), 67
 force_value() (*TelnetCommunicationConfig* method), 29
 force_value() (*TiePieDeviceConfig* method), 71
 force_value() (*VisaCommunicationConfig* method), 31
 force_value() (*VisaDeviceConfig* method), 172
 FORTY_MHZ (*LabJack.ClockFrequency* attribute), 118
 FORTY_VOLT (*TiePieOscilloscopeRange* attribute), 74
 FOUR (*HeinzingerConfig.RecordingsEnum* attribute), 111
 FOUR_HUNDRED_MILLI_VOLT (*TiePieOscilloscopeRange* attribute), 74
 FOUR_VOLT (*TiePieOscilloscopeRange* attribute), 74
 FOURTEEN_BIT (*TiePieOscilloscopeResolution* attribute), 81
 FREERUN (*RTO1024.TriggerModes* attribute), 153
 frequency (*PICube* property), 41
 frequency (*Power* attribute), 60
 frequency (*T560* property), 48
 frequency (*TiePieGeneratorConfig* property), 76
 FRM (*NewportConfigCommands* attribute), 133
 from_json() (*ConfigurationMixin* class method), 179
 FRS (*NewportConfigCommands* attribute), 133
 fso_reset() (*Supercube2015WithFU* method), 63
 FuG (class in *hvl_ccb.dev.fug*), 99
 FuGConfig (class in *hvl_ccb.dev.fug*), 101
 FuGDigitalVal (class in *hvl_ccb.dev.fug*), 101
 FuGError, 102
 FuGErrorcodes (class in *hvl_ccb.dev.fug*), 102
 FuGMonitorModes (class in *hvl_ccb.dev.fug*), 103
 FuGPolarities (class in *hvl_ccb.dev.fug*), 104
 FuGProbusIV (class in *hvl_ccb.dev.fug*), 104
 FuGProbusIVCommands (class in *hvl_ccb.dev.fug*), 104
 FuGProbusV (class in *hvl_ccb.dev.fug*), 105
 FuGProbusVConfigRegisters (class in *hvl_ccb.dev.fug*), 105
 FuGProbusVDIRegisters (class in *hvl_ccb.dev.fug*), 106
 FuGProbusVDORegisters (class in *hvl_ccb.dev.fug*), 106
 FuGProbusVMonitorRegisters (class in *hvl_ccb.dev.fug*), 107
 FuGProbusVRegisterGroups (class in *hvl_ccb.dev.fug*), 107
 FuGProbusVSetRegisters (class in *hvl_ccb.dev.fug*), 107
 FuGRampModes (class in *hvl_ccb.dev.fug*), 108
 FuGReadbackChannels (class in *hvl_ccb.dev.fug*), 108
 FuGSerialCommunication (class in *hvl_ccb.dev.fug*), 108
 FuGSerialCommunicationConfig (class in *hvl_ccb.dev.fug*), 109
 FuGTerminators (class in *hvl_ccb.dev.fug*), 110
- ## G
- gate_mode (*T560* property), 48
 gate_polarity (*T560* property), 48
 GateMode (class in *hvl_ccb.dev.highland_t560.base*), 44
 GeneralSockets (class in *hvl_ccb.dev.supercube2015.constants*), 58
 GeneralSupport (class in *hvl_ccb.dev.supercube2015.constants*), 58
 GENERATOR (*TiePieDeviceType* attribute), 72
 generator_start() (*TiePieGeneratorMixin* method), 77
 generator_stop() (*TiePieGeneratorMixin* method), 77
 get() (*AlarmText* class method), 55
 get() (*MeasurementsDividerRatio* class method), 59
 get() (*MeasurementsScaledInput* class method), 60
 get_acceleration() (*NewportSMC100PP* method), 135
 get_acquire_length() (*RTO1024* method), 153
 get_ain() (*LabJack* method), 119
 get_bath_temp() (*LaudaProRp245e* method), 123
 get_by_p_id() (*LabJack.DeviceType* class method), 118
 get_by_p_id() (*LJMCommunicationConfig.DeviceType* class method), 16
 get_cal_current_source() (*LabJack* method), 119
 get_cee16_socket() (*Supercube2015Base* method), 50
 get_channel_offset() (*RTO1024* method), 153
 get_channel_position() (*RTO1024* method), 154
 get_channel_range() (*RTO1024* method), 154
 get_channel_scale() (*RTO1024* method), 154
 get_channel_state() (*RTO1024* method), 154
 get_clock() (*LabJack* method), 120
 get_controller_information() (*NewportSMC100PP* method), 135
 get_current() (*HeinzingerDI* method), 112
 get_dc_volt() (*ILS2T* method), 162
 get_device() (*DeviceSequenceMixin* method), 84
 get_device_by_serial_number() (in module *hvl_ccb.dev.tiepie.base*), 72
 get_device_type() (*LaudaProRp245e* method), 123
 get_devices() (*DeviceSequenceMixin* method), 84
 get_digital_input() (*LabJack* method), 120
 get_door_status() (*Supercube2015Base* method), 50
 get_earthing_manual() (*Supercube2015Base* method), 50

`get_earthing_status()` (*Supercube2015Base method*), 50
`get_error_code()` (*ILS2T method*), 162
`get_error_queue()` (*VisaDevice method*), 172
`get_frequency()` (*Supercube2015WithFU method*), 63
`get_fso_active()` (*Supercube2015WithFU method*), 63
`get_full_scale_mbar()` (*PfeifferTPG method*), 147
`get_full_scale_unitless()` (*PfeifferTPG method*), 148
`get_identification()` (*VisaDevice method*), 172
`get_interface_version()` (*HeinzingerDI method*), 112
`get_max_voltage()` (*Supercube2015WithFU method*), 63
`get_measurement_ratio()` (*Supercube2015Base method*), 50
`get_measurement_voltage()` (*Supercube2015Base method*), 50
`get_motor_configuration()` (*NewportSMC100PP method*), 136
`get_move_duration()` (*NewportSMC100PP method*), 136
`get_negative_software_limit()` (*NewportSMC100PP method*), 136
`get_number_of_recordings()` (*HeinzingerDI method*), 112
`get_objects_node()` (*Client method*), 20
`get_objects_node()` (*Server method*), 22
`get_output()` (*PSI9000 method*), 95
`get_position()` (*ILS2T method*), 162
`get_position()` (*NewportSMC100PP method*), 136
`get_positive_software_limit()` (*NewportSMC100PP method*), 136
`get_power_setup()` (*Supercube2015WithFU method*), 63
`get_primary_current()` (*Supercube2015WithFU method*), 63
`get_primary_voltage()` (*Supercube2015WithFU method*), 63
`get_product_id()` (*LabJack method*), 120
`get_product_name()` (*LabJack method*), 120
`get_product_type()` (*LabJack method*), 120
`get_pulse_energy_and_rate()` (*CryLasLaser method*), 89
`get_reference_point()` (*RTO1024 method*), 154
`get_register()` (*FuGProbusV method*), 105
`get_repetitions()` (*RTO1024 method*), 154
`get_sbus_rh()` (*LabJack method*), 120
`get_sbus_temp()` (*LabJack method*), 120
`get_serial_number()` (*HeinzingerDI method*), 112
`get_serial_number()` (*LabJack method*), 120
`get_state()` (*NewportSMC100PP method*), 137
`get_status()` (*ILS2T method*), 162
`get_status()` (*Supercube2015Base method*), 50
`get_support_input()` (*Supercube2015Base method*), 50
`get_support_output()` (*Supercube2015Base method*), 51
`get_system_lock()` (*PSI9000 method*), 95
`get_t13_socket()` (*Supercube2015Base method*), 51
`get_target_voltage()` (*Supercube2015WithFU method*), 63
`get_temperature()` (*ILS2T method*), 162
`get_timestamps()` (*RTO1024 method*), 154
`get_ui_lower_limits()` (*PSI9000 method*), 96
`get_uip_upper_limits()` (*PSI9000 method*), 96
`get_voltage()` (*HeinzingerDI method*), 112
`get_voltage_current_setpoint()` (*PSI9000 method*), 96
`go_home()` (*NewportSMC100PP method*), 137
`go_to_configuration()` (*NewportSMC100PP method*), 137
GREEN_NOT_READY (*SafetyStatus attribute*), 40
GREEN_READY (*SafetyStatus attribute*), 41
GreenNotReady (*SafetyStatus attribute*), 61
GreenReady (*SafetyStatus attribute*), 61

H

HARDWARE (*CryLasLaser.RepetitionRates attribute*), 89
has_safe_ground (*TiePieOscilloscopeChannelConfig property*), 73
HEAD (*CryLasLaser.AnswersStatus attribute*), 88
HeinzingerConfig (*class in hvl_ccb.dev.heinzinger*), 110
HeinzingerConfig.RecordingsEnum (*class in hvl_ccb.dev.heinzinger*), 111
HeinzingerDI (*class in hvl_ccb.dev.heinzinger*), 111
HeinzingerDI.OutputStatus (*class in hvl_ccb.dev.heinzinger*), 112
HeinzingerPNC (*class in hvl_ccb.dev.heinzinger*), 113
HeinzingerPNC.UnitCurrent (*class in hvl_ccb.dev.heinzinger*), 113
HeinzingerPNC.UnitVoltage (*class in hvl_ccb.dev.heinzinger*), 113
HeinzingerPNCDeviceNotRecognizedException, 114
HeinzingerPNCError, 114
HeinzingerPNCMaxCurrentExceededException, 114
HeinzingerPNCMaxVoltageExceededException, 114
HeinzingerSerialCommunication (*class in hvl_ccb.dev.heinzinger*), 114
HeinzingerSerialCommunicationConfig (*class in hvl_ccb.dev.heinzinger*), 115
HIGH (*LabJack.DIOStatus attribute*), 118
high_resolution (*FuGProbusVSetRegisters property*), 107

home_search_polling_interval	(NewportSMC100PPConfig attribute), 141	hvl_ccb.dev
home_search_timeout	(NewportSMC100PPConfig attribute), 141	module, 173
home_search_type	(NewportSMC100PPConfig attribute), 141	hvl_ccb.dev.base
home_search_velocity	(NewportSMC100PPConfig attribute), 141	module, 83
HOME_STARTED	(NewportSMC100PPSerialCommunication.KubotaFlexcube attribute), 142	hvl_ccb.dev.crylas
HomeSwitch	(NewportSMC100PPConfig.HomeSearch attribute), 140	module, 85
HomeSwitch_and_Index	(NewportSMC100PPConfig.HomeSearch attribute), 140	hvl_ccb.dev.cube
HOMING	(NewportStates attribute), 145	module, 44
HOMING_FROM_RS232	(NewportSMC100PP.StateMessages attribute), 134	hvl_ccb.dev.cube.base
HOMING_FROM_SMC	(NewportSMC100PP.StateMessages attribute), 134	module, 33
HOMING_TIMEOUT	(NewportSMC100PP.MotorErrors attribute), 134	hvl_ccb.dev.cube.constants
horn	(Safety attribute), 61	module, 38
horn()	(Supercube2015Base method), 51	hvl_ccb.dev.cube.picube
host	(ModbusTcpCommunicationConfig attribute), 19	module, 41
host	(OpcUaCommunicationConfig attribute), 21	hvl_ccb.dev.ea_psi9000
host	(TcpCommunicationConfig attribute), 27	module, 95
host	(TelnetCommunicationConfig attribute), 29	hvl_ccb.dev.fug
host	(VisaCommunicationConfig attribute), 32	module, 99
hPascal	(PfeifferTPG.PressureUnits attribute), 147	hvl_ccb.dev.heinzinger
HT	(NewportConfigCommands attribute), 133	module, 110
hvl_ccb		hvl_ccb.dev.highland_t560
module, 182		module, 49
hvl_ccb.comm		hvl_ccb.dev.highland_t560.base
module, 33		module, 44
hvl_ccb.comm.base		hvl_ccb.dev.highland_t560.channel
module, 11		module, 46
hvl_ccb.comm.labjack_ljm		hvl_ccb.dev.highland_t560.device
module, 15		module, 47
hvl_ccb.comm.modbus_tcp		hvl_ccb.dev.labjack
module, 18		module, 116
hvl_ccb.comm.opc		hvl_ccb.dev.lauda
module, 20		module, 123
hvl_ccb.comm.serial		hvl_ccb.dev.mbw973
module, 23		module, 129
hvl_ccb.comm.tcp		hvl_ccb.dev.newport
module, 26		module, 133
hvl_ccb.comm.telnet		hvl_ccb.dev.pfeiffer_tpg
module, 28		module, 146
hvl_ccb.comm.visa		hvl_ccb.dev.rs_rto1024
module, 30		module, 152
hvl_ccb.configuration		hvl_ccb.dev.se_ils2t
module, 178		module, 160
		hvl_ccb.dev.sst_luminos
		module, 166
		hvl_ccb.dev.supercube2015
		module, 64
		hvl_ccb.dev.supercube2015.base
		module, 49
		hvl_ccb.dev.supercube2015.constants
		module, 55
		hvl_ccb.dev.supercube2015.typ_a
		module, 62
		hvl_ccb.dev.technix
		module, 70

`hvl_ccb.dev.technix.base`
 module, 65

`hvl_ccb.dev.technix.device`
 module, 68

`hvl_ccb.dev.tiepie`
 module, 82

`hvl_ccb.dev.tiepie.base`
 module, 71

`hvl_ccb.dev.tiepie.channel`
 module, 73

`hvl_ccb.dev.tiepie.device`
 module, 75

`hvl_ccb.dev.tiepie.generator`
 module, 76

`hvl_ccb.dev.tiepie.i2c`
 module, 78

`hvl_ccb.dev.tiepie.oscilloscope`
 module, 79

`hvl_ccb.dev.tiepie.utils`
 module, 81

`hvl_ccb.dev.utils`
 module, 170

`hvl_ccb.dev.visa`
 module, 171

`hvl_ccb.exception`
 module, 180

`hvl_ccb.experiment_manager`
 module, 180

`hvl_ccb.utils`
 module, 178

`hvl_ccb.utils.conversion_sensor`
 module, 173

`hvl_ccb.utils.conversion_unit`
 module, 174

`hvl_ccb.utils.enum`
 module, 176

`hvl_ccb.utils.typing`
 module, 177

`hvl_ccb.utils.validation`
 module, 177

`hysteresis_compensation` (New-
 portSMC100PPConfig attribute), 141

|

I2C (*TiePieDeviceType* attribute), 72

ID (*FuGProbusIVCommands* attribute), 105

Identification_error (*PfeifferTPG.SensorStatus* attribute), 147

identifier (*LJMCommunicationConfig* attribute), 17

identify_device() (*FuG* method), 100

identify_device() (*HeinzingerPNC* method), 114

identify_sensors() (*PfeifferTPG* method), 148

IKR (*PfeifferTPG.SensorTypes* attribute), 147

IKR11 (*PfeifferTPG.SensorTypes* attribute), 147

IKR9 (*PfeifferTPG.SensorTypes* attribute), 147

ILS2T (class in *hvl_ccb.dev.se_ils2t*), 160

ILS2T.ActionsPtp (class in *hvl_ccb.dev.se_ils2t*), 160

ILS2T.Mode (class in *hvl_ccb.dev.se_ils2t*), 160

ILS2T.Ref16Jog (class in *hvl_ccb.dev.se_ils2t*), 160

ILS2T.State (class in *hvl_ccb.dev.se_ils2t*), 161

ILS2TConfig (class in *hvl_ccb.dev.se_ils2t*), 163

ILS2TException, 164

ILS2TModbusTcpCommunication (class in
 hvl_ccb.dev.se_ils2t), 164

ILS2TModbusTcpCommunicationConfig (class in
 hvl_ccb.dev.se_ils2t), 164

ILS2TRegAddr (class in *hvl_ccb.dev.se_ils2t*), 165

ILS2TRegDatatype (class in *hvl_ccb.dev.se_ils2t*), 165

IMMEDIATELY (*FuGRampModes* attribute), 108

IMPULSE_140KV (*PowerSetup* attribute), 40

IMR (*PfeifferTPG.SensorTypes* attribute), 147

in_1_1 (*GeneralSupport* attribute), 58

in_1_2 (*GeneralSupport* attribute), 58

in_2_1 (*GeneralSupport* attribute), 58

in_2_2 (*GeneralSupport* attribute), 58

in_3_1 (*GeneralSupport* attribute), 58

in_3_2 (*GeneralSupport* attribute), 58

in_4_1 (*GeneralSupport* attribute), 58

in_4_2 (*GeneralSupport* attribute), 58

in_5_1 (*GeneralSupport* attribute), 58

in_5_2 (*GeneralSupport* attribute), 58

in_6_1 (*GeneralSupport* attribute), 58

in_6_2 (*GeneralSupport* attribute), 59

INACTIVE (*CryLasLaser.AnswersStatus* attribute), 88

INACTIVE (*DoorStatus* attribute), 38

inactive (*DoorStatus* attribute), 56

INACTIVE (*EarthingStickStatus* attribute), 39

inactive (*EarthingStickStatus* attribute), 58

inhibit (*Technix* property), 68

init_attenuation (*CryLasAttenuatorConfig* attribute),
 86

init_monitored_nodes() (*OpcUaCommunication*
 method), 20

init_shutter_status (*CryLasLaserConfig* attribute),
 92

initialize() (*NewportSMC100PP* method), 137

INITIALIZED (*ExperimentStatus* attribute), 181

INITIALIZING (*ExperimentStatus* attribute), 181

INITIALIZING (*SafetyStatus* attribute), 41

Initializing (*SafetyStatus* attribute), 61

INPUT (*FuGProbusVRegisterGroups* attribute), 107

INPUT (*GateMode* attribute), 44

input() (*GeneralSupport* class method), 59

input_1 (*MeasurementsDividerRatio* attribute), 59

input_1 (*MeasurementsScaledInput* attribute), 60

input_2 (*MeasurementsScaledInput* attribute), 60

input_3 (*MeasurementsScaledInput* attribute), 60

input_4 (*MeasurementsScaledInput* attribute), 60

- input_range (*TiePieOscilloscopeChannelConfig* property), 74
- INSTALL (*AutoInstallMode* attribute), 44
- INT32 (*ILS2TRegDatatype* attribute), 166
- INT_SYNTHESIZER (*TriggerMode* attribute), 46
- interface_type (*PSI9000VisaCommunicationConfig* attribute), 98
- interface_type (*RTO1024VisaCommunicationConfig* attribute), 159
- interface_type (*VisaCommunicationConfig* attribute), 32
- internal (*LabJack.CjcType* attribute), 117
- INTERNAL (*LaudaProRp245eConfig.ExtControlModeEnum* attribute), 126
- Internal (*PowerSetup* attribute), 61
- InvalidSupercubeStatusError, 49
- IO_SCANNING (*ILS2TRegAddr* attribute), 165
- IoScanningModeValueError, 166
- is_configdataclass (*AsyncCommunicationProtocolConfig* attribute), 13
- is_configdataclass (*BaseCubeConfiguration* attribute), 36
- is_configdataclass (*CryLasAttenuatorConfig* attribute), 86
- is_configdataclass (*CryLasLaserConfig* attribute), 92
- is_configdataclass (*EmptyConfig* attribute), 85, 179
- is_configdataclass (*FuGConfig* attribute), 101
- is_configdataclass (*HeinzingerConfig* attribute), 111
- is_configdataclass (*ILS2TConfig* attribute), 164
- is_configdataclass (*LaudaProRp245eConfig* attribute), 127
- is_configdataclass (*LJMCommunicationConfig* attribute), 17
- is_configdataclass (*LuminoxConfig* attribute), 168
- is_configdataclass (*MBW973Config* attribute), 131
- is_configdataclass (*ModbusTcpCommunicationConfig* attribute), 19
- is_configdataclass (*NewportSMC100PPConfig* attribute), 141
- is_configdataclass (*OpcUaCommunicationConfig* attribute), 21
- is_configdataclass (*PfeifferTPGConfig* attribute), 149
- is_configdataclass (*SupercubeConfiguration* attribute), 53
- is_configdataclass (*T560Config* attribute), 48
- is_configdataclass (*TcpCommunicationConfig* attribute), 27
- is_configdataclass (*TechnixConfig* attribute), 69
- is_configdataclass (*TiePieDeviceConfig* attribute), 71
- is_configdataclass (*VisaCommunicationConfig* attribute), 32
- is_data_ready_polling_interval_sec (*TiePieDeviceConfig* attribute), 71
- is_done() (*MBW973* method), 129
- is_error() (*ExperimentManager* method), 181
- is_finished() (*ExperimentManager* method), 181
- is_generic_type_hint() (in module *hvl_ccb.utils.typing*), 177
- is_in_range() (*ILS2TRegDatatype* method), 166
- is_inactive (*CryLasLaser.LaserStatus* property), 89
- is_measurement_data_ready() (*TiePieOscilloscope* method), 80
- is_open (*Client* property), 20
- is_open (*LJMCommunication* property), 15
- is_open (*OpcUaCommunication* property), 20
- is_open (*SerialCommunication* property), 23
- is_open (*TelnetCommunication* property), 28
- is_polling() (*Poller* method), 170
- is_ready (*CryLasLaser.LaserStatus* property), 89
- is_running() (*ExperimentManager* method), 181
- is_started (*Technix* property), 68
- is_triggered() (*TiePieOscilloscope* method), 80
- is_valid_scale_range_reversed_str() (*PfeifferTPGConfig.Model* method), 149
- ## J
- J (*LabJack.ThermocoupleType* attribute), 119
- jerk_time (*NewportSMC100PPConfig* attribute), 141
- JOG (*ILS2T.Mode* attribute), 160
- jog_run() (*ILS2T* method), 162
- jog_stop() (*ILS2T* method), 162
- JOGGING (*NewportStates* attribute), 146
- JOGGING_FROM_DISABLE (NewportSMC100PP.StateMessages attribute), 134
- JOGGING_FROM_READY (NewportSMC100PP.StateMessages attribute), 134
- JOGN_FAST (*ILS2TRegAddr* attribute), 165
- JOGN_SLOW (*ILS2TRegAddr* attribute), 165
- JR (*NewportConfigCommands* attribute), 133
- ## K
- K (*LabJack.TemperatureUnit* attribute), 118
- K (*LabJack.ThermocoupleType* attribute), 119
- K (*Temperature* attribute), 175
- KELVIN (*Temperature* attribute), 175
- keys() (*AsyncCommunicationProtocolConfig* class method), 13
- keys() (*BaseCubeConfiguration* class method), 36
- keys() (*BaseCubeOpcUaCommunicationConfig* class method), 37
- keys() (*CryLasAttenuatorConfig* class method), 86
- keys() (*CryLasAttenuatorSerialCommunicationConfig* class method), 88

- `keys()` (*CryLasLaserConfig* class method), 92
 - `keys()` (*CryLasLaserSerialCommunicationConfig* class method), 94
 - `keys()` (*EmptyConfig* class method), 85, 179
 - `keys()` (*FuGConfig* class method), 101
 - `keys()` (*FuGSerialCommunicationConfig* class method), 109
 - `keys()` (*HeinzingerConfig* class method), 111
 - `keys()` (*HeinzingerSerialCommunicationConfig* class method), 115
 - `keys()` (*ILS2TConfig* class method), 164
 - `keys()` (*ILS2TModbusTcpCommunicationConfig* class method), 165
 - `keys()` (*LaudaProRp245eConfig* class method), 127
 - `keys()` (*LaudaProRp245eTcpCommunicationConfig* class method), 128
 - `keys()` (*LJMCommunicationConfig* class method), 17
 - `keys()` (*LuminosConfig* class method), 168
 - `keys()` (*LuminosSerialCommunicationConfig* class method), 170
 - `keys()` (*MBW973Config* class method), 131
 - `keys()` (*MBW973SerialCommunicationConfig* class method), 132
 - `keys()` (*ModbusTcpCommunicationConfig* class method), 19
 - `keys()` (*NewportSMC100PPConfig* class method), 141
 - `keys()` (*NewportSMC100PPSerialCommunicationConfig* class method), 145
 - `keys()` (*OpcUaCommunicationConfig* class method), 21
 - `keys()` (*PfeifferTPGConfig* class method), 149
 - `keys()` (*PfeifferTPGSerialCommunicationConfig* class method), 151
 - `keys()` (*PICubeConfiguration* class method), 42
 - `keys()` (*PICubeOpcUaCommunicationConfig* class method), 43
 - `keys()` (*PSI9000Config* class method), 97
 - `keys()` (*PSI9000VisaCommunicationConfig* class method), 99
 - `keys()` (*RTO1024Config* class method), 158
 - `keys()` (*RTO1024VisaCommunicationConfig* class method), 159
 - `keys()` (*SerialCommunicationConfig* class method), 25
 - `keys()` (*SupercubeAOpcUaConfiguration* class method), 64
 - `keys()` (*SupercubeConfiguration* class method), 53
 - `keys()` (*SupercubeOpcUaCommunicationConfig* class method), 54
 - `keys()` (*T560CommunicationConfig* class method), 45
 - `keys()` (*T560Config* class method), 48
 - `keys()` (*TcpCommunicationConfig* class method), 27
 - `keys()` (*TechnixConfig* class method), 69
 - `keys()` (*TechnixSerialCommunicationConfig* class method), 66
 - `keys()` (*TechnixTelnetCommunicationConfig* class method), 67
 - `keys()` (*TelnetCommunicationConfig* class method), 29
 - `keys()` (*TiePieDeviceConfig* class method), 71
 - `keys()` (*VisaCommunicationConfig* class method), 32
 - `keys()` (*VisaDeviceConfig* class method), 173
 - `kV` (*HeinzingerPNC.UnitVoltage* attribute), 114
- ## L
- LabJack* (class in *hvl_ccb.dev.labjack*), 117
 - LabJack.AInRange* (class in *hvl_ccb.dev.labjack*), 117
 - LabJack.BitLimit* (class in *hvl_ccb.dev.labjack*), 117
 - LabJack.CalMicroAmpere* (class in *hvl_ccb.dev.labjack*), 117
 - LabJack.CjcType* (class in *hvl_ccb.dev.labjack*), 117
 - LabJack.ClockFrequency* (class in *hvl_ccb.dev.labjack*), 117
 - LabJack.DeviceType* (class in *hvl_ccb.dev.labjack*), 118
 - LabJack.DIOStatus* (class in *hvl_ccb.dev.labjack*), 118
 - LabJack.TemperatureUnit* (class in *hvl_ccb.dev.labjack*), 118
 - LabJack.ThermocoupleType* (class in *hvl_ccb.dev.labjack*), 118
 - LabJackError*, 122
 - LabJackIdentifierDIOError*, 122
 - laser_off()* (*CryLasLaser* method), 89
 - laser_on()* (*CryLasLaser* method), 90
 - LaudaProRp245e* (class in *hvl_ccb.dev.lauda*), 123
 - LaudaProRp245eCommand* (class in *hvl_ccb.dev.lauda*), 125
 - LaudaProRp245eCommandError*, 126
 - LaudaProRp245eConfig* (class in *hvl_ccb.dev.lauda*), 126
 - LaudaProRp245eConfig.ExtControlModeEnum* (class in *hvl_ccb.dev.lauda*), 126
 - LaudaProRp245eConfig.OperationModeEnum* (class in *hvl_ccb.dev.lauda*), 127
 - LaudaProRp245eTcpCommunication* (class in *hvl_ccb.dev.lauda*), 128
 - LaudaProRp245eTcpCommunicationConfig* (class in *hvl_ccb.dev.lauda*), 128
 - LEM4000S* (class in *hvl_ccb.utils.conversion_sensor*), 173
 - LF* (*FuGTerminators* attribute), 110
 - LF CR* (*FuGTerminators* attribute), 110
 - LINE_1* (*MessageBoard* attribute), 39
 - LINE_10* (*MessageBoard* attribute), 39
 - LINE_11* (*MessageBoard* attribute), 39
 - LINE_12* (*MessageBoard* attribute), 39
 - LINE_13* (*MessageBoard* attribute), 39
 - LINE_14* (*MessageBoard* attribute), 39
 - LINE_15* (*MessageBoard* attribute), 39
 - LINE_2* (*MessageBoard* attribute), 40
 - LINE_3* (*MessageBoard* attribute), 40
 - LINE_4* (*MessageBoard* attribute), 40

LINE_5 (*MessageBoard* attribute), 40
 LINE_6 (*MessageBoard* attribute), 40
 LINE_7 (*MessageBoard* attribute), 40
 LINE_8 (*MessageBoard* attribute), 40
 LINE_9 (*MessageBoard* attribute), 40
 list_devices() (*TiePieOscilloscope* static method), 80
 list_directory() (*RTO1024* method), 154
 LJMCommunication (class in *hvl_ccb.comm.labjack_ljm*), 15
 LJMCommunicationConfig (class in *hvl_ccb.comm.labjack_ljm*), 16
 LJMCommunicationConfig.ConnectionType (class in *hvl_ccb.comm.labjack_ljm*), 16
 LJMCommunicationConfig.DeviceType (class in *hvl_ccb.comm.labjack_ljm*), 16
 LJMCommunicationError, 17
 lm34 (*LabJack.CjcType* attribute), 117
 LMT70A (class in *hvl_ccb.utils.conversion_sensor*), 174
 load_configuration() (*RTO1024* method), 155
 load_device_configuration() (*T560* method), 48
 local_display() (*RTO1024* method), 155
 LOCKED (*DoorStatus* attribute), 38
 locked (*DoorStatus* attribute), 56
 LOW (*LabJack.DIOStatus* attribute), 118
 LOWER_TEMP (*LaudaProRp245eCommand* attribute), 125
 lower_temp (*LaudaProRp245eConfig* attribute), 127
 Luminox (class in *hvl_ccb.dev.sst_luminox*), 166
 LuminoxConfig (class in *hvl_ccb.dev.sst_luminox*), 167
 LuminoxException, 168
 LuminoxMeasurementType (class in *hvl_ccb.dev.sst_luminox*), 168
 LuminoxMeasurementTypeDict (in module *hvl_ccb.dev.sst_luminox*), 169
 LuminoxMeasurementTypeError, 169
 LuminoxMeasurementTypeValue (in module *hvl_ccb.dev.sst_luminox*), 169
 LuminoxOutputMode (class in *hvl_ccb.dev.sst_luminox*), 169
 LuminoxOutputModeError, 169
 LuminoxSerialCommunication (class in *hvl_ccb.dev.sst_luminox*), 169
 LuminoxSerialCommunicationConfig (class in *hvl_ccb.dev.sst_luminox*), 169
 LUT (*LMT70A* attribute), 174
 manual_6 (*EarthingStick* attribute), 57
 MARK (*SerialCommunicationParity* attribute), 25
 max_current (*FuG* property), 100
 max_current (*HeinzingerPNC* property), 114
 max_current (*Technix* property), 68
 max_current (*TechnixConfig* attribute), 70
 max_current_hardware (*FuG* property), 100
 max_current_hardware (*HeinzingerPNC* property), 114
 max_pr_number (*LaudaProRp245eConfig* attribute), 127
 max_pump_level (*LaudaProRp245eConfig* attribute), 127
 max_timeout_retry_nr (*OpcUaCommunicationConfig* attribute), 22
 max_voltage (*FuG* property), 100
 max_voltage (*HeinzingerPNC* property), 114
 max_voltage (*Technix* property), 68
 max_voltage (*TechnixConfig* attribute), 70
 max_voltage_hardware (*FuG* property), 100
 max_voltage_hardware (*HeinzingerPNC* property), 114
 MAXIMUM (*LabJack.ClockFrequency* attribute), 118
 mbar (*PfeifferTPG.PressureUnits* attribute), 147
 MBW973 (class in *hvl_ccb.dev.mbw973*), 129
 MBW973Config (class in *hvl_ccb.dev.mbw973*), 130
 MBW973ControlRunningException, 131
 MBW973Error, 131
 MBW973PumpRunningException, 131
 MBW973SerialCommunication (class in *hvl_ccb.dev.mbw973*), 131
 MBW973SerialCommunicationConfig (class in *hvl_ccb.dev.mbw973*), 131
 measure() (*PfeifferTPG* method), 148
 measure_all() (*PfeifferTPG* method), 148
 measure_current() (*HeinzingerDI* method), 112
 measure_voltage() (*HeinzingerDI* method), 112
 measure_voltage_current() (*PSI9000* method), 96
 MeasurementsDividerRatio (class in *hvl_ccb.dev.supercube2015.constants*), 59
 MeasurementsScaledInput (class in *hvl_ccb.dev.supercube2015.constants*), 59
 MessageBoard (class in *hvl_ccb.dev.cube.constants*), 39
 micro_step_per_full_step_factor (*NewportSMC100PPConfig* attribute), 141
 Micron (*PfeifferTPG.PressureUnits* attribute), 146
 MILLIMETER_MERCURY (*Pressure* attribute), 175
 MINIMUM (*LabJack.ClockFrequency* attribute), 118
 MMHG (*Pressure* attribute), 175
 ModbusTcpCommunication (class in *hvl_ccb.comm.modbus_tcp*), 18
 ModbusTcpCommunicationConfig (class in *hvl_ccb.comm.modbus_tcp*), 19
 ModbusTcpConnectionFailedException, 19
 model (*PfeifferTPGConfig* attribute), 150

module

- hvl_ccb, 182
- hvl_ccb.comm, 33
- hvl_ccb.comm.base, 11
- hvl_ccb.comm.labjack_ljm, 15
- hvl_ccb.comm.modbus_tcp, 18
- hvl_ccb.comm.opc, 20
- hvl_ccb.comm.serial, 23
- hvl_ccb.comm.tcp, 26
- hvl_ccb.comm.telnet, 28
- hvl_ccb.comm.visa, 30
- hvl_ccb.configuration, 178
- hvl_ccb.dev, 173
- hvl_ccb.dev.base, 83
- hvl_ccb.dev.crylas, 85
- hvl_ccb.dev.cube, 44
- hvl_ccb.dev.cube.base, 33
- hvl_ccb.dev.cube.constants, 38
- hvl_ccb.dev.cube.picube, 41
- hvl_ccb.dev.ea_psi9000, 95
- hvl_ccb.dev.fug, 99
- hvl_ccb.dev.heinzinger, 110
- hvl_ccb.dev.highland_t560, 49
- hvl_ccb.dev.highland_t560.base, 44
- hvl_ccb.dev.highland_t560.channel, 46
- hvl_ccb.dev.highland_t560.device, 47
- hvl_ccb.dev.labjack, 116
- hvl_ccb.dev.lauda, 123
- hvl_ccb.dev.mbw973, 129
- hvl_ccb.dev.newport, 133
- hvl_ccb.dev.pfeiffer_tpg, 146
- hvl_ccb.dev.rs_rto1024, 152
- hvl_ccb.dev.se_ils2t, 160
- hvl_ccb.dev.sst_luminox, 166
- hvl_ccb.dev.supercube2015, 64
- hvl_ccb.dev.supercube2015.base, 49
- hvl_ccb.dev.supercube2015.constants, 55
- hvl_ccb.dev.supercube2015.typ_a, 62
- hvl_ccb.dev.technix, 70
- hvl_ccb.dev.technix.base, 65
- hvl_ccb.dev.technix.device, 68
- hvl_ccb.dev.tiepie, 82
- hvl_ccb.dev.tiepie.base, 71
- hvl_ccb.dev.tiepie.channel, 73
- hvl_ccb.dev.tiepie.device, 75
- hvl_ccb.dev.tiepie.generator, 76
- hvl_ccb.dev.tiepie.i2c, 78
- hvl_ccb.dev.tiepie.oscilloscope, 79
- hvl_ccb.dev.tiepie.utils, 81
- hvl_ccb.dev.utils, 170
- hvl_ccb.dev.visa, 171
- hvl_ccb.exception, 180
- hvl_ccb.experiment_manager, 180
- hvl_ccb.utils, 178

- hvl_ccb.utils.conversion_sensor, 173
- hvl_ccb.utils.conversion_unit, 174
- hvl_ccb.utils.enum, 176
- hvl_ccb.utils.typing, 177
- hvl_ccb.utils.validation, 177
- MONITOR_I (*FuGProbusVRegisterGroups* attribute), 107
- MONITOR_V (*FuGProbusVRegisterGroups* attribute), 107
- most_recent_error (*FuGProbusVConfigRegisters* property), 105
- motion_distance_per_full_step (*NewportSMC100PPConfig* attribute), 141
- motor_config (*NewportSMC100PPConfig* property), 141
- move_to_absolute_position() (*NewportSMC100PP* method), 138
- move_to_relative_position() (*NewportSMC100PP* method), 138
- move_wait_sec (*NewportSMC100PPConfig* attribute), 141
- MOVING (*NewportSMC100PP.StateMessages* attribute), 134
- MOVING (*NewportStates* attribute), 146
- MS_NOMINAL_CURRENT (*PSI9000* attribute), 95
- MS_NOMINAL_VOLTAGE (*PSI9000* attribute), 95
- MULTI_COMMANDS_MAX (*VisaCommunication* attribute), 30
- MULTI_COMMANDS_SEPARATOR (*VisaCommunication* attribute), 30

N

- n_channels (*TiePieOscilloscope* property), 80
- n_max_try_get_device (*TiePieDeviceConfig* attribute), 71
- NameEnum (*class in hvl_ccb.utils.enum*), 176
- NAMES (*SerialCommunicationParity* attribute), 25
- names() (*RTO1024.TriggerModes* class method), 153
- namespace_index (*BaseCubeConfiguration* attribute), 36
- namespace_index (*SupercubeConfiguration* attribute), 53
- NATIVEONLY (*TiePieOscilloscopeAutoResolutionModes* attribute), 80
- NED_END_OF_TURN (*NewportSMC100PP.MotorErrors* attribute), 134
- NEG (*ILS2T.Ref16Jog* attribute), 160
- NEG_FAST (*ILS2T.Ref16Jog* attribute), 160
- NEGATIVE (*FuGPolarities* attribute), 104
- NEGATIVE (*Polarity* attribute), 40
- negative_software_limit (*NewportSMC100PPConfig* attribute), 141
- NewportConfigCommands (*class in hvl_ccb.dev.newport*), 133
- NewportControllerError, 133
- NewportError, 133

NewportMotorError, 133
 NewportMotorPowerSupplyWasCutError, 134
 NewportSerialCommunicationError, 145
 NewportSMC100PP (class in *hvl_ccb.dev.newport*), 134
 NewportSMC100PP.MotorErrors (class in *hvl_ccb.dev.newport*), 134
 NewportSMC100PP.StateMessages (class in *hvl_ccb.dev.newport*), 134
 NewportSMC100PPConfig (class in *hvl_ccb.dev.newport*), 140
 NewportSMC100PPConfig.EspStageConfig (class in *hvl_ccb.dev.newport*), 140
 NewportSMC100PPConfig.HomeSearch (class in *hvl_ccb.dev.newport*), 140
 NewportSMC100PPSerialCommunication (class in *hvl_ccb.dev.newport*), 142
 NewportSMC100PPSerialCommunication.ControllerError (class in *hvl_ccb.dev.newport*), 142
 NewportSMC100PPSerialCommunicationConfig (class in *hvl_ccb.dev.newport*), 144
 NewportStates (class in *hvl_ccb.dev.newport*), 145
 NewportUncertainPositionError, 146
 NO (*FuGDigitalVal* attribute), 102
 NO_ERROR (*NewportSMC100PPSerialCommunication.ControllerError* attribute), 142
 NO_REF (*NewportStates* attribute), 146
 NO_REF_ESP_STAGE_ERROR (NewportSMC100PP.StateMessages attribute), 134
 NO_REF_FROM_CONFIG (NewportSMC100PP.StateMessages attribute), 134
 NO_REF_FROM_DISABLED (NewportSMC100PP.StateMessages attribute), 134
 NO_REF_FROM_HOMING (NewportSMC100PP.StateMessages attribute), 135
 NO_REF_FROM_JOGGING (NewportSMC100PP.StateMessages attribute), 135
 NO_REF_FROM_MOVING (NewportSMC100PP.StateMessages attribute), 135
 NO_REF_FROM_READY (NewportSMC100PP.StateMessages attribute), 135
 NO_REF_FROM_RESET (NewportSMC100PP.StateMessages attribute), 135
 No_sensor (*PfeifferTPG.SensorStatus* attribute), 147
 NO_SOURCE (*PowerSetup* attribute), 40
 NOISE (*TiePieGeneratorSignalType* attribute), 77
 noise_level_measurement_channel_1 (*BaseCube-Configuration* attribute), 36
 noise_level_measurement_channel_2 (*BaseCube-Configuration* attribute), 36
 noise_level_measurement_channel_3 (*BaseCube-Configuration* attribute), 36
 noise_level_measurement_channel_4 (*BaseCube-Configuration* attribute), 36
 NONE (*ILS2T.Ref16Jog* attribute), 160
 NONE (*LabJack.ThermocoupleType* attribute), 119
 None (*PfeifferTPG.SensorTypes* attribute), 147
 NONE (*SerialCommunicationParity* attribute), 25
 NORMAL (*RTO1024.TriggerModes* attribute), 153
 noSen (*PfeifferTPG.SensorTypes* attribute), 147
 noSENSOR (*PfeifferTPG.SensorTypes* attribute), 147
 not_defined (*AlarmText* attribute), 56
 nr_trials_activate (*LuminosConfig* attribute), 168
 OpcUaCommunicationProtocol (class in *hvl_ccb.comm.base*), 14
 Number (in module *hvl_ccb.utils.typing*), 177
 number_of_decimals (*HeinzingerConfig* attribute), 111
 number_of_sensors (*PfeifferTPG* property), 148
 O
 OFF (*SerialCommunicationParity* attribute), 25
 OFF (*AutoInstallMode* attribute), 44
 OFF (*FuGDigitalVal* attribute), 102
 OFF (*GateMode* attribute), 44
 OFF (*HeinzingerDI.OutputStatus* attribute), 112
 OFF (*TriggerMode* attribute), 46
 offset (*TiePieGeneratorConfig* property), 77
 OH (*NewportConfigCommands* attribute), 133
 Ok (*PfeifferTPG.SensorStatus* attribute), 147
 on (*FuG* property), 100
 ON (*FuGDigitalVal* attribute), 102
 on (*FuGProbusVDIRegisters* property), 106
 ON (*HeinzingerDI.OutputStatus* attribute), 112
 ON (*ILS2T.State* attribute), 161
 ONE (*HeinzingerConfig.RecordingsEnum* attribute), 111
 ONE (*LabJack.AInRange* attribute), 117
 ONE (*SerialCommunicationStopbits* attribute), 26
 ONE_HUNDREDTH (*LabJack.AInRange* attribute), 117
 ONE_POINT_FIVE (*SerialCommunicationStopbits* attribute), 26
 ONE_TENTH (*LabJack.AInRange* attribute), 117
 ONLYUPWARDSOFFTOZERO (*FuGRampModes* attribute), 108
 OPC_MAX_YEAR (*BaseCube* attribute), 33
 OPC_MIN_YEAR (*BaseCube* attribute), 33
 OpcUaCommunication (class in *hvl_ccb.comm.opc*), 20
 OpcUaCommunicationConfig (class in *hvl_ccb.comm.opc*), 21
 OpcUaCommunicationIOError, 22
 OpcUaCommunicationTimeoutError, 22
 OpcUaSubHandler (class in *hvl_ccb.comm.opc*), 22

OPEN (*DoorStatus* attribute), 38
open (*DoorStatus* attribute), 56
OPEN (*EarthingStickOperation* attribute), 39
OPEN (*EarthingStickStatus* attribute), 39
open (*EarthingStickStatus* attribute), 58
open() (*CommunicationProtocol* method), 14
open() (*LaudaProRp245eTcpCommunication* method), 128
open() (*LJMCommunication* method), 15
open() (*ModbusTcpCommunication* method), 18
open() (*NullCommunicationProtocol* method), 14
open() (*OpcUaCommunication* method), 20
open() (*SerialCommunication* method), 23
open() (*Tcp* method), 26
open() (*TelnetCommunication* method), 28
open() (*VisaCommunication* method), 30
open_interlock (*Technix* property), 69
open_shutter() (*CryLasLaser* method), 90
open_timeout (*VisaCommunicationConfig* attribute), 32
OPENED (*CryLasLaser.AnswersShutter* attribute), 88
OPENED (*CryLasLaserShutterStatus* attribute), 94
operate (*BaseCube* property), 35
operate (*PICube* property), 42
operate() (*Supercube2015Base* method), 51
OPERATION_MODE (*LaudaProRp245eCommand* attribute), 125
operation_mode (*LaudaProRp245eConfig* attribute), 127
optional_defaults() (*AsyncCommunicationProtocolConfig* class method), 13
optional_defaults() (*BaseCubeConfiguration* class method), 36
optional_defaults() (*BaseCubeOpcUaCommunicationConfig* class method), 37
optional_defaults() (*CryLasAttenuatorConfig* class method), 87
optional_defaults() (*CryLasAttenuatorSerialCommunicationConfig* class method), 88
optional_defaults() (*CryLasLaserConfig* class method), 92
optional_defaults() (*CryLasLaserSerialCommunicationConfig* class method), 94
optional_defaults() (*EmptyConfig* class method), 85, 179
optional_defaults() (*FuGConfig* class method), 101
optional_defaults() (*FuGSerialCommunicationConfig* class method), 109
optional_defaults() (*HeinzingerConfig* class method), 111
optional_defaults() (*HeinzingerSerialCommunicationConfig* class method), 115
optional_defaults() (*ILS2TConfig* class method), 164
optional_defaults() (*ILS2TModbusTcpCommunicationConfig* class method), 165
optional_defaults() (*LaudaProRp245eConfig* class method), 127
optional_defaults() (*LaudaProRp245eTcpCommunicationConfig* class method), 129
optional_defaults() (*LJMCommunicationConfig* class method), 17
optional_defaults() (*LuminoxConfig* class method), 168
optional_defaults() (*LuminoxSerialCommunicationConfig* class method), 170
optional_defaults() (*MBW973Config* class method), 131
optional_defaults() (*MBW973SerialCommunicationConfig* class method), 132
optional_defaults() (*ModbusTcpCommunicationConfig* class method), 19
optional_defaults() (*NewportSMC100PPConfig* class method), 141
optional_defaults() (*NewportSMC100PPSerialCommunicationConfig* class method), 145
optional_defaults() (*OpcUaCommunicationConfig* class method), 22
optional_defaults() (*PfeifferTPGConfig* class method), 150
optional_defaults() (*PfeifferTPGSerialCommunicationConfig* class method), 151
optional_defaults() (*PICubeConfiguration* class method), 42
optional_defaults() (*PICubeOpcUaCommunicationConfig* class method), 43
optional_defaults() (*PSI9000Config* class method), 97
optional_defaults() (*PSI9000VisaCommunicationConfig* class method), 99
optional_defaults() (*RTO1024Config* class method), 158
optional_defaults() (*RTO1024VisaCommunicationConfig* class method), 159
optional_defaults() (*SerialCommunicationConfig* class method), 25
optional_defaults() (*SupercubeAOpcUaConfiguration* class method), 64
optional_defaults() (*SupercubeConfiguration* class method), 53
optional_defaults() (*SupercubeOpcUaCommunicationConfig* class method), 54
optional_defaults() (*T560CommunicationConfig*

- class method*), 45
 - `optional_defaults()` (*T560Config class method*), 48
 - `optional_defaults()` (*TcpCommunicationConfig class method*), 27
 - `optional_defaults()` (*TechnixConfig class method*), 70
 - `optional_defaults()` (*TechnixSerialCommunicationConfig class method*), 66
 - `optional_defaults()` (*TechnixTelnetCommunicationConfig class method*), 67
 - `optional_defaults()` (*TelnetCommunicationConfig class method*), 29
 - `optional_defaults()` (*TiePieDeviceConfig class method*), 72
 - `optional_defaults()` (*VisaCommunicationConfig class method*), 32
 - `optional_defaults()` (*VisaDeviceConfig class method*), 173
 - `OSCILLOSCOPE` (*TiePieDeviceType attribute*), 72
 - `OT` (*NewportConfigCommands attribute*), 133
 - `out` (*FuGProbusVDORegisters property*), 106
 - `out_1_1` (*GeneralSupport attribute*), 59
 - `out_1_2` (*GeneralSupport attribute*), 59
 - `out_2_1` (*GeneralSupport attribute*), 59
 - `out_2_2` (*GeneralSupport attribute*), 59
 - `out_3_1` (*GeneralSupport attribute*), 59
 - `out_3_2` (*GeneralSupport attribute*), 59
 - `out_4_1` (*GeneralSupport attribute*), 59
 - `out_4_2` (*GeneralSupport attribute*), 59
 - `out_5_1` (*GeneralSupport attribute*), 59
 - `out_5_2` (*GeneralSupport attribute*), 59
 - `out_6_1` (*GeneralSupport attribute*), 59
 - `out_6_2` (*GeneralSupport attribute*), 59
 - `OUTPUT` (*FuGProbusIVCommands attribute*), 105
 - `OUTPUT` (*GateMode attribute*), 44
 - `output` (*Technix property*), 69
 - `output()` (*GeneralSupport class method*), 59
 - `output_off()` (*FuGProbusIV method*), 104
 - `output_off()` (*HeinzingerDI method*), 112
 - `output_on()` (*HeinzingerDI method*), 113
 - `OUTPUT_POWER_EXCEEDED` (*NewportSMC100PP.MotorErrors attribute*), 134
 - `output_status` (*HeinzingerDI property*), 113
 - `OUTPUTONCMD` (*FuGProbusVRegisterGroups attribute*), 107
 - `OUTPUTX0` (*FuGProbusVRegisterGroups attribute*), 107
 - `OUTPUTX1` (*FuGProbusVRegisterGroups attribute*), 107
 - `OUTPUTX2` (*FuGProbusVRegisterGroups attribute*), 107
 - `OUTPUTXCMD` (*FuGProbusVRegisterGroups attribute*), 107
 - `outX0` (*FuG property*), 100
 - `outX1` (*FuG property*), 100
 - `outX2` (*FuG property*), 100
 - `outXCMD` (*FuG property*), 100
 - `Overrange` (*PfeifferTPG.SensorStatus attribute*), 147
- ## P
- `PA` (*Pressure attribute*), 175
 - `PARAM_MISSING_OR_INVALID` (*NewportSMC100PPSerialCommunication.ControllerErrors attribute*), 142
 - `parity` (*CryLasAttenuatorSerialCommunicationConfig attribute*), 88
 - `parity` (*CryLasLaserSerialCommunicationConfig attribute*), 94
 - `parity` (*FuGSerialCommunicationConfig attribute*), 109
 - `parity` (*HeinzingerSerialCommunicationConfig attribute*), 115
 - `parity` (*LuminoxSerialCommunicationConfig attribute*), 170
 - `parity` (*MBW973SerialCommunicationConfig attribute*), 132
 - `parity` (*NewportSMC100PPSerialCommunicationConfig attribute*), 145
 - `parity` (*PfeifferTPGSerialCommunicationConfig attribute*), 151
 - `Parity` (*SerialCommunicationConfig attribute*), 24
 - `parity` (*SerialCommunicationConfig attribute*), 25
 - `parse_read_measurement_value()` (*LuminoxMeasurementType method*), 168
 - `partial_pressure_o2` (*LuminoxMeasurementType attribute*), 168
 - `Pascal` (*PfeifferTPG.PressureUnits attribute*), 146
 - `PASCAL` (*Pressure attribute*), 175
 - `pause()` (*LaudaProRp245e method*), 123
 - `pause_ramp()` (*LaudaProRp245e method*), 123
 - `PBR` (*PfeifferTPG.SensorTypes attribute*), 147
 - `PEAK_CURRENT_LIMIT` (*NewportSMC100PP.MotorErrors attribute*), 134
 - `peak_output_current_limit` (*NewportSMC100PPConfig attribute*), 141
 - `percent_o2` (*LuminoxMeasurementType attribute*), 168
 - `period` (*T560 property*), 48
 - `PfeifferTPG` (*class in hvl_ccb.dev.pfeiffer_tpg*), 146
 - `PfeifferTPG.PressureUnits` (*class in hvl_ccb.dev.pfeiffer_tpg*), 146
 - `PfeifferTPG.SensorStatus` (*class in hvl_ccb.dev.pfeiffer_tpg*), 147
 - `PfeifferTPG.SensorTypes` (*class in hvl_ccb.dev.pfeiffer_tpg*), 147
 - `PfeifferTPGConfig` (*class in hvl_ccb.dev.pfeiffer_tpg*), 149
 - `PfeifferTPGConfig.Model` (*class in hvl_ccb.dev.pfeiffer_tpg*), 149
 - `PfeifferTPGError`, 150

PfeifferTPGSerialCommunication (class in *hvl_ccb.dev.pfeiffer_tpg*), 150
PfeifferTPGSerialCommunicationConfig (class in *hvl_ccb.dev.pfeiffer_tpg*), 150
PICube (class in *hvl_ccb.dev.cube.picube*), 41
PICubeConfiguration (class in *hvl_ccb.dev.cube.picube*), 42
PICubeOpcUaCommunication (class in *hvl_ccb.dev.cube.picube*), 43
PICubeOpcUaCommunicationConfig (class in *hvl_ccb.dev.cube.picube*), 43
PICubeTestParameterError, 40
PKR (*PfeifferTPG.SensorTypes* attribute), 147
Polarity (class in *hvl_ccb.dev.cube.constants*), 40
Polarity (class in *hvl_ccb.dev.highland_t560.base*), 44
POLARITY (*FuGProbusIVCommands* attribute), 105
polarity (*PICube* property), 42
Poller (class in *hvl_ccb.dev.utils*), 170
polling (*LuminoxOutputMode* attribute), 169
polling_delay_sec (*BaseCubeConfiguration* attribute), 37
polling_interval (*MBW973Config* attribute), 131
polling_interval_sec (*BaseCubeConfiguration* attribute), 37
polling_interval_sec (*TechnixConfig* attribute), 70
polling_period (*CryLasLaserConfig* attribute), 92
polling_timeout (*CryLasLaserConfig* attribute), 92
port (*ModbusTcpCommunicationConfig* attribute), 19
port (*OpcUaCommunicationConfig* attribute), 22
port (*SerialCommunicationConfig* attribute), 25
port (*SupercubeOpcUaCommunicationConfig* attribute), 54
port (*T560CommunicationConfig* attribute), 45
port (*TcpCommunicationConfig* attribute), 27
port (*TechnixTelnetCommunicationConfig* attribute), 67
port (*TelnetCommunicationConfig* attribute), 29
port (*VisaCommunicationConfig* attribute), 32
POS (*ILS2T.Ref16Jog* attribute), 161
POS_END_OF_TURN (*NewportSMC100PP.MotorErrors* attribute), 134
POS_FAST (*ILS2T.Ref16Jog* attribute), 161
POSITION (*ILS2T.RegAddr* attribute), 165
POSITION_OUT_OF_LIMIT (*NewportSMC100PPSerialCommunication.ControllerErrors* attribute), 142
POSITIVE (*FuGPolarities* attribute), 104
POSITIVE (*Polarity* attribute), 40
positive_software_limit (*NewportSMC100PPConfig* attribute), 141
post_force_value() (*NewportSMC100PPConfig* method), 141
post_stop_pause_sec (*TechnixConfig* attribute), 70
POUNDS_PER_SQUARE_INCH (*Pressure* attribute), 175
Power (class in *hvl_ccb.dev.supercube2015.constants*), 60
POWER_INVERTER_220V (*PowerSetup* attribute), 40
power_limit (*PSI9000Config* attribute), 98
power_setup (*PICube* property), 42
PowerSetup (class in *hvl_ccb.dev.cube.constants*), 40
PowerSetup (class in *hvl_ccb.dev.supercube2015.constants*), 60
pre_sample_ratio (*TiePieOscilloscopeConfig* property), 81
prepare_ultra_segmentation() (*RTO1024* method), 155
preserve_type() (in module *hvl_ccb.utils.conversion_unit*), 175
Pressure (class in *hvl_ccb.utils.conversion_unit*), 174
probe_offset (*TiePieOscilloscopeChannelConfig* property), 74
PSI (*Pressure* attribute), 175
PSI9000 (class in *hvl_ccb.dev.ea_psi9000*), 95
PSI9000Config (class in *hvl_ccb.dev.ea_psi9000*), 97
PSI9000Error, 98
PSI9000VisaCommunication (class in *hvl_ccb.dev.ea_psi9000*), 98
PSI9000VisaCommunicationConfig (class in *hvl_ccb.dev.ea_psi9000*), 98
PT100 (*LabJack.ThermocoupleType* attribute), 119
PT1000 (*LabJack.ThermocoupleType* attribute), 119
PT500 (*LabJack.ThermocoupleType* attribute), 119
PTP (*ILS2T.Mode* attribute), 160
PublicPropertiesReprMixin (class in *hvl_ccb.dev.tiepie.utils*), 81
PULSE (*TiePieGeneratorSignalType* attribute), 77
pump_init (*LaudaProRp245eConfig* attribute), 127
PUMP_LEVEL (*LaudaProRp245eCommand* attribute), 125
Q
QIL (*NewportConfigCommands* attribute), 133
QUERY (*FuGProbusIVCommands* attribute), 105
query() (*CryLasLaserSerialCommunication* method), 93
query() (*FuGSerialCommunication* method), 109
query() (*NewportSMC100PPSerialCommunication* method), 143
query() (*PfeifferTPGSerialCommunication* method), 150
query() (*SyncCommunicationProtocol* method), 14
query() (*T560Communication* method), 45
query() (*VisaCommunication* method), 30
query_all() (*CryLasLaserSerialCommunication* method), 93
query_command() (*LaudaProRp245eTcpCommunication* method), 128

- query_multiple() (NewportSMC100PPSerialCommunication method), 143
 query_polling() (Luminox method), 167
 query_status() (Technix method), 69
 QUEUE (AutoInstallMode attribute), 44
 QUICK_STOP (SafetyStatus attribute), 41
 QUICKSTOP (ILS2T.State attribute), 161
 QuickStop (SafetyStatus attribute), 61
 quickstop() (ILS2T method), 162
 quit (Errors attribute), 58
 quit_error() (BaseCube method), 35
 quit_error() (Supercube2015Base method), 51
- ## R
- R (LabJack.ThermocoupleType attribute), 119
 raise_() (FuGErrorcodes method), 103
 RAMP_ACC (ILS2TRegAddr attribute), 165
 RAMP_CONTINUE (LaudaProRp245eCommand attribute), 125
 RAMP_DECEL (ILS2TRegAddr attribute), 165
 RAMP_DELETE (LaudaProRp245eCommand attribute), 125
 RAMP_ITERATIONS (LaudaProRp245eCommand attribute), 125
 RAMP_N_MAX (ILS2TRegAddr attribute), 165
 RAMP_PAUSE (LaudaProRp245eCommand attribute), 125
 RAMP_SELECT (LaudaProRp245eCommand attribute), 125
 RAMP_SET (LaudaProRp245eCommand attribute), 125
 RAMP_START (LaudaProRp245eCommand attribute), 126
 RAMP_STOP (LaudaProRp245eCommand attribute), 126
 RAMP_TYPE (ILS2TRegAddr attribute), 165
 rampmode (FuGProbusVSetRegisters property), 108
 ramprate (FuGProbusVSetRegisters property), 108
 rampstate (FuGProbusVSetRegisters property), 108
 RAMPUPWARDS (FuGRampModes attribute), 108
 RangeEnum (class in hvl_ccb.utils.enum), 176
 RATEDCURRENT (FuGReadbackChannels attribute), 108
 RATEDVOLTAGE (FuGReadbackChannels attribute), 108
 read() (AsyncCommunicationProtocol method), 11
 read() (BaseCube method), 35
 read() (CryLasLaserSerialCommunication method), 93
 read() (LaudaProRp245eTcpCommunication method), 128
 read() (MBW973 method), 130
 read() (OpcUaCommunication method), 21
 read() (Supercube2015Base method), 51
 read() (Tcp method), 26
 read_all() (AsyncCommunicationProtocol method), 11
 read_bytes() (AsyncCommunicationProtocol method), 12
 read_bytes() (SerialCommunication method), 23
 read_bytes() (TelnetCommunication method), 28
 read_float() (MBW973 method), 130
 read_holding_registers() (ModbusTcpCommunication method), 18
 read_input_registers() (ModbusTcpCommunication method), 18
 read_int() (MBW973 method), 130
 read_measurement() (RTO1024 method), 155
 read_measurements() (MBW973 method), 130
 read_name() (LJMCommunication method), 15
 read_nonempty() (AsyncCommunicationProtocol method), 12
 read_output_while_polling (TechnixConfig attribute), 70
 read_resistance() (LabJack method), 120
 read_single_bytes() (SerialCommunication method), 23
 read_streaming() (Luminox method), 167
 read_termination (VisaCommunicationConfig attribute), 32
 read_text() (AsyncCommunicationProtocol method), 12
 read_text() (NewportSMC100PPSerialCommunication method), 143
 read_text_nonempty() (AsyncCommunicationProtocol method), 12
 READ_TEXT_SKIP_PREFIXES (CryLasLaserSerialCommunication attribute), 92
 read_thermocouple() (LabJack method), 121
 readback_data (FuGProbusVConfigRegisters property), 105
 READBACKCHANNEL (FuGProbusIVCommands attribute), 105
 ready (BaseCube property), 35
 READY (CryLasLaser.AnswersStatus attribute), 88
 READY (ILS2T.State attribute), 161
 READY (NewportStates attribute), 146
 ready() (Supercube2015Base method), 51
 READY_ACTIVE (CryLasLaser.LaserStatus attribute), 89
 READY_FROM_DISABLE (NewportSMC100PP.StateMessages attribute), 135
 READY_FROM_HOMING (NewportSMC100PP.StateMessages attribute), 135
 READY_FROM_JOGGING (NewportSMC100PP.StateMessages attribute), 135
 READY_FROM_MOVING (NewportSMC100PP.StateMessages attribute), 135
 READY_INACTIVE (CryLasLaser.LaserStatus attribute), 89
 record_length (TiePieOscilloscopeConfig property), 81

`RED_OPERATE` (*SafetyStatus* attribute), 41
`RED_READY` (*SafetyStatus* attribute), 41
`RedOperate` (*SafetyStatus* attribute), 62
`RedReady` (*SafetyStatus* attribute), 62
`reg_3` (*FuGProbusVDIRegisters* property), 106
`RegAddr` (*ILS2T* attribute), 161
`RegDatatype` (*ILS2T* attribute), 161
`register_pulse_time` (*TechnixConfig* attribute), 70
`RELATIVE` (*TiePieOscilloscopeTriggerLevelMode* attribute), 75
`RELATIVE_POSITION_MOTOR` (*ILS2T.ActionsPtp* attribute), 160
`RELATIVE_POSITION_TARGET` (*ILS2T.ActionsPtp* attribute), 160
`remote` (*Technix* property), 69
`remove_device()` (*DeviceSequenceMixin* method), 84
`require_block_measurement_support` (*TiePieDeviceConfig* attribute), 72
`required_keys()` (*AsyncCommunicationProtocolConfig* class method), 13
`required_keys()` (*BaseCubeConfiguration* class method), 37
`required_keys()` (*BaseCubeOpcUaCommunicationConfig* class method), 38
`required_keys()` (*CryLasAttenuatorConfig* class method), 87
`required_keys()` (*CryLasAttenuatorSerialCommunicationConfig* class method), 88
`required_keys()` (*CryLasLaserConfig* class method), 92
`required_keys()` (*CryLasLaserSerialCommunicationConfig* class method), 94
`required_keys()` (*EmptyConfig* class method), 85, 179
`required_keys()` (*FuGConfig* class method), 101
`required_keys()` (*FuGSerialCommunicationConfig* class method), 110
`required_keys()` (*HeinzingerConfig* class method), 111
`required_keys()` (*HeinzingerSerialCommunicationConfig* class method), 115
`required_keys()` (*ILS2TConfig* class method), 164
`required_keys()` (*ILS2TModbusTcpCommunicationConfig* class method), 165
`required_keys()` (*LaudaProRp245eConfig* class method), 127
`required_keys()` (*LaudaProRp245eTcpCommunicationConfig* class method), 129
`required_keys()` (*LJMCommunicationConfig* class method), 17
`required_keys()` (*LuminoxConfig* class method), 168
`required_keys()` (*LuminoxSerialCommunicationConfig* class method), 170
`required_keys()` (*MBW973Config* class method), 131
`required_keys()` (*MBW973SerialCommunicationConfig* class method), 132
`required_keys()` (*ModbusTcpCommunicationConfig* class method), 19
`required_keys()` (*NewportSMC100PPConfig* class method), 142
`required_keys()` (*NewportSMC100PPSerialCommunicationConfig* class method), 145
`required_keys()` (*OpcUaCommunicationConfig* class method), 22
`required_keys()` (*PfeifferTPGConfig* class method), 150
`required_keys()` (*PfeifferTPGSerialCommunicationConfig* class method), 151
`required_keys()` (*PICubeConfiguration* class method), 43
`required_keys()` (*PICubeOpcUaCommunicationConfig* class method), 43
`required_keys()` (*PSI9000Config* class method), 98
`required_keys()` (*PSI9000VisaCommunicationConfig* class method), 99
`required_keys()` (*RTO1024Config* class method), 159
`required_keys()` (*RTO1024VisaCommunicationConfig* class method), 159
`required_keys()` (*SerialCommunicationConfig* class method), 25
`required_keys()` (*SupercubeAOpcUaConfiguration* class method), 64
`required_keys()` (*SupercubeConfiguration* class method), 53
`required_keys()` (*SupercubeOpcUaCommunicationConfig* class method), 54
`required_keys()` (*T560CommunicationConfig* class method), 45
`required_keys()` (*T560Config* class method), 49
`required_keys()` (*TcpCommunicationConfig* class method), 27
`required_keys()` (*TechnixConfig* class method), 70
`required_keys()` (*TechnixSerialCommunicationConfig* class method), 66
`required_keys()` (*TechnixTelnetCommunicationConfig* class method), 67
`required_keys()` (*TelnetCommunicationConfig* class method), 29
`required_keys()` (*TiePieDeviceConfig* class method), 72
`required_keys()` (*VisaCommunicationConfig* class method), 32
`required_keys()` (*VisaDeviceConfig* class method), 173
`reset` (*BreakdownDetection* attribute), 56
`RESET` (*FuGProbusIVCommands* attribute), 105
`reset()` (*FuGProbusIV* method), 104

- reset() (*NewportSMC100PP method*), 138
 reset() (*VisaDevice method*), 172
 reset_error() (*ILS2T method*), 162
 reset_interface() (*HeinzingerDI method*), 113
 reset_ramp() (*LaudaProRp245e method*), 123
 resolution (*TiePieOscilloscopeConfig property*), 81
 response_sleep_time (*CryLasAttenuatorConfig attribute*), 87
 RISING (*TiePieOscilloscopeTriggerKind attribute*), 75
 RISING_OR_FALLING (*TiePieOscilloscopeTriggerKind attribute*), 75
 RMS_CURRENT_LIMIT (*NewportSMC100PP.MotorErrors attribute*), 134
 rpm_max_init (*ILS2TConfig attribute*), 164
 rs485_address (*NewportSMC100PPConfig attribute*), 142
 RTO1024 (*class in hvl_ccb.dev.rs_rto1024*), 152
 RTO1024.TriggerModes (*class in hvl_ccb.dev.rs_rto1024*), 152
 RTO1024Config (*class in hvl_ccb.dev.rs_rto1024*), 158
 RTO1024Error, 159
 RTO1024VisaCommunication (*class in hvl_ccb.dev.rs_rto1024*), 159
 RTO1024VisaCommunicationConfig (*class in hvl_ccb.dev.rs_rto1024*), 159
 run() (*ExperimentManager method*), 181
 run() (*LaudaProRp245e method*), 123
 run_continuous_acquisition() (*RTO1024 method*), 155
 run_single_acquisition() (*RTO1024 method*), 155
 RUNNING (*ExperimentStatus attribute*), 182
- ## S
- S (*LabJack.ThermocoupleType attribute*), 119
 SA (*NewportConfigCommands attribute*), 133
 SafeGround (*class in hvl_ccb.dev.tiepie.channel*), 73
 Safety (*class in hvl_ccb.dev.supercube2015.constants*), 61
 SafetyStatus (*class in hvl_ccb.dev.cube.constants*), 40
 SafetyStatus (*class in hvl_ccb.dev.supercube2015.constants*), 61
 sample_frequency (*TiePieOscilloscopeConfig property*), 81
 save_configuration() (*RTO1024 method*), 155
 save_device_configuration() (*T560 method*), 48
 save_waveform_history() (*RTO1024 method*), 155
 SCALE (*ILS2TRegAddr attribute*), 165
 ScalingFactorValueError, 166
 screw_scaling (*NewportSMC100PPConfig attribute*), 142
 send_command() (*NewportSMC100PPSerialCommunication method*), 144
 send_command() (*PfeifferTPGSerialCommunication method*), 150
 send_hello() (*Client method*), 20
 send_pulses() (*LabJack method*), 121
 send_stop() (*NewportSMC100PPSerialCommunication method*), 144
 Sensor (*class in hvl_ccb.utils.conversion_sensor*), 174
 Sensor_error (*PfeifferTPG.SensorStatus attribute*), 147
 Sensor_off (*PfeifferTPG.SensorStatus attribute*), 147
 sensor_status (*LuminosMeasurementType attribute*), 168
 SERIAL (*LaudaProRp245eConfig.ExtControlModeEnum attribute*), 126
 serial_number (*LuminosMeasurementType attribute*), 168
 serial_number (*TiePieDeviceConfig attribute*), 72
 SerialCommunication (*class in hvl_ccb.comm.serial*), 23
 SerialCommunicationBytesize (*class in hvl_ccb.comm.serial*), 24
 SerialCommunicationConfig (*class in hvl_ccb.comm.serial*), 24
 SerialCommunicationIOError, 25
 SerialCommunicationParity (*class in hvl_ccb.comm.serial*), 25
 SerialCommunicationStopbits (*class in hvl_ccb.comm.serial*), 25
 Server (*class in hvl_ccb.comm.opc*), 22
 set_acceleration() (*NewportSMC100PP method*), 138
 set_acquire_length() (*RTO1024 method*), 155
 set_ain_differential() (*LabJack method*), 121
 set_ain_range() (*LabJack method*), 121
 set_ain_resistance() (*LabJack method*), 121
 set_ain_resolution() (*LabJack method*), 121
 set_ain_thermocouple() (*LabJack method*), 121
 set_analog_output() (*LabJack method*), 122
 set_attenuation() (*CryLasAttenuator method*), 86
 set_cee16_socket() (*Supercube2015Base method*), 51
 set_channel_offset() (*RTO1024 method*), 156
 set_channel_position() (*RTO1024 method*), 156
 set_channel_range() (*RTO1024 method*), 156
 set_channel_scale() (*RTO1024 method*), 156
 set_channel_state() (*RTO1024 method*), 157
 set_clock() (*LabJack method*), 122
 set_control_mode() (*LaudaProRp245e method*), 123
 set_current() (*HeinzingerDI method*), 113
 set_current() (*HeinzingerPNC method*), 114
 set_digital_output() (*LabJack method*), 122
 set_display_unit() (*PfeifferTPG method*), 148
 set_earthing_manual() (*Supercube2015Base method*), 51
 set_external_temp() (*LaudaProRp245e method*), 124

`set_full_scale_mbar()` (*PfeifferTPG method*), 148
`set_full_scale_unitless()` (*PfeifferTPG method*), 149
`set_init_attenuation()` (*CryLasAttenuator method*), 86
`set_init_shutter_status()` (*CryLasLaser method*), 90
`set_jog_speed()` (*ILS2T method*), 162
`set_lower_limits()` (*PSI9000 method*), 96
`set_max_acceleration()` (*ILS2T method*), 162
`set_max_deceleration()` (*ILS2T method*), 162
`set_max_rpm()` (*ILS2T method*), 163
`set_measuring_options()` (*MBW973 method*), 130
`set_message_board()` (*BaseCube method*), 35
`set_motor_configuration()` (*NewportSMC100PP method*), 138
`set_negative_software_limit()` (*NewportSMC100PP method*), 139
`set_number_of_recordings()` (*HeinzingerDI method*), 113
`set_output()` (*PSI9000 method*), 96
`set_positive_software_limit()` (*NewportSMC100PP method*), 139
`set_pulse_energy()` (*CryLasLaser method*), 90
`set_pump_level()` (*LaudaProRp245e method*), 124
`set_ramp_iterations()` (*LaudaProRp245e method*), 124
`set_ramp_program()` (*LaudaProRp245e method*), 124
`set_ramp_segment()` (*LaudaProRp245e method*), 124
`set_ramp_type()` (*ILS2T method*), 163
`set_reference_point()` (*RTO1024 method*), 157
`set_register()` (*FuGProbusV method*), 105
`set_remote_control()` (*Supercube2015Base method*), 51
`set_repetition_rate()` (*CryLasLaser method*), 90
`set_repetitions()` (*RTO1024 method*), 157
`set_slope()` (*Supercube2015WithFU method*), 63
`set_status_board()` (*BaseCube method*), 35
`set_support_output()` (*Supercube2015Base method*), 52
`set_support_output_impulse()` (*Supercube2015Base method*), 52
`set_system_lock()` (*PSI9000 method*), 96
`set_t13_socket()` (*Supercube2015Base method*), 52
`set_target_voltage()` (*Supercube2015WithFU method*), 63
`set_temp_set_point()` (*LaudaProRp245e method*), 124
`set_transmission()` (*CryLasAttenuator method*), 86
`set_trigger_level()` (*RTO1024 method*), 157
`set_trigger_mode()` (*RTO1024 method*), 158
`set_trigger_source()` (*RTO1024 method*), 158
`set_upper_limits()` (*PSI9000 method*), 96
`set_voltage()` (*HeinzingerDI method*), 113
`set_voltage()` (*HeinzingerPNC method*), 114
`set_voltage_current()` (*PSI9000 method*), 96
`SETCURRENT` (*FuGProbusVRegisterGroups attribute*), 107
`setup` (*Power attribute*), 60
`setvalue` (*FuGProbusVSetRegisters property*), 108
`SETVOLTAGE` (*FuGProbusVRegisterGroups attribute*), 107
`SEVENBITS` (*SerialCommunicationBytesize attribute*), 24
`SHORT_CIRCUIT` (*NewportSMC100PP.MotorErrors attribute*), 134
`shunt` (*LEM4000S attribute*), 174
`SHUTDOWN_CURRENT_LIMIT` (*PSI9000 attribute*), 95
`SHUTDOWN_VOLTAGE_LIMIT` (*PSI9000 attribute*), 95
`ShutterStatus` (*CryLasLaser attribute*), 89
`ShutterStatus` (*CryLasLaserConfig attribute*), 91
`signal_type` (*TiePieGeneratorConfig property*), 77
`SINE` (*TiePieGeneratorSignalType attribute*), 77
`SingleCommDevice` (*class in hvl_ccb.dev.base*), 85
`SIXBITS` (*SerialCommunicationBytesize attribute*), 24
`SIXTEEN` (*HeinzingerConfig.RecordingsEnum attribute*), 111
`SIXTEEN_BIT` (*TiePieOscilloscopeResolution attribute*), 81
`SL` (*NewportConfigCommands attribute*), 133
`SN` (*FuGReadbackChannels attribute*), 108
`SOFTWARE_INTERNAL_SIXTY` (*CryLasLaser.RepetitionRates attribute*), 89
`SOFTWARE_INTERNAL_TEN` (*CryLasLaser.RepetitionRates attribute*), 89
`SOFTWARE_INTERNAL_TWENTY` (*CryLasLaser.RepetitionRates attribute*), 89
`software_revision` (*LuminosMeasurementType attribute*), 168
`SPACE` (*SerialCommunicationParity attribute*), 25
`SPECIALRAMPUPWARDS` (*FuGRampModes attribute*), 108
`spoll()` (*VisaCommunication method*), 31
`spoll_handler()` (*VisaDevice method*), 172
`SQUARE` (*TiePieGeneratorSignalType attribute*), 77
`SR` (*NewportConfigCommands attribute*), 133
`srq_mask` (*FuGProbusVConfigRegisters property*), 105
`srq_status` (*FuGProbusVConfigRegisters property*), 106
`stage_configuration` (*NewportSMC100PPConfig attribute*), 142
`START` (*LaudaProRp245eCommand attribute*), 126
`start()` (*BaseCube method*), 35
`start()` (*CryLasAttenuator method*), 86
`start()` (*CryLasLaser method*), 90
`start()` (*Device method*), 83
`start()` (*DeviceSequenceMixin method*), 84
`start()` (*ExperimentManager method*), 181
`start()` (*FuG method*), 100
`start()` (*FuGProbusIV method*), 104

start() (*HeinzingerDI method*), 113
 start() (*HeinzingerPNC method*), 114
 start() (*ILS2T method*), 163
 start() (*LabJack method*), 122
 start() (*LaudaProRp245e method*), 124
 start() (*Luminos method*), 167
 start() (*MBW973 method*), 130
 start() (*NewportSMC100PP method*), 139
 start() (*PfeifferTPG method*), 149
 start() (*PSI9000 method*), 97
 start() (*RTO1024 method*), 158
 start() (*SingleCommDevice method*), 85
 start() (*Supercube2015Base method*), 52
 start() (*Technix method*), 69
 start() (*TiePieGeneratorMixin method*), 77
 start() (*TiePieI2CHostMixin method*), 78
 start() (*TiePieOscilloscope method*), 80
 start() (*VisaDevice method*), 172
 start_control() (*MBW973 method*), 130
 start_measurement() (*TiePieOscilloscope method*), 80
 start_polling() (*Poller method*), 171
 start_ramp() (*LaudaProRp245e method*), 124
 STARTING (*ExperimentStatus attribute*), 182
 States (*NewportSMC100PP attribute*), 135
 status (*BaseCube property*), 35
 status (*ExperimentManager property*), 181
 status (*FuGProbusVConfigRegisters property*), 106
 status (*FuGProbusVDOResisters property*), 107
 status (*Technix property*), 69
 status_1_closed (*EarthingStick attribute*), 57
 status_1_connected (*EarthingStick attribute*), 57
 status_1_open (*EarthingStick attribute*), 57
 status_2_closed (*EarthingStick attribute*), 57
 status_2_connected (*EarthingStick attribute*), 57
 status_2_open (*EarthingStick attribute*), 57
 status_3_closed (*EarthingStick attribute*), 57
 status_3_connected (*EarthingStick attribute*), 57
 status_3_open (*EarthingStick attribute*), 57
 status_4_closed (*EarthingStick attribute*), 57
 status_4_connected (*EarthingStick attribute*), 57
 status_4_open (*EarthingStick attribute*), 57
 status_5_closed (*EarthingStick attribute*), 57
 status_5_connected (*EarthingStick attribute*), 57
 status_5_open (*EarthingStick attribute*), 57
 status_6_closed (*EarthingStick attribute*), 57
 status_6_connected (*EarthingStick attribute*), 57
 status_6_open (*EarthingStick attribute*), 57
 status_closed() (*EarthingStick class method*), 57
 status_connected() (*EarthingStick class method*), 57
 status_error (*Safety attribute*), 61
 status_green (*Safety attribute*), 61
 status_open() (*EarthingStick class method*), 57
 status_ready_for_red (*Safety attribute*), 61
 status_red (*Safety attribute*), 61
 STATUSBYTE (*FuGReadbackChannels attribute*), 108
 stop (*Errors attribute*), 58
 STOP (*LaudaProRp245eCommand attribute*), 126
 stop() (*BaseCube method*), 35
 stop() (*CryLasLaser method*), 90
 stop() (*Device method*), 83
 stop() (*DeviceSequenceMixin method*), 84
 stop() (*ExperimentManager method*), 181
 stop() (*FuGProbusIV method*), 104
 stop() (*HeinzingerDI method*), 113
 stop() (*ILS2T method*), 163
 stop() (*LabJack method*), 122
 stop() (*LaudaProRp245e method*), 125
 stop() (*Luminos method*), 167
 stop() (*MBW973 method*), 130
 stop() (*NewportSMC100PP method*), 139
 stop() (*PfeifferTPG method*), 149
 stop() (*PSI9000 method*), 97
 stop() (*RTO1024 method*), 158
 stop() (*SingleCommDevice method*), 85
 stop() (*Supercube2015Base method*), 52
 stop() (*Technix method*), 69
 stop() (*TiePieGeneratorMixin method*), 77
 stop() (*TiePieI2CHostMixin method*), 78
 stop() (*TiePieOscilloscope method*), 80
 stop() (*VisaDevice method*), 172
 stop_acquisition() (*RTO1024 method*), 158
 stop_motion() (*NewportSMC100PP method*), 139
 stop_number (*Errors attribute*), 58
 stop_polling() (*Poller method*), 171
 stop_ramp() (*LaudaProRp245e method*), 125
 STOP_SAFETY_STATUSES (in module *hvl_ccb.dev.cube.constants*), 40
 stopbits (*CryLasAttenuatorSerialCommunicationConfig attribute*), 88
 stopbits (*CryLasLaserSerialCommunicationConfig attribute*), 94
 stopbits (*FuGSerialCommunicationConfig attribute*), 110
 stopbits (*HeinzingerSerialCommunicationConfig attribute*), 116
 stopbits (*LuminosSerialCommunicationConfig attribute*), 170
 stopbits (*MBW973SerialCommunicationConfig attribute*), 132
 stopbits (*NewportSMC100PPSerialCommunicationConfig attribute*), 145
 stopbits (*PfeifferTPGSerialCommunicationConfig attribute*), 151
 Stopbits (*SerialCommunicationConfig attribute*), 24
 stopbits (*SerialCommunicationConfig attribute*), 25
 streaming (*LuminosOutputMode attribute*), 169
 StrEnumBase (class in *hvl_ccb.utils.enum*), 176

sub_handler (*BaseCubeOpcUaCommunicationConfig* attribute), 38
 sub_handler (*OpcUaCommunicationConfig* attribute), 22
 sub_handler (*SupercubeOpcUaCommunicationConfig* attribute), 54
 suitable_range() (*TiePieOscilloscopeRange* static method), 74
 Supercube2015Base (class in *hvl_ccb.dev.supercube2015.base*), 49
 Supercube2015WithFU (class in *hvl_ccb.dev.supercube2015.typ_a*), 62
 SupercubeAOpcUaCommunication (class in *hvl_ccb.dev.supercube2015.typ_a*), 63
 SupercubeAOpcUaConfiguration (class in *hvl_ccb.dev.supercube2015.typ_a*), 63
 SupercubeConfiguration (class in *hvl_ccb.dev.supercube2015.base*), 52
 SupercubeOpcEndpoint (class in *hvl_ccb.dev.supercube2015.constants*), 62
 SupercubeOpcUaCommunication (class in *hvl_ccb.dev.supercube2015.base*), 53
 SupercubeOpcUaCommunicationConfig (class in *hvl_ccb.dev.supercube2015.base*), 53
 SupercubeSubscriptionHandler (class in *hvl_ccb.dev.supercube2015.base*), 54
 switchto_green (*Safety* attribute), 61
 switchto_operate (*Safety* attribute), 61
 switchto_ready (*Safety* attribute), 61
 SyncCommunicationProtocol (class in *hvl_ccb.comm.base*), 14
 SyncCommunicationProtocolConfig (class in *hvl_ccb.comm.base*), 14

T

T (*LabJack.ThermocoupleType* attribute), 119
 t13_1 (*GeneralSockets* attribute), 58
 t13_2 (*GeneralSockets* attribute), 58
 t13_3 (*GeneralSockets* attribute), 58
 t13_socket_1 (*BaseCube* attribute), 36
 t13_socket_2 (*BaseCube* attribute), 36
 t13_socket_3 (*BaseCube* attribute), 36
 T13_SOCKET_PORTS (in *hvl_ccb.dev.supercube2015.constants*), 62
 T1MS (*FuGMonitorModes* attribute), 103
 T200MS (*FuGMonitorModes* attribute), 103
 T20MS (*FuGMonitorModes* attribute), 103
 T256US (*FuGMonitorModes* attribute), 103
 T4 (*LabJack.DeviceType* attribute), 118
 T4 (*LJMCommunicationConfig.DeviceType* attribute), 16
 T40MS (*FuGMonitorModes* attribute), 103
 T4MS (*FuGMonitorModes* attribute), 103
 T560 (class in *hvl_ccb.dev.highland_t560.device*), 47
 T560Communication (class in *hvl_ccb.dev.highland_t560.base*), 45
 T560CommunicationConfig (class in *hvl_ccb.dev.highland_t560.base*), 45
 T560Config (class in *hvl_ccb.dev.highland_t560.device*), 48
 T560Error, 46
 T7 (*LabJack.DeviceType* attribute), 118
 T7 (*LJMCommunicationConfig.DeviceType* attribute), 16
 T7_PRO (*LabJack.DeviceType* attribute), 118
 T7_PRO (*LJMCommunicationConfig.DeviceType* attribute), 16
 T800MS (*FuGMonitorModes* attribute), 103
 T80MS (*FuGMonitorModes* attribute), 104
 target_pulse_energy (*CryLasLaser* property), 91
 Tcp (class in *hvl_ccb.comm.tcp*), 26
 TCP (*LJMCommunicationConfig.ConnectionType* attribute), 16
 TcpCommunicationConfig (class in *hvl_ccb.comm.tcp*), 27
 TCPIP_INSTR (*VisaCommunicationConfig.InterfaceType* attribute), 31
 TCPIP_SOCKET (*VisaCommunicationConfig.InterfaceType* attribute), 31
 TEC1 (*CryLasLaser.AnswersStatus* attribute), 88
 TEC2 (*CryLasLaser.AnswersStatus* attribute), 89
 Technix (class in *hvl_ccb.dev.technix.device*), 68
 TechnixConfig (class in *hvl_ccb.dev.technix.device*), 69
 TechnixError, 65
 TechnixFaultError, 65
 TechnixSerialCommunication (class in *hvl_ccb.dev.technix.base*), 65
 TechnixSerialCommunicationConfig (class in *hvl_ccb.dev.technix.base*), 65
 TechnixTelnetCommunication (class in *hvl_ccb.dev.technix.base*), 66
 TechnixTelnetCommunicationConfig (class in *hvl_ccb.dev.technix.base*), 66
 TelnetCommunication (class in *hvl_ccb.comm.telnet*), 28
 TelnetCommunicationConfig (class in *hvl_ccb.comm.telnet*), 28
 TelnetError, 29
 TEMP (*ILS2TRegAddr* attribute), 165
 TEMP_SET_POINT (*LaudaProRp245eCommand* attribute), 126
 temp_set_point_init (*LaudaProRp245eConfig* attribute), 128
 Temperature (class in *hvl_ccb.utils.conversion_unit*), 175
 temperature_sensor (*LuminoxMeasurementType* attribute), 169
 temperature_unit (*LMT70A* attribute), 174
 TEN (*LabJack.AInRange* attribute), 117

TEN (*LabJack.CalMicroAmpere* attribute), 117
 TEN_MHZ (*LabJack.ClockFrequency* attribute), 118
 terminator (*AsyncCommunicationProtocolConfig* attribute), 14
 terminator (*CryLasAttenuatorSerialCommunicationConfig* attribute), 88
 terminator (*CryLasLaserSerialCommunicationConfig* attribute), 94
 TERMINATOR (*FuGProbusIVCommands* attribute), 105
 terminator (*FuGProbusVConfigRegisters* property), 106
 terminator (*FuGSerialCommunicationConfig* attribute), 110
 terminator (*HeinzingerSerialCommunicationConfig* attribute), 116
 terminator (*LaudaProRp245eTcpCommunicationConfig* attribute), 129
 terminator (*LuminosSerialCommunicationConfig* attribute), 170
 terminator (*MBW973SerialCommunicationConfig* attribute), 132
 terminator (*NewportSMC100PPSerialCommunicationConfig* attribute), 145
 terminator (*PfeifferTPGSerialCommunicationConfig* attribute), 151
 terminator (*T560CommunicationConfig* attribute), 46
 terminator_str() (*SerialCommunicationConfig* method), 25
 THIRTY_TWO_BIT (*LabJack.BitLimit* attribute), 117
 TiePieDeviceConfig (class in *hvl_ccb.dev.tiepie.base*), 71
 TiePieDeviceType (class in *hvl_ccb.dev.tiepie.base*), 72
 TiePieError, 72
 TiePieGeneratorConfig (class in *hvl_ccb.dev.tiepie.generator*), 76
 TiePieGeneratorConfigLimits (class in *hvl_ccb.dev.tiepie.generator*), 77
 TiePieGeneratorMixin (class in *hvl_ccb.dev.tiepie.generator*), 77
 TiePieGeneratorSignalType (class in *hvl_ccb.dev.tiepie.generator*), 77
 TiePieHS5 (class in *hvl_ccb.dev.tiepie.device*), 75
 TiePieHS6 (class in *hvl_ccb.dev.tiepie.device*), 76
 TiePieI2CHostConfig (class in *hvl_ccb.dev.tiepie.i2c*), 78
 TiePieI2CHostConfigLimits (class in *hvl_ccb.dev.tiepie.i2c*), 78
 TiePieI2CHostMixin (class in *hvl_ccb.dev.tiepie.i2c*), 78
 TiePieOscilloscope (class in *hvl_ccb.dev.tiepie.oscilloscope*), 79
 TiePieOscilloscopeAutoResolutionModes (class in *hvl_ccb.dev.tiepie.oscilloscope*), 80
 TiePieOscilloscopeChannelConfig (class in *hvl_ccb.dev.tiepie.channel*), 73
 TiePieOscilloscopeChannelConfigLimits (class in *hvl_ccb.dev.tiepie.channel*), 74
 TiePieOscilloscopeChannelCoupling (class in *hvl_ccb.dev.tiepie.channel*), 74
 TiePieOscilloscopeConfig (class in *hvl_ccb.dev.tiepie.oscilloscope*), 80
 TiePieOscilloscopeConfigLimits (class in *hvl_ccb.dev.tiepie.oscilloscope*), 81
 TiePieOscilloscopeRange (class in *hvl_ccb.dev.tiepie.channel*), 74
 TiePieOscilloscopeResolution (class in *hvl_ccb.dev.tiepie.oscilloscope*), 81
 TiePieOscilloscopeTriggerKind (class in *hvl_ccb.dev.tiepie.channel*), 74
 TiePieOscilloscopeTriggerLevelMode (class in *hvl_ccb.dev.tiepie.channel*), 75
 TiePieWS5 (class in *hvl_ccb.dev.tiepie.device*), 76
 timeout (*CryLasAttenuatorSerialCommunicationConfig* attribute), 88
 timeout (*CryLasLaserSerialCommunicationConfig* attribute), 94
 timeout (*FuGSerialCommunicationConfig* attribute), 110
 timeout (*HeinzingerSerialCommunicationConfig* attribute), 116
 timeout (*LuminosSerialCommunicationConfig* attribute), 170
 timeout (*MBW973SerialCommunicationConfig* attribute), 132
 timeout (*NewportSMC100PPSerialCommunicationConfig* attribute), 145
 timeout (*PfeifferTPGSerialCommunicationConfig* attribute), 151
 timeout (*SerialCommunicationConfig* attribute), 25
 timeout (*TelnetCommunicationConfig* attribute), 29
 timeout (*VisaCommunicationConfig* attribute), 32
 timeout_interval (*BaseCubeConfiguration* attribute), 37
 timeout_status_change (*BaseCubeConfiguration* attribute), 37
 timeout_test_parameters (*PICubeConfiguration* attribute), 43
 Torr (*PfeifferTPG.PressureUnits* attribute), 146
 TORR (*Pressure* attribute), 175
 TPG25xA (*PfeifferTPGConfig.Model* attribute), 149
 TPGx6x (*PfeifferTPGConfig.Model* attribute), 149
 TPR (*PfeifferTPG.SensorTypes* attribute), 147
 transmission (*CryLasAttenuator* property), 86
 TRIANGLE (*TiePieGeneratorSignalType* attribute), 77
 trigger_enabled (*TiePieOscilloscopeChannelConfig* property), 74
 trigger_hysteresis (*TiePieOscilloscopeChannel-*

Config property), 74
 trigger_kind (*TiePieOscilloscopeChannelConfig property*), 74
 trigger_level (*T560 property*), 48
 trigger_level (*TiePieOscilloscopeChannelConfig property*), 74
 trigger_level_mode (*TiePieOscilloscopeChannelConfig property*), 74
 trigger_mode (*T560 property*), 48
 trigger_timeout (*TiePieOscilloscopeConfig property*), 81
 triggered (*BreakdownDetection attribute*), 56
 TriggerMode (*class in hvl_ccb.dev.highland_t560.base*), 46
 TWELVE_BIT (*TiePieOscilloscopeResolution attribute*), 81
 TWELVE_HUNDRED_FIFTY_KHZ (*LabJack.ClockFrequency attribute*), 118
 TWENTY_FIVE_HUNDRED_KHZ (*LabJack.ClockFrequency attribute*), 118
 TWENTY_MHZ (*LabJack.ClockFrequency attribute*), 118
 TWENTY_VOLT (*TiePieOscilloscopeRange attribute*), 74
 TWO (*HeinzingerConfig.RecordingsEnum attribute*), 111
 TWO (*SerialCommunicationStopbits attribute*), 26
 TWO_HUNDRED (*LabJack.CalMicroAmpere attribute*), 117
 TWO_HUNDRED_MILLI_VOLT (*TiePieOscilloscopeRange attribute*), 74
 TWO_VOLT (*TiePieOscilloscopeRange attribute*), 74

U

Underrange (*PfeifferTPG.SensorStatus attribute*), 147
 Unit (*class in hvl_ccb.utils.conversion_unit*), 175
 unit (*ILS2TModbusTcpCommunicationConfig attribute*), 165
 unit (*ModbusTcpCommunicationConfig attribute*), 19
 unit (*PfeifferTPG property*), 149
 unit() (*RangeEnum class method*), 176
 unit_current (*HeinzingerPNC property*), 114
 unit_voltage (*HeinzingerPNC property*), 114
 UNKNOWN (*HeinzingerDI.OutputStatus attribute*), 112
 UNKNOWN (*HeinzingerPNC.UnitCurrent attribute*), 113
 UNKNOWN (*HeinzingerPNC.UnitVoltage attribute*), 114
 UNKNOWN (*TiePieGeneratorSignalType attribute*), 77
 UNKNOWN (*TiePieOscilloscopeAutoResolutionModes attribute*), 80
 UNKNOWN (*TiePieOscilloscopeTriggerLevelMode attribute*), 75
 UNREADY_INACTIVE (*CryLasLaser.LaserStatus attribute*), 89
 update_laser_status() (*CryLasLaser method*), 91
 update_parameter (*OpcUaCommunicationConfig attribute*), 22
 update_repetition_rate() (*CryLasLaser method*), 91

update_shutter_status() (*CryLasLaser method*), 91
 update_target_pulse_energy() (*CryLasLaser method*), 91
 UpdateEspStageInfo (*NewportSMC100PPConfig.EspStageConfig attribute*), 140
 UPPER_TEMP (*LaudaProRp245eCommand attribute*), 126
 upper_temp (*LaudaProRp245eConfig attribute*), 128
 USB (*LaudaProRp245eConfig.ExtControlModeEnum attribute*), 126
 USB (*LJMCommunicationConfig.ConnectionType attribute*), 16
 use_external_clock() (*T560 method*), 48
 user_position_offset (*NewportSMC100PPConfig attribute*), 142
 user_steps() (*ILS2T method*), 163

V

V (*HeinzingerPNC.UnitVoltage attribute*), 114
 VA (*NewportConfigCommands attribute*), 133
 validate_and_resolve_host() (*in module hvl_ccb.utils.validation*), 177
 validate_bool() (*in module hvl_ccb.utils.validation*), 177
 validate_number() (*in module hvl_ccb.utils.validation*), 177
 validate_pump_level() (*LaudaProRp245e method*), 125
 validate_tcp_port() (*in module hvl_ccb.utils.validation*), 178
 value (*FuGProbusVMonitorRegisters property*), 107
 value (*LabJack.AInRange property*), 117
 value_raw (*FuGProbusVMonitorRegisters property*), 107
 ValueEnum (*class in hvl_ccb.utils.enum*), 176
 VB (*NewportConfigCommands attribute*), 133
 velocity (*NewportSMC100PPConfig attribute*), 142
 visa_backend (*VisaCommunicationConfig attribute*), 32
 VisaCommunication (*class in hvl_ccb.comm.visa*), 30
 VisaCommunicationConfig (*class in hvl_ccb.comm.visa*), 31
 VisaCommunicationConfig.InterfaceType (*class in hvl_ccb.comm.visa*), 31
 VisaCommunicationError, 32
 VisaDevice (*class in hvl_ccb.dev.visa*), 171
 VisaDeviceConfig (*class in hvl_ccb.dev.visa*), 172
 VOLT (*ILS2TRegAddr attribute*), 165
 Volt (*PfeifferTPG.PressureUnits attribute*), 146
 voltage (*FuG property*), 101
 VOLTAGE (*FuGProbusIVCommands attribute*), 105
 VOLTAGE (*FuGReadbackChannels attribute*), 108
 voltage (*Technix property*), 69
 voltage_actual (*PICube property*), 42
 voltage_lower_limit (*PSI9000Config attribute*), 98

voltage_max (*PICube* property), 42
 voltage_max (*Power* attribute), 60
 voltage_monitor (*FuG* property), 101
 voltage_primary (*PICube* property), 42
 voltage_primary (*Power* attribute), 60
 voltage_regulation (*Technix* property), 69
 voltage_slope (*Power* attribute), 60
 voltage_target (*Power* attribute), 60
 voltage_upper_limit (*PSI9000Config* attribute), 98

W

WAIT_AFTER_WRITE (*VisaCommunication* attribute), 30
 wait_for_polling_result() (*Poller* method), 171
 wait_operation_complete() (*VisaDevice* method), 172
 wait_sec_initialisation (*PSI9000Config* attribute), 98
 wait_sec_max_disable (*ILS2TConfig* attribute), 164
 wait_sec_post_absolute_position (*ILS2TConfig* attribute), 164
 wait_sec_post_activate (*LuminoxConfig* attribute), 168
 wait_sec_post_cannot_disable (*ILS2TConfig* attribute), 164
 wait_sec_post_enable (*ILS2TConfig* attribute), 164
 wait_sec_post_relative_step (*ILS2TConfig* attribute), 164
 wait_sec_pre_read_or_write (*Lau-daProRp245eTcpCommunicationConfig* attribute), 129
 wait_sec_read_text_nonempty (*AsyncCommunicationProtocolConfig* attribute), 14
 wait_sec_read_text_nonempty (*FuGSerialCommunicationConfig* attribute), 110
 wait_sec_read_text_nonempty (*HeinzingerSerialCommunicationConfig* attribute), 116
 wait_sec_retry_get_device (*TiePieDeviceConfig* attribute), 72
 wait_sec_settings_effect (*PSI9000Config* attribute), 98
 wait_sec_stop_commands (*FuGConfig* attribute), 101
 wait_sec_stop_commands (*HeinzingerConfig* attribute), 111
 wait_sec_system_lock (*PSI9000Config* attribute), 98
 wait_sec_trials_activate (*LuminoxConfig* attribute), 168
 wait_timeout_retry_sec (*OpcUaCommunicationConfig* attribute), 22
 wait_until_motor_initialized() (*NewportSMC100PP* method), 140
 wait_until_ready() (*CryLasLaser* method), 91
 waveform (*TiePieGeneratorConfig* property), 77
 WIFI (*LJMCommunicationConfig.ConnectionType* attribute), 16

wrap_libtiepie_exception() (in module *hvl_ccb.dev.tiepie.base*), 72
 write() (*AsyncCommunicationProtocol* method), 12
 write() (*BaseCube* method), 36
 write() (*MBW973* method), 130
 write() (*OpcUaCommunication* method), 21
 write() (*Supercube2015Base* method), 52
 write() (*Tcp* method), 26
 write() (*VisaCommunication* method), 31
 write_absolute_position() (*ILS2T* method), 163
 write_bytes() (*AsyncCommunicationProtocol* method), 12
 write_bytes() (*SerialCommunication* method), 23
 write_bytes() (*TelnetCommunication* method), 28
 write_command() (*Lau-daProRp245eTcpCommunication* method), 128
 write_name() (*LJMCommunication* method), 16
 write_names() (*LJMCommunication* method), 16
 write_registers() (*ModbusTcpCommunication* method), 18
 write_relative_step() (*ILS2T* method), 163
 write_termination (*VisaCommunicationConfig* attribute), 32
 write_text() (*AsyncCommunicationProtocol* method), 13
 WRONG_ESP_STAGE (*NewportSMC100PP.MotorErrors* attribute), 134

X

x_stat (*FuGProbusVDIRegisters* property), 106
 XOUTPUTS (*FuGProbusIVCommands* attribute), 105

Y

YES (*FuGDigitalVal* attribute), 102

Z

ZX (*NewportConfigCommands* attribute), 133