
HVL Common Code Base Documentation

Release 0.12.0

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

Oct 17, 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	187
5.1	Types of Contributions	187
5.2	Get Started!	188
5.3	Merge Request Guidelines	189
5.4	Tips	189
5.5	Deploying	190
6	Credits	193
6.1	Maintainers	193
6.2	Authors	193
6.3	Contributors	193
7	History	195
7.1	0.12.0 (2022-10-17)	195
7.2	0.11.1 (2022-09-15)	195
7.3	0.11.0 (2022-06-22)	195
7.4	0.10.3 (2022-03-21)	196
7.5	0.10.2 (2022-02-28)	196
7.6	0.10.1 (2022-01-24)	196
7.7	0.10.0 (2022-01-17)	196
7.8	0.9.0 (2022-01-07)	197
7.9	0.8.5 (2021-11-05)	198
7.10	0.8.4 (2021-10-22)	198
7.11	0.8.3 (2021-09-27)	198
7.12	0.8.2 (2021-08-27)	198
7.13	0.8.1 (2021-08-13)	198

7.14	0.8.0 (2021-07-02)	199
7.15	0.7.1 (2021-06-04)	199
7.16	0.7.0 (2021-05-25)	199
7.17	0.6.1 (2021-05-08)	199
7.18	0.6.0 (2021-04-23)	200
7.19	0.5.0 (2020-11-11)	200
7.20	0.4.0 (2020-07-16)	201
7.21	0.3.5 (2020-02-18)	202
7.22	0.3.4 (2019-12-20)	202
7.23	0.3.3 (2019-05-08)	202
7.24	0.3.2 (2019-05-08)	202
7.25	0.3.1 (2019-05-02)	202
7.26	0.3 (2019-05-02)	203
7.27	0.2.1 (2019-04-01)	203
7.28	0.2.0 (2019-03-31)	203
7.29	0.1.0 (2019-02-06)	203
8	Indices and tables	205
	Python Module Index	207
	Index	209

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
Bench Multimeter	Fluke 8845A and 8846A 6.5 Digit Precision Multimeter
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	

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 Luminox 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	<p>Technix capacitor charger</p> <p>Fluke 8845A and 8846</p>
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 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. Furthermore it is possible to map two ranges on to each other. This can be useful to convert between for example 4 - 20 mA and 0 - 10 V, both of them are common as sensor out- or input. Moreover, a subclass allows the mapping of a bit-range to any other range. For example a 12 bit number (0-4095) to 0 - 10. All utilities 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

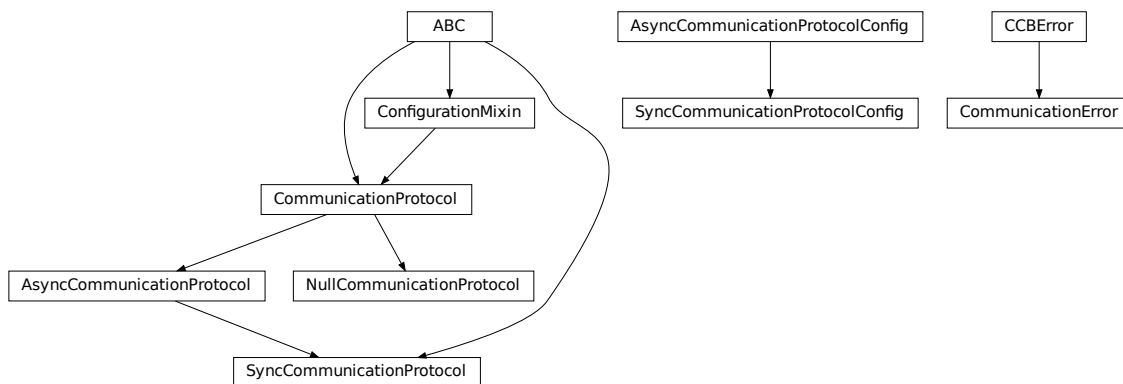
4.1 hvl_ccb

4.1.1 Subpackages

`hvl_ccb.comm`

Submodules

`hvl_ccb.comm.base`



Module with base classes for communication protocols.

class `AsyncCommunicationProtocol`(*config*)

Bases: `CommunicationProtocol`

Abstract base class for asynchronous communication protocols

static `config_cls()` → `Type[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'\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 CommunicationErrorBases: *CSError***class CommunicationProtocol(config)**Bases: *ConfigurationMixin*, 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: *CommunicationProtocol*

Communication protocol that does nothing.

close() → *None*

Void close function.

static config_cls() → Type[*EmptyConfig*]

Empty configuration

Returns

EmptyConfig

open() → *None*

Void open function.

class SyncCommunicationProtocol(*config*)

Bases: *AsyncCommunicationProtocol*, ABC

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

static config_cls() → Type[*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. Possibly raises an exception.

Parameters

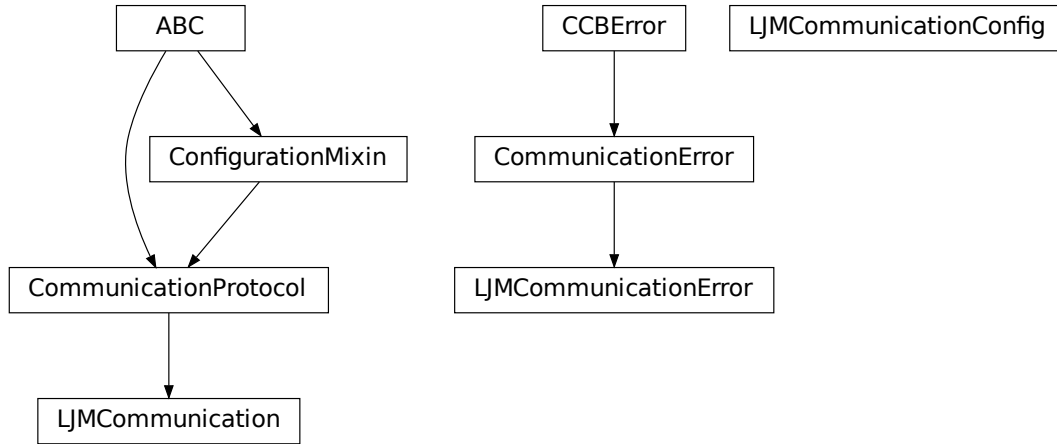
command – Command to send

Returns

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

class SyncCommunicationProtocolConfig(*terminator: bytes = b'\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: *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: `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[Real, Sequence[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: 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, 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, DeviceType] = 'ANY', connection_type: Union[str, ConnectionType] = 'ANY', identifier: str = 'ANY'*)

Bases: object

Configuration dataclass for *LJMCommunication*.

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

Bases: *AutoNumberNameEnum*

LabJack connection type.

ANY = 1

ETHERNET = 4

TCP = 3

USB = 2

WIFI = 5

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

Bases: *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[*DeviceType*, List[*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, *ConnectionType*] = 'ANY'

Can be either string or of enum *ConnectionType*.

device_type: Union[str, *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/ljmmopens/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]

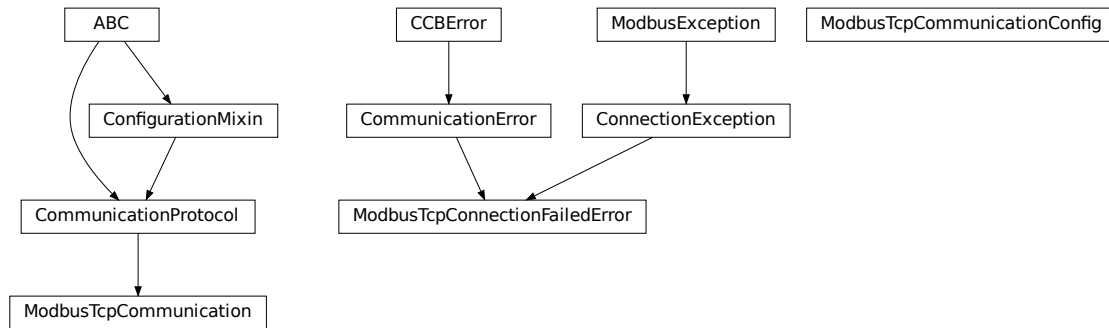
Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on instantiation.

Returns

a list of strings containing all required keys.

exception LJMCommunicationErrorBases: *CommunicationError*

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: *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*ModbusTcpConnectionFailedError* – 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

Returnslist 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, IPv4Address, 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, IPv4Address, 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.

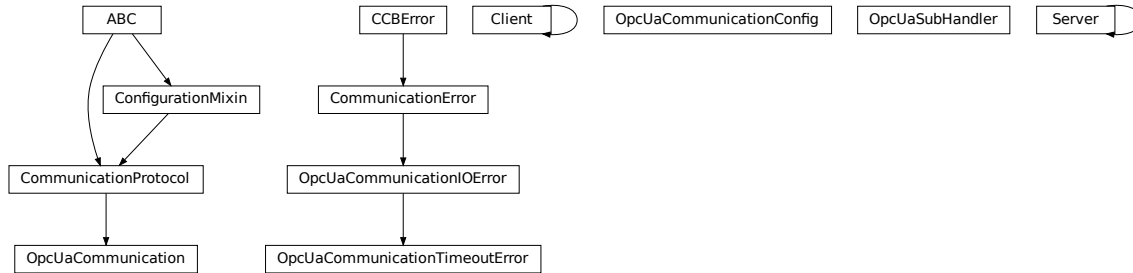
unit: int

exception ModbusTcpConnectionFailedError(string="")

Bases: ConnectionException, *CommunicationError*

Error raised when the connection failed.

hvl_ccb.comm.opc



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

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

Bases: 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: *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. —DEPRECATED! DO NOT USE!!!—

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.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: 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, IPv4Address, 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: *OpcUaSubHandler* = <hvl_ccb.comm.opc.OpcUaSubHandler object>

object to use for handling subscriptions.

update_parameter: 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`, `CommunicationError`

OPC-UA communication I/O error.

```
exception OpcUaCommunicationTimeoutError
```

Bases: `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)
```

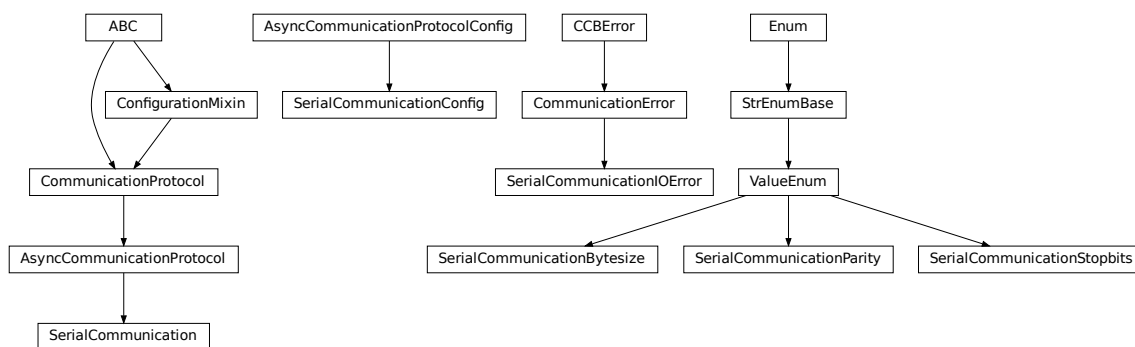
```
class Server(shelf_file=None, tloop=None)
```

Bases: `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: `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, qualname=None,
                                   type=None, start=1, boundary=None)
```

Bases: *ValueEnum*

Serial communication bytesize.

EIGHTBITS = 8

FIVEBITS = 5

SEVENBITS = 7

SIXBITS = 6

```
class SerialCommunicationConfig(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: Optional[str] = None, baudrate: int = 9600, parity: Union[str,
                                SerialCommunicationParity] = SerialCommunicationParity.NONE,
                                stopbits: Union[int, float, SerialCommunicationStopbits] =
                                SerialCommunicationStopbits.ONE, bytesize: Union[int,
                                SerialCommunicationBytesize] =
                                SerialCommunicationBytesize.EIGHTBITS, timeout: Union[int, float] = 2)
```

Bases: [AsyncCommunicationProtocolConfig](#)

Configuration dataclass for [SerialCommunication](#).

Bytesize

alias of [SerialCommunicationBytesize](#)

Parity

alias of [SerialCommunicationParity](#)

Stopbits

alias of [SerialCommunicationStopbits](#)

baudrate: **int** = 9600

Baudrate of the serial port

bytesize: **Union**[**int**, [SerialCommunicationBytesize](#)] = 8

Size of a byte, 5 to 8

clean_values()

create_serial_port() → [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, *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, *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, *CommunicationError*

Serial communication related I/O errors.

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

Bases: *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, qualname=None, type=None, start=1, boundary=None)

Bases: *ValueEnum*

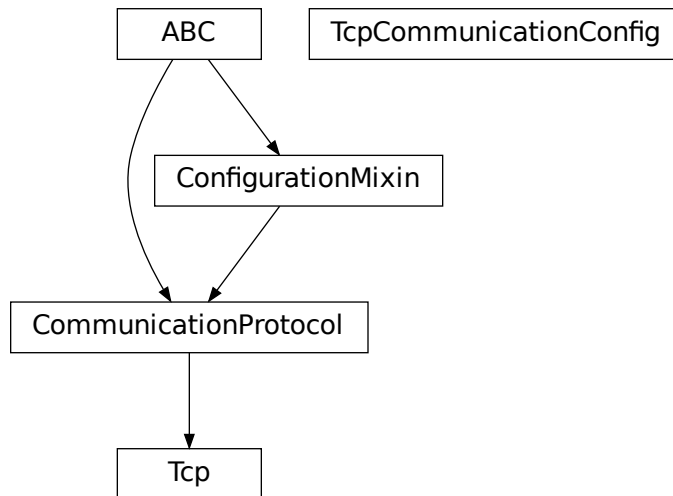
Serial communication stopbits.

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: *CommunicationProtocol*

Tcp Communication Protocol.

```
close() → None
```

Close TCP connection.

```
static config_cls() → Type[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, IPv4Address, 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, IPv4Address, 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.

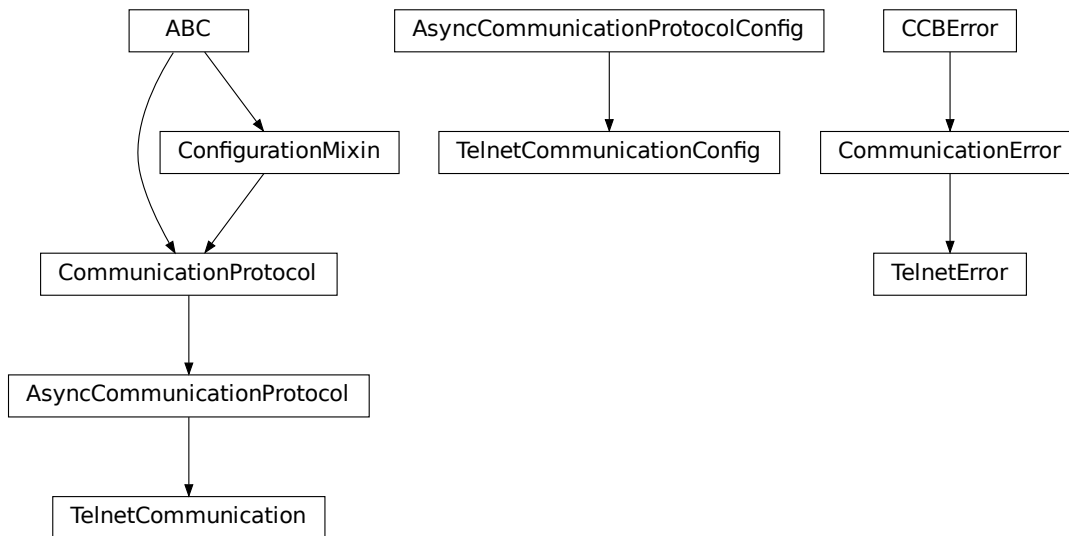
port: int = 54321

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: [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'\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,
                                host: Optional[Union[str, IPv4Address, IPv6Address]] = None, port: int =
                                0, timeout: Union[int, float] = 0.2)
```

Bases: *AsyncCommunicationProtocolConfig*

Configuration dataclass for *TelnetCommunication*.

clean_values()

create_telnet() → Optional[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, IPv4Address, 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
```

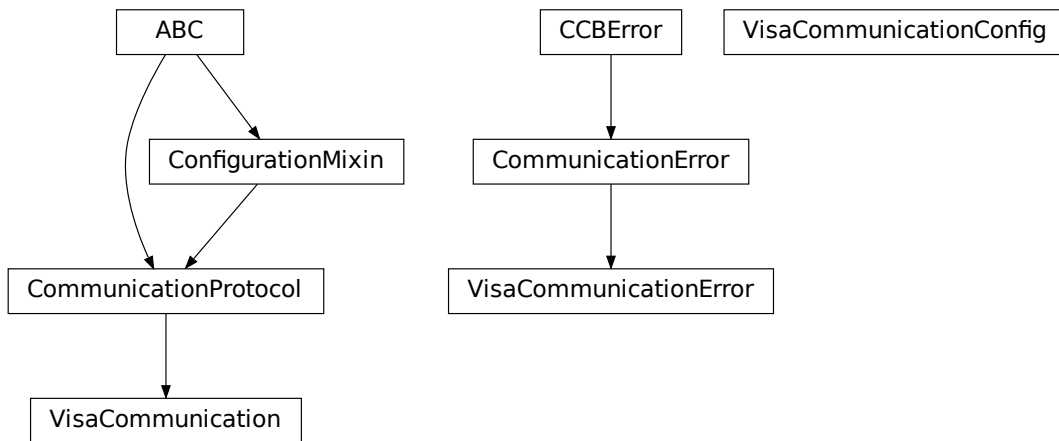
Timeout for reading a line

exception TelnetError

Bases: `OSError`, `CommunicationError`

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: `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[*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, IPv4Address, IPv6Address], interface_type: Union[str,
    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, qualname=None, type=None,
    start=1, boundary=None)
```

Bases: *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, IPv4Address, IPv6Address]

interface_type: Union[str, [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`, `CommunicationError`

Base class for VisaCommunication errors.

Module contents

Communication protocols subpackage.

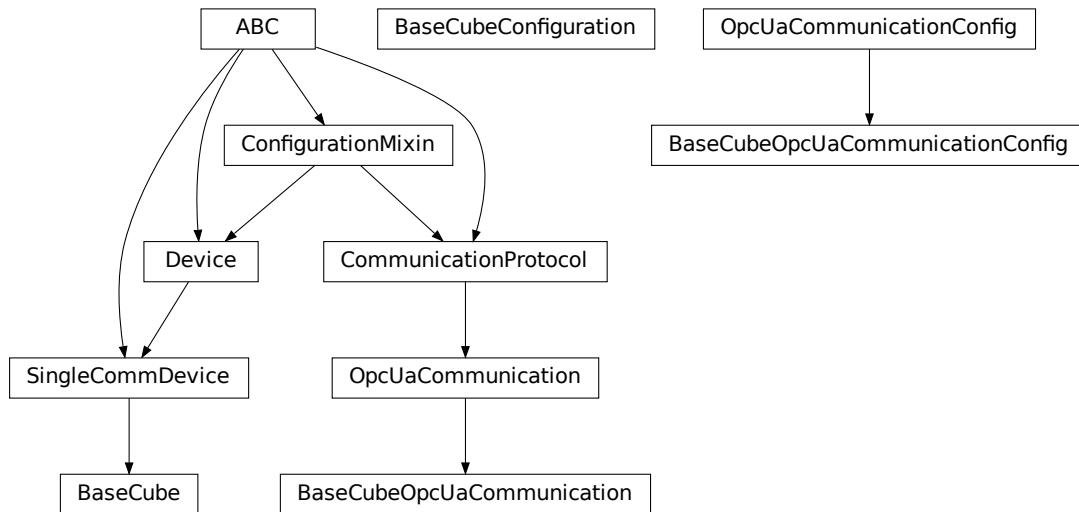
`hvl_ccb.dev`

Subpackages

`hvl_ccb.dev.cube`

Submodules

hvl_ccb.dev.cube.base



Classes for the BaseCube device.

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

Bases: `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) → 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: *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() `→ 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: *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:
~asyncua.ua.uaprotoocol_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: *OpCuaCommunicationConfig*

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

endpoint_name: `_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**() → 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.

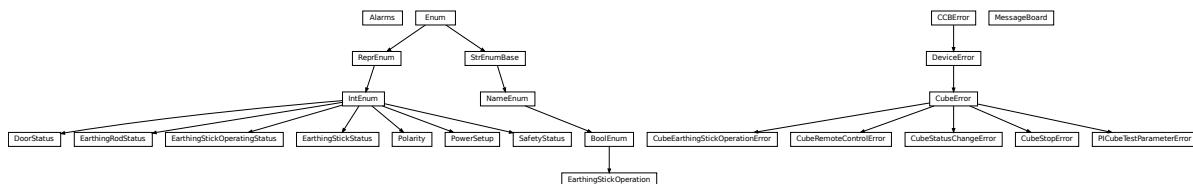
Returns

a list of strings containing all required keys.

sub_handler: *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: *CubeError*

exception CubeError

Bases: *DeviceError*

exception CubeRemoteControlError

Bases: *CubeError*

exception CubeStatusChangeError

Bases: *CubeError*

exception CubeStopError

Bases: *CubeError*

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

Bases: *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, qualname=None, type=None, start=1, boundary=None*)

Bases: *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, qualname=None, type=None, start=1, boundary=None*)

Bases: *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, qualname=None, type=None, start=1, boundary=None*)

Bases: *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, qualname=None, type=None, start=1, boundary=None*)

Bases: 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, qualname=None, type=None, start=1, boundary=None*)

Bases: _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: [CubeError](#)

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

Bases: IntEnum

An enumeration.

NEGATIVE = 0

POSITIVE = 1

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

Bases: 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[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, qualname=None, type=None, start=1,
                    boundary=None)
```

Bases: 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

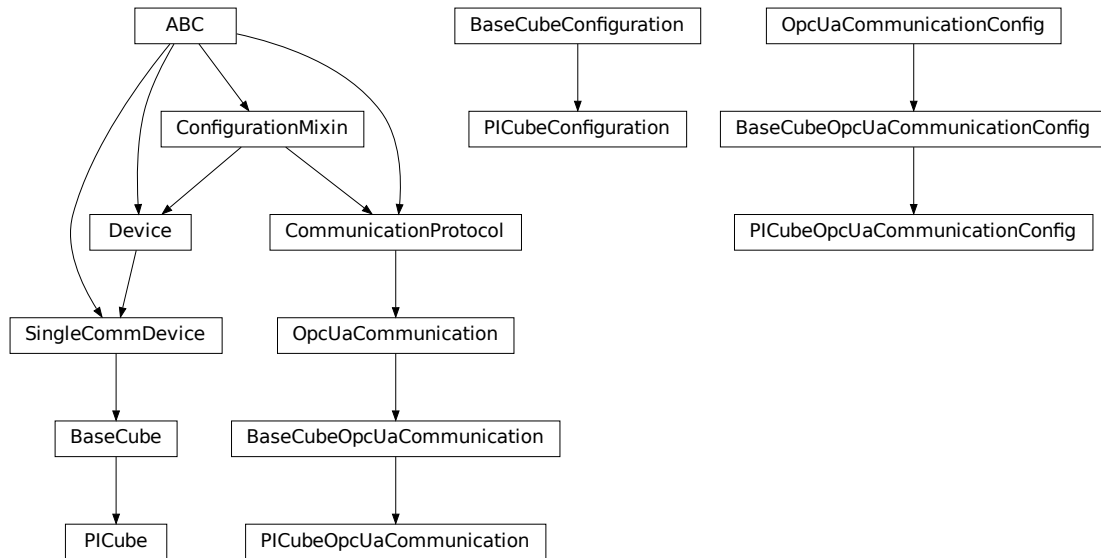
INITIALIZING = 0

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: `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[Polarity]`

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

property power_setup: `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: `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: *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 0x7f9c72f48190>, update_parameter: asyncua.ua.uaproTOCOL_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: *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.

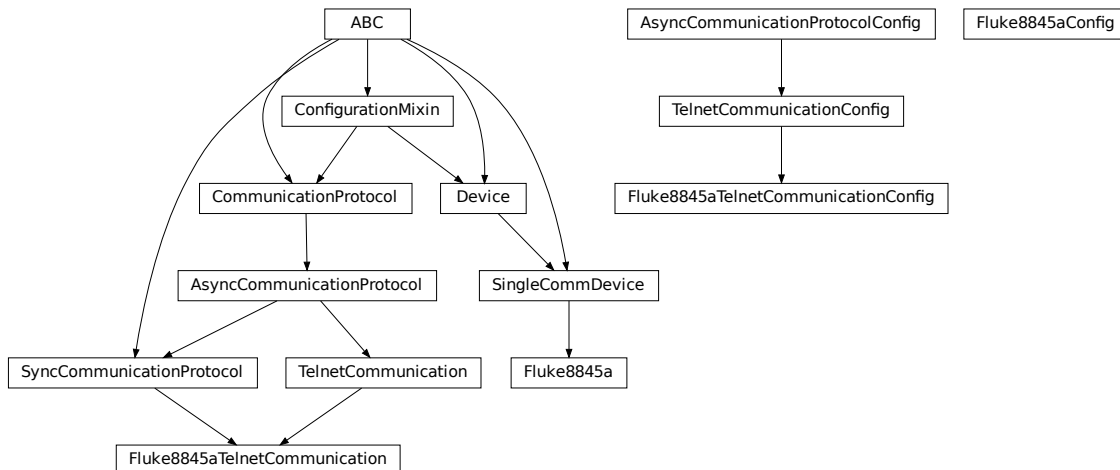
Module contents

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

hvl_ccb.dev.fluke884x

Submodules

hvl_ccb.dev.fluke884x.base



Python module for the Fluke8845a Multimeter. The communication to the device is through Telnet. 8845A/8846A Programmers Manual is available in the following link. All page numbers mentioned in this script refer to this manual. https://download.flukecal.com/pub/literature/8845A___pmeng0300.pdf

class Fluke8845a(*com, dev_config=None*)

Bases: *SingleCommDevice*

Device class to control Fluke8845a

DISPLAY_MAX_LENGTH = 12

ac_current_range

ac_voltage_range

activate_remote_mode() → *None*

Page 66

Places the Meter in the remote mode for RS-232 or Ethernet remote control. All front-panel keys, except the local key, are disabled.

clear_display_message() → *None*

Page 59

Clears the displayed message on the Meter's display.

clear_error_queue() → *None*

Page 62

Sets all bits to zero in the Meter's status byte register and all event registers. Also clears the error queue

static config_cls() → Type[*Fluke8845aConfig*]

Return the default configdataclass class.

Returns

a reference to the default configdataclass class

current_filter

dc_current_range

dc_voltage_range

static default_com_cls() → Type[*Fluke8845aTelnetCommunication*]

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

Returns

the type of the standard communication protocol for this device

property display_enable: bool

Page 59

get if the display is enabled or not fluke answer string "1" for ON and "0" for off bool(int("1")) = 1 and bool(int("0")) = 0

Returns

bool enabled = True, else False

property display_message: str

Page 59

Retrieves the text sent to the Meter's display.

fetch() → float

Page 36

Transfer stored readings to output buffer

four_wire_resistance_range

frequency_aperture

property identification: str

Page 60

Queries “*IDN?” and returns the identification string of the connected device.

Returns

the identification string of the connected device e.g. “*FLUKE, 8845A, 2540017, 08/02/10-11:53*”

initiate_trigger() → *None*

Set trigger system to wait-for-trigger

measure() → float

Page 42

Taking measurement

Once the Meter has been configured for a measurement, the INITiate command causes the Meter to take a measurement when the trigger condition have been met. To process readings from the Meter’s internal memory to the output buffer, send the Meter a FETCh? command.

property measurement_function: *MeasurementFunction*

input_function getter, query what the input function is

Raises

Fluke8845aUnknownCommandError – if the input function is unknown

period_aperture

reset() → *None*

Page 60

resets the meter to its power-up configuration

start() → *None*

Start this device as recommended by the manual

stop() → *None*

Stop this device. Disables access and closes the communication protocol.

trigger() → *None*

Causes the meter to trigger a measurement when paused

property trigger_delay: int

input_trigger_delay getter, query what the input trigger delay is in second answer format from Fluke: string, ‘+1.00000000E+00’, so convert to float and then to int

Returns

input trigger delay in second

property trigger_source: *TriggerSource*

input_trigger_source getter, query what the input trigger source is

Raises

Fluke8845aUnknownCommandError – if the input trigger source is unknown

two_wire_resistance_range

voltage_filter

class Fluke8845aConfig(name: str = 'Fluke 1')

Bases: object

Config for Fluke8845a

name: the name of the 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.

name: str = 'Fluke 1'

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 Fluke8845aTelnetCommunication(configuration)

Bases: *TelnetCommunication*, *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 interface and handle the status message. Eventually raises an error.

Parameters

command – Command to send

Raises

[*Fluke8845aError*](#) – if the connection is broken

Returns

Answer from the interface

```
class Fluke8845aTelnetCommunicationConfig(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 = 3490, timeout: Union[int,
                                         float] = 0.2)
```

Bases: [*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 = 3490

Port at which Fluke 8845a 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.

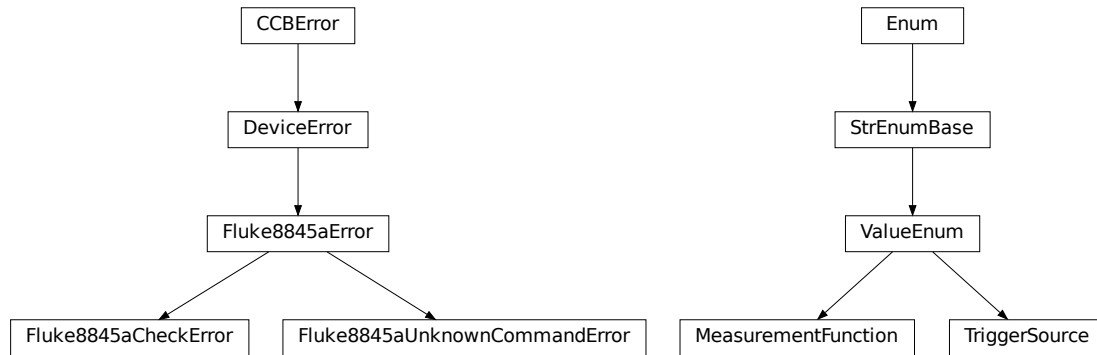
Returns

a list of strings containing all required keys.


```
terminator: bytes = b'\r'
```

The terminator is CR

hvl_ccb.dev.fluke884x.constants



Constants, ValueEnum: MeasurementFunction and TriggerSource Descriptors for range, filter and aperture

exception Fluke8845aCheckError

Bases: *Fluke8845aError*

exception Fluke8845aError

Bases: *DeviceError*

exception Fluke8845aUnknownCommandError

Bases: *Fluke8845aError*

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

Bases: *ValueEnum*

Page 40

Sets the Meter function. This command must be followed by the INIT and FETCh? commands to cause the meter to take a measurement.

CURRENT_AC = 'CURR:AC'

CURRENT_DC = 'CURR'

DIODE = 'DIOD'

FOUR_WIRE_RESISTANCE = 'FRES'

FREQUENCY = 'FREQ'

PERIOD = 'PER'

TWO_WIRE_RESISTANCE = 'RES'

VOLTAGE_AC = 'VOLT:AC'

VOLTAGE_DC = 'VOLT'

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

Bases: [ValueEnum](#)

Page 57

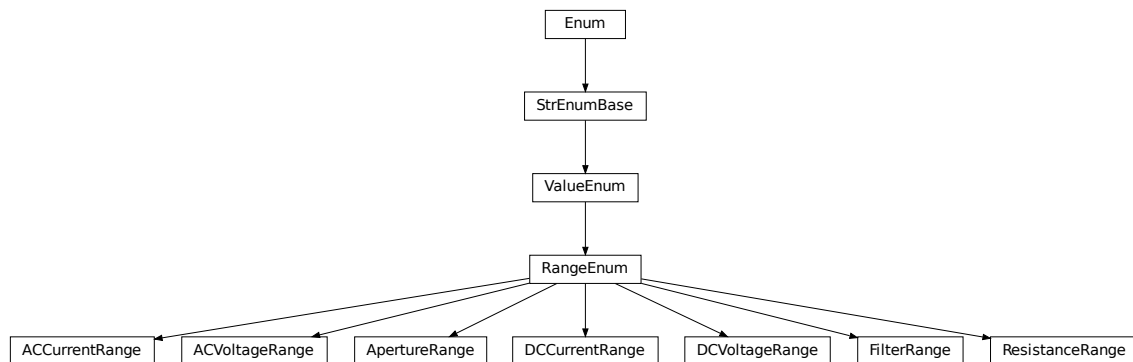
BUS: Sets the Meter to expect a trigger through the IEEE-488 bus or upon execution of a *TRG command
 IMM: Selects Meter's internal triggering system
 EXT: Sets the Meter to sense triggers through the trigger jack on the rear panel of the Meter

BUS = 'BUS'

EXTERNAL = 'EXT'

IMMEDIATE = 'IMM'

hvl_ccb.dev.fluke884x.ranges



Ranges, RangeEnum for Fluke8845a device

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

Bases: [RangeEnum](#)

possible measurement ranges for AC current with unit Ampere

FOUR_HUNDRED_MILLI_AMPERE = 0.4

ONE_AMPERE = 1.0

ONE_HUNDRED_MILLI_AMPERE = 0.1

TEN_AMPERE = 10.0

TEN_MILLI_AMPERE = 0.01

THREE_AMPERE = 3.0

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

Bases: [RangeEnum](#)

possible measurement ranges for AC voltage with unit volt

HUNDRED_VOLT = 100.0

ONE_HUNDRED_MILLI_VOLT = 0.1

ONE_VOLT = 1.0

SEVEN_HUNDRED_FIFTY_VOLT = 750.0

TEN_VOLT = 10.0

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

Bases: [RangeEnum](#)

Page 46

Sets the gate time for the frequency/period function to the value
 10ms = 4 1/2 digits
 100ms = 5 1/2 digits
 1s = 6 1/2 digits

ONE_HUNDRED_MILLI_SECOND = 0.1

ONE_SECOND = 1.0

TEN_MILLI_SECOND = 0.01

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

Bases: [RangeEnum](#)

possible measurement ranges for DC current with unit Ampere

FOUR_HUNDRED_MILLI_AMPERE = 0.4

ONE_AMPERE = 1.0

ONE_HUNDRED_MICRO_AMPERE = 0.0001

ONE_HUNDRED_MILLI_AMPERE = 0.1

ONE_MILLI_AMPERE = 0.001

TEN_AMPERE = 10.0

TEN_MILLI_AMPERE = 0.01

THREE_AMPERE = 3.0

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

Bases: [RangeEnum](#)

possible measurement ranges for DC voltage with unit volt

```
HUNDRED_VOLT = 100.0
```

```
ONE_HUNDRED_MILLI_VOLT = 0.1
```

```
ONE_THOUSAND_VOLT = 1000.0
```

```
ONE_VOLT = 1.0
```

```
TEN_VOLT = 10.0
```

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

Bases: [RangeEnum](#)

Page 47

Sets the appropriate filter for the frequency specified by <n>

High pass filter

For ``VOLTAGE_AC``: <n> Hz to 300 kHz

For ``CURRENT_AC``: <n> Hz to 10 kHz

parameters <n> = 3 slow filter

 20 medium filter

 200 fast filter

For ``CURRENT_AC`` and ``VOLTAGE_AC``

```
FAST_FILTER = 200.0
```

```
MEDIUM_FILTER = 20.0
```

```
SLOW_FILTER = 3.0
```

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

Bases: [RangeEnum](#)

possible measurement ranges for resistance with unit Ohm

```
ONE_HUNDRED_MILLION_OHM = 100000000.0
```

```
ONE_HUNDRED_OHM = 100.0
```

```
ONE_HUNDRED_THOUSAND_OHM = 100000.0
```

```
ONE_MILLION_OHM = 1000000.0
```

```
ONE_THOUSAND_OHM = 1000.0
```

```
TEN_MILLION_OHM = 10000000.0
```

```
TEN_THOUSAND_OHM = 10000.0
```

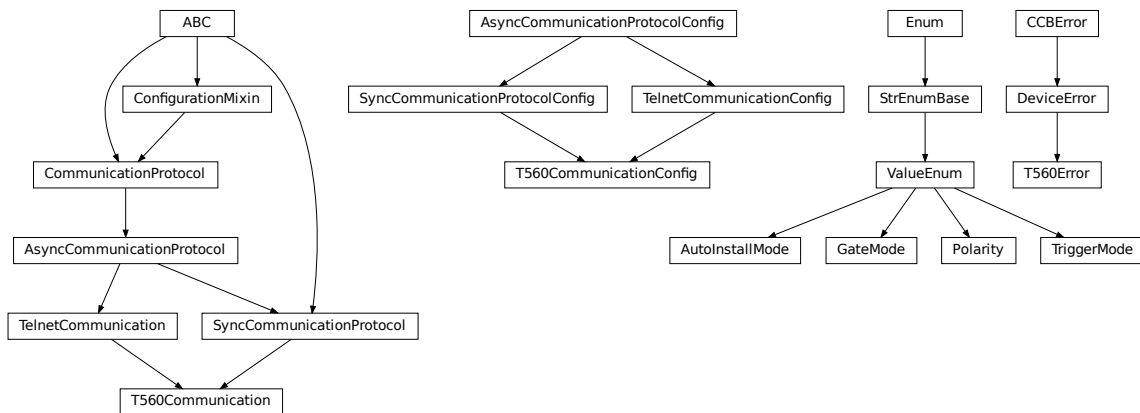
Module contents

Fluke 8845A multimeter implementation using Telnet communication

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, qualname=None, type=None, start=1,
                        boundary=None)
```

Bases: *ValueEnum*

Modes for installing configuration settings to the device.

INSTALL = 1

OFF = 0

QUEUE = 2

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

Bases: *ValueEnum*

Available T560 gate modes

INPUT = 'INP'

OFF = 'OFF'

OUTPUT = 'OUT'

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

Bases: [ValueEnum](#)

Possible channel polarity states

ACTIVE_HIGH = 'POS'

ACTIVE_LOW = 'NEG'

```
class T560Communication(configuration)
```

Bases: [SyncCommunicationProtocol](#), [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: [SyncCommunicationProtocolConfig](#), [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: *DeviceError*

T560 related errors.

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

Bases: *ValueEnum*

Available T560 trigger modes

COMMAND = 'REM'

EXT_FALLING_EDGE = 'NEG'

EXT_RISING_EDGE = 'POS'

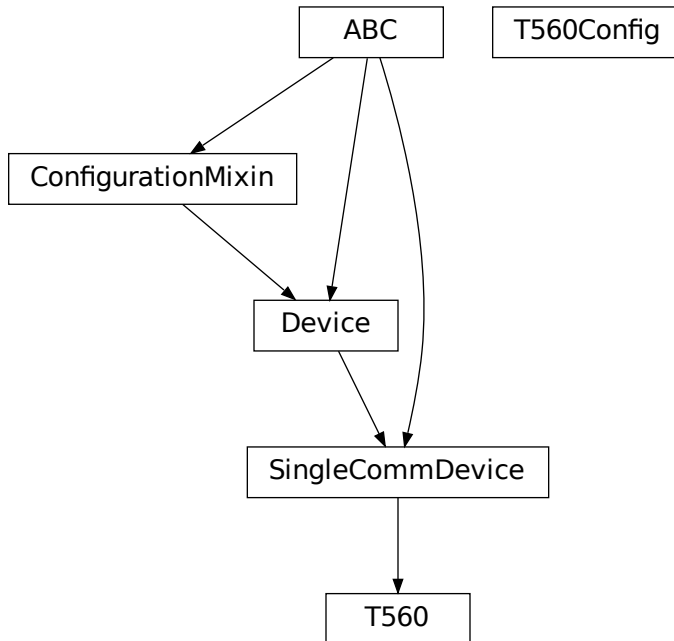
INT_SYNTHESIZER = 'SYN'

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: *SingleCommDevice*

```
activate_clock_output()
```

Outputs 10 MHz clock signal

```
property auto_install_mode: 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: _Channel
```

Channel A of T560

```
property ch_b: _Channel
```

Channel B of T560

```
property ch_c: _Channel
```

Channel C of T560

```
property ch_d: _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: GateMode

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

property gate_polarity: 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.

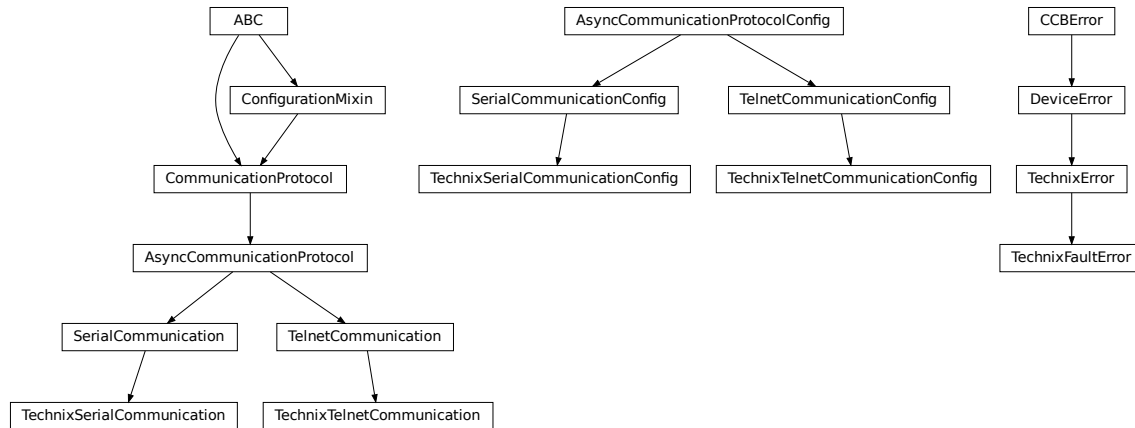
Module contents

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

Communication and auxiliary classes for Technix

exception TechnixErrorBases: *DeviceError*

Technix related errors.

exception TechnixFaultErrorBases: *TechnixError*

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

class TechnixSerialCommunication(configuration)Bases: *_TechnixCommunication*, *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,
                                       SerialCommunicationParity] =
                                       SerialCommunicationParity.NONE, stopbits: Union[int, float,
                                       SerialCommunicationStopbits] =
                                       SerialCommunicationStopbits.ONE, bytesize: Union[int,
                                       SerialCommunicationBytesize] =
                                       SerialCommunicationBytesize.EIGHTBITS, timeout: Union[int,
                                       float] = 2)
```

Bases: `_TechnixCommunicationConfig`, `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: `TelnetCommunication`, `_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, IPv4Address, IPv6Address]] = None, port:
                                     int = 4660, timeout: Union[int, float] = 0.2)
```

Bases: `_TechnixCommunicationConfig`, `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

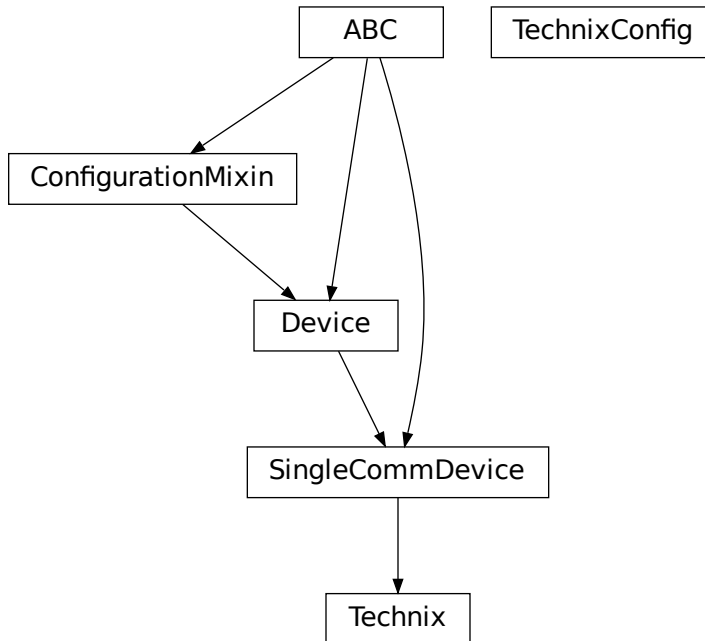
classmethod **required_keys**() → Sequence[str]

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

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

Bases: *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[TechnixSerialCommunication], Type[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?

The device class *Technix* and

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[_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[TechnixSerialCommunication],
Type[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

NOTE: `libtiepie` tries to load the `libtiepie SDK` during import. Thus, the API docs can only be generated in a system with the latter installed.

To build the API docs for this subpackage locally edit all files that match `docs/hvl_ccb.dev.tiepie.XXX.rst` and remove the `.. code-block::` directive preceding the following directives:

```
.. inheritance-diagram:: hvl_ccb.dev.tiepie.base
   :parts: 1

.. automodule:: hvl_ccb.dev.tiepie.base
   :members:
   :undoc-members:
   :show-inheritance:
```

hvl_ccb.dev.tiepie.channel

NOTE: `libtiepie` tries to load the `libtiepie SDK` during import. Thus, the API docs can only be generated in a system with the latter installed.

To build the API docs for this subpackage locally edit all files that match `docs/hvl_ccb.dev.tiepie.XXX.rst` and remove the `.. code-block::` directive preceding the following directives:

```
.. inheritance-diagram:: hvl_ccb.dev.tiepie.channel
   :parts: 1

.. automodule:: hvl_ccb.dev.tiepie.channel
   :members:
   :undoc-members:
   :show-inheritance:
```

hvl_ccb.dev.tiepie.device

NOTE: `libtiepie` tries to load the `libtiepie SDK` during import. Thus, the API docs can only be generated in a system with the latter installed.

To build the API docs for this subpackage locally edit all files that match `docs/hvl_ccb.dev.tiepie.XXX.rst` and remove the `.. code-block::` directive preceding the following directives:

```
.. inheritance-diagram:: hvl_ccb.dev.tiepie.device
   :parts: 1

.. automodule:: hvl_ccb.dev.tiepie.device
   :members:
   :undoc-members:
   :show-inheritance:
```

hvl_ccb.dev.tiepie.generator

NOTE: `libtiepie` tries to load the `libtiepie SDK` during import. Thus, the API docs can only be generated in a system with the latter installed.

To build the API docs for this subpackage locally edit all files that match `docs/hvl_ccb.dev.tiepie.XXX.rst` and remove the `.. code-block::` directive preceding the following directives:

```
.. inheritance-diagram:: hvl_ccb.dev.tiepie.generator
   :parts: 1

.. automodule:: hvl_ccb.dev.tiepie.generator
   :members:
   :undoc-members:
   :show-inheritance:
```

hvl_ccb.dev.tiepie.i2c

NOTE: `libtiepie` tries to load the `libtiepie SDK` during import. Thus, the API docs can only be generated in a system with the latter installed.

To build the API docs for this subpackage locally edit all files that match `docs/hvl_ccb.dev.tiepie.XXX.rst` and remove the `.. code-block::` directive preceding the following directives:

```
.. inheritance-diagram:: hvl_ccb.dev.tiepie.i2c
   :parts: 1

.. automodule:: hvl_ccb.dev.tiepie.i2c
   :members:
   :undoc-members:
   :show-inheritance:
```

hvl_ccb.dev.tiepie.oscilloscope

NOTE: `libtiepie` tries to load the `libtiepie SDK` during import. Thus, the API docs can only be generated in a system with the latter installed.

To build the API docs for this subpackage locally edit all files that match `docs/hvl_ccb.dev.tiepie.XXX.rst` and remove the `.. code-block::` directive preceding the following directives:

```
.. inheritance-diagram:: hvl_ccb.dev.tiepie.oscilloscope
   :parts: 1

.. automodule:: hvl_ccb.dev.tiepie.oscilloscope
   :members:
   :undoc-members:
   :show-inheritance:
```

hvl_ccb.dev.tiepie.utils

NOTE: `libtiepie` tries to load the `libtiepie SDK` during import. Thus, the API docs can only be generated in a system with the latter installed.

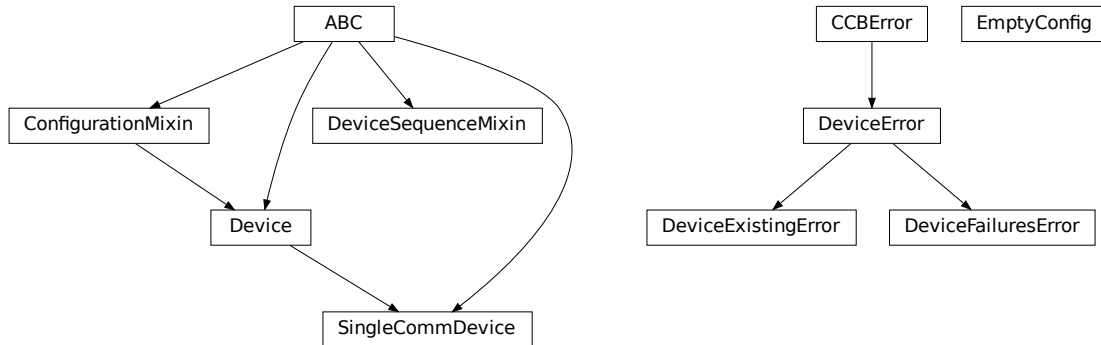
To build the API docs for this subpackage locally edit all files that match `docs/hvl_ccb.dev.tiepie.XXX.rst` and remove the `.. code-block::` directive preceding the following directives:

```
.. inheritance-diagram:: hvl_ccb.dev.tiepie.utils
   :parts: 1

.. automodule:: hvl_ccb.dev.tiepie.utils
   :members:
   :undoc-members:
   :show-inheritance:
```

Module contents

Submodules

hvl_ccb.dev.base

Module with base classes for devices.

class `Device`(*dev_config=None*)

Bases: `ConfigurationMixin`, `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 `DeviceError`

Bases: `CCBError`

exception `DeviceExistingError`

Bases: `DeviceError`

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

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

Bases: `DeviceError`

Error to indicate that one or several devices failed.

failures: `Dict[str, Exception]`

A dictionary of named devices failures (exceptions).

class DeviceSequenceMixin(*devices: Dict[str, Device]*)

Bases: 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: 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

DeviceExistingError –

devices_failed_start: Dict[str, 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, 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*) → *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, 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*) → *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

DeviceFailuresError – if one or several devices failed to start

stop() → *None*

Stop all devices in this sequence in their reverse order.

Raises

DeviceFailuresError – 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: *Device*, `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[*CommunicationProtocol*]

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

Returns

the type of the standard communication protocol for this device

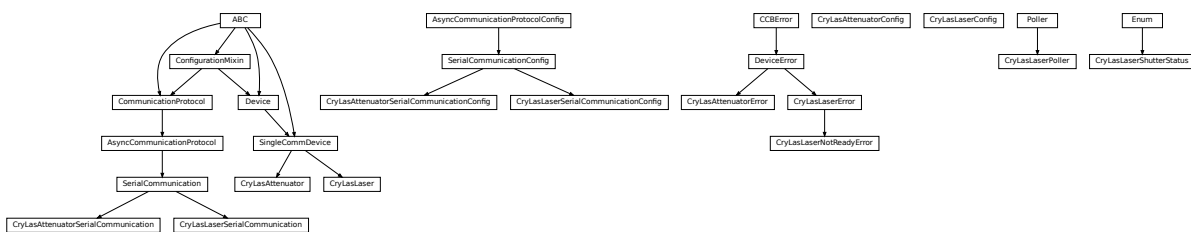
start() → *None*

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: *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]*) → *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]) → 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() → 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: [DeviceError](#)

General error with the CryLas Attenuator.

class CryLasAttenuatorSerialCommunication(configuration)

Bases: [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: [SerialCommunicationConfig](#)

baudrate: int = 9600

Baudrate for CryLas attenuator is 9600 baud

bytesize: Union[int, [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, *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, *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: *SingleCommDevice*

CryLas laser controller device class.

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

Bases: 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*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: 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, qualname=None, type=None, start=1,
                  boundary=None)

    Bases: 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, qualname=None, type=None,
                      start=1, boundary=None)

    Bases: 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 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, 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, CryLasLaserShutterStatus] =  
                        CryLasLaserShutterStatus.CLOSED)
```

Bases: object

Device configuration dataclass for the CryLas laser controller.

ShutterStatus

alias of [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, [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: [DeviceError](#)

General error with the CryLas Laser.

exception CryLasLaserNotReadyErrorBases: *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: *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: *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: **`SerialCommunicationConfig`**

`baudrate: int = 19200`

Baudrate for CryLas laser is 19200 baud

`bytesize: Union[int, 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, *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, *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*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

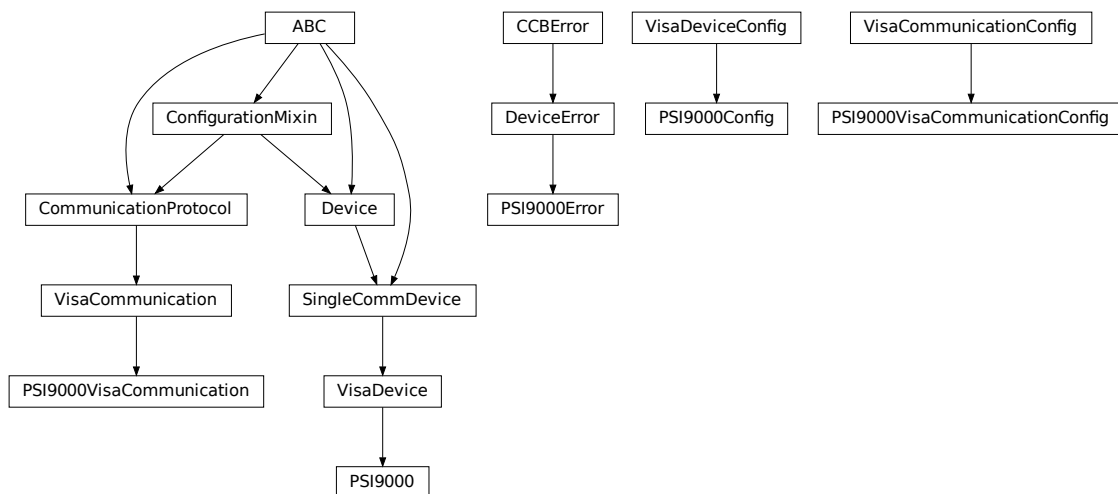
Bases: Enum

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[PSI9000VisaCommunication, PSI9000VisaCommunicationConfig, dict], dev_config: Optional[Union[PSI9000Config, dict]] = None)
```

Bases: *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: *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: *DeviceError*

Base error class regarding problems with the PSI 9000 supply.

class PSI9000VisaCommunication(configuration)

Bases: *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, IPv4Address, IPv6Address], interface_type: Union[str, *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: *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

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

Returns a list of all configdataclass fields key-names.

a list of strings containing all keys.

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

a list of strings containing all optional keys.

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

a list of strings containing all required keys.

Bases: *FuGProbusV*

property config_status: *FuGProbusVConfigRegisters*

Returns the registers for the registers with the configuration and status values

Returns

FuGProbusVConfigRegisters

property current: *FuGProbusVSetRegisters*

Returns the registers for the current output

Returns

property current_monitor: *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: *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: *FuGProbusVDORegisters*

Returns the registers for the output switch to turn the output on or off

Returns

FuGProbusVDORegisters

property outX0: *FuGProbusVDORegisters*

Returns the registers for the digital output X0

Returns

FuGProbusVDORegisters

property outX1: *FuGProbusVDORegisters*

Returns the registers for the digital output X1

Returns

FuGProbusVDORegisters

property outX2: *FuGProbusVDORegisters*

Returns the registers for the digital output X2

Returns

FuGProbusVDORegisters

property outXCMD: *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: *FuGProbusVSetRegisters*

Returns the registers for the voltage output

Returns

property voltage_monitor: *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: IntEnum

An enumeration.

NO = 0

OFF = 0

ON = 1

YES = 1

exception **FuGError**(*args, **kwargs)

Bases: *DeviceError*

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

Bases: *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: `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: `IntEnum`

An enumeration.

NEGATIVE = 1

POSITIVE = 0

class **FuGProbusIV**(*com*, *dev_config=None*)

Bases: `SingleCommDevice`, `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*: [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, *qualname*=None, *type*=None, *start*=1, *boundary*=None)

Bases: [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: [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: FuGProbusVRegisterGroups)
```

Bases: object

Configuration and Status values, acc. 4.2.5

```
property execute_on_x: FuGDigitalVal
```

status of Execute-on-X

Returns

FuGDigitalVal of the status

```
property most_recent_error: FuGErrorcodes
```

Reads the Error-Code of the most recent command

Return FuGError

Raises

[FuGError](#) – if code is not “E0”

```
property readback_data: 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`: `FuGTerminators`

Terminator character for answer strings from ADDA

Returns

FuGTerminators

class `FuGProbusVDIRegisters`(*fug*, *super_register*: `FuGProbusVRegisterGroups`)

Bases: object

Digital Inputs acc. 4.2.4

property `analog_control`: `FuGDigitalVal`

Returns

shows 1 if power supply is controlled by the analog interface

property `calibration_mode`: `FuGDigitalVal`

Returns

shows 1 if power supply is in calibration mode

property `cc_mode`: `FuGDigitalVal`

Returns

shows 1 if power supply is in CC mode

property `cv_mode`: `FuGDigitalVal`

Returns

shows 1 if power supply is in CV mode

property `digital_control`: `FuGDigitalVal`

Returns

shows 1 if power supply is digitally controlled

property `on`: `FuGDigitalVal`

Returns

shows 1 if power supply ON

property `reg_3`: `FuGDigitalVal`

For special applications.

Returns

input from bit 3-REG

property x_stat: *FuGPolarities*

Returns

polarity of HVPS with polarity reversal

class FuGProbusVDORegisters(*fug, super_register: FuGProbusVRegisterGroups*)

Bases: object

Digital outputs acc. 4.2.2

property out: *Union[int, 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: *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: FuGProbusVRegisterGroups*)

Bases: object

Analog monitors acc. 4.2.3

property adc_mode: *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, qualname=None, type=None, start=1, boundary=None*)

Bases: *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: [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`: [FuGDigitalVal](#)

Status of the high resolution mode of the output.

Return 0

normal operation

Return 1

High Res. Mode

property `rampmode`: [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`: [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: `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: `IntEnum`

An enumeration.

CURRENT = 1

FIRMWARE = 5

RATEDCURRENT = 4

RATEDVOLTAGE = 3

SN = 6

STATUSBYTE = 2

VOLTAGE = 0

class `FuGSerialCommunication`(*configuration*)

Bases: `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. Raises an error, if the answer starts with “E”.

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: [SerialCommunicationConfig](#)

baudrate: int = 9600

Baudrate for FuG power supplies is 9600 baud

bytesize: Union[int, [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, [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, 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: IntEnum

An enumeration.

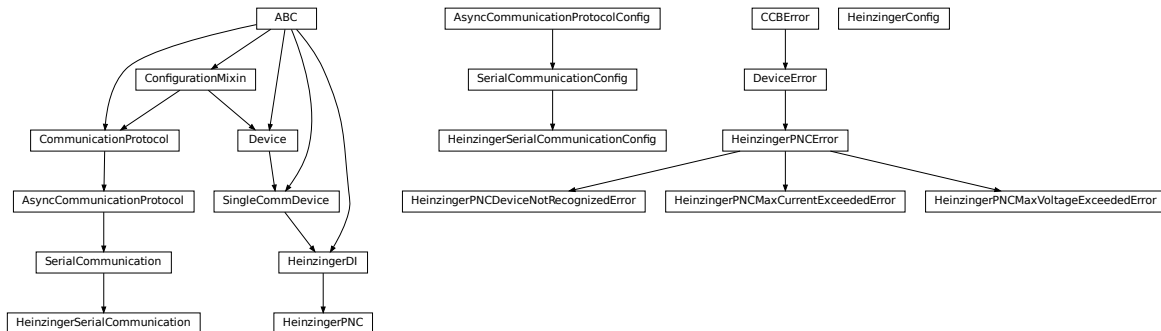
```
CR = 3
```

```
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, 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: `IntEnum`

An enumeration.

EIGHT = 8

FOUR = 4

ONE = 1

SIXTEEN = 16

TWO = 2

clean_values()

default_number_of_recordings: `Union[int, 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: `SingleCommDevice`, `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: `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: *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, 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: *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, qualname=None, type=None, start=1, boundary=None)

Bases: *AutoNumberNameEnum*

An enumeration.

A = 3

UNKNOWN = 1

mA = 2

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

Bases: *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`: `UnitCurrent`

property `unit_voltage`: `UnitVoltage`

exception `HeinzingerPNCDeviceNotRecognizedError`

Bases: `HeinzingerPNCError`

Error indicating that the serial number of the device is not recognized.

exception `HeinzingerPNCError`

Bases: `DeviceError`

General error with the Heinzinger PNC voltage source.

exception `HeinzingerPNCMaxCurrentExceededError`

Bases: `HeinzingerPNCError`

Error indicating that program attempted to set the current to a value exceeding ‘max_current’.

exception `HeinzingerPNCMaxVoltageExceededError`

Bases: `HeinzingerPNCError`

Error indicating that program attempted to set the voltage to a value exceeding ‘max_voltage’.

class `HeinzingerSerialCommunication`(*configuration*)

Bases: `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: `SerialCommunicationConfig`

baudrate: `int = 9600`

Baudrate for Heinzinger power supplies is 9600 baud

bytesize: `Union[int, 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, 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, SerialCommunicationStopbits] = 1
```

Heinzinger uses one stop bit

```
terminator: bytes = b'\n'
```

The terminator is LF

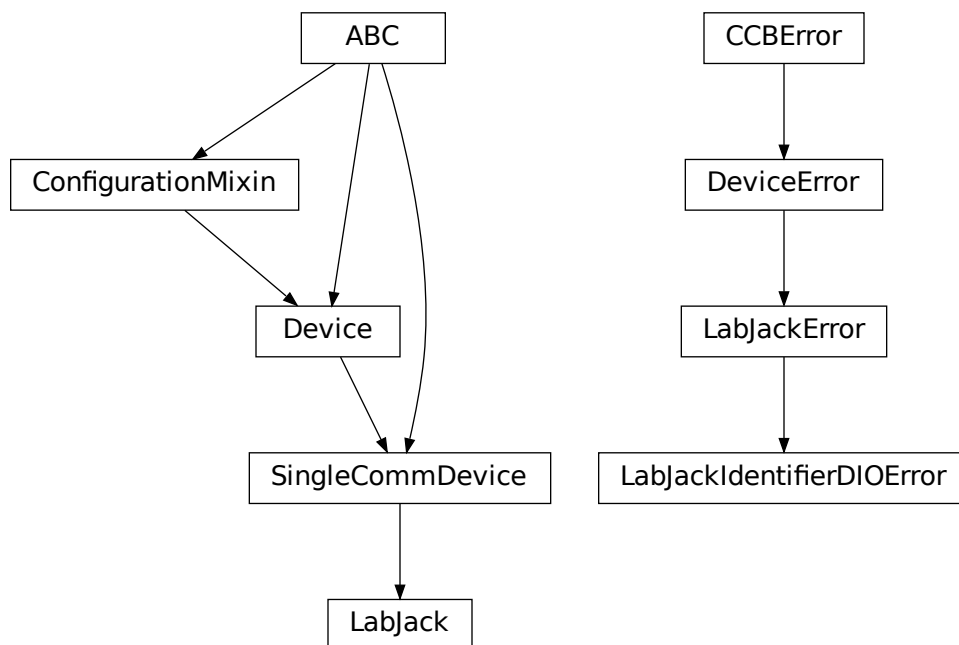
```
timeout: Union[int, float] = 3
```

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 hv1_ccb package with a labjack extra feature:

```
$ pip install "hv1_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: *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, qualname=None, type=None, start=1, boundary=None*)

Bases: *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, qualname=None, type=None, start=1, boundary=None*)

Bases: *IntEnum*

Maximum integer values for clock settings

THIRTY_TWO_BIT = 4294967295

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

Bases: *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, qualname=None, type=None, start=1, boundary=None*)

Bases: *NameEnum*

CJC slope and offset

```
internal = (1, 0)
```

```
lm34 = (55.56, 255.37)
```

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

Bases: `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 `TSeriesDIOChannel`

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

Bases: `IntEnum`

State of a digital I/O channel.

```
HIGH = 1
```

```
LOW = 0
```

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

Bases: `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[DeviceType, List[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, qualname=None, type=None,
                       start=1, boundary=None)
```

Bases: *NameEnum*

Temperature unit (to be returned)

C = 1**F** = 2**K** = 0

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

Bases: *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, 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, 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, TSeriesDIOChannel]) → *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*) → *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, 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: *DeviceError*

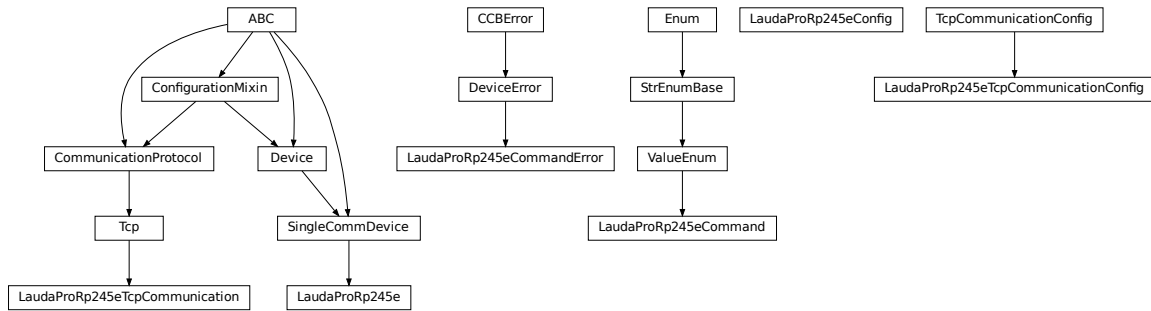
General Error for the LabJack device.

exception LabJackIdentifierDIOError

Bases: *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: `SingleCommDevice`

Lauda RP245E circulation chiller class.

static `config_cls()` → `Type[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[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, 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, qualname=None, type=None, start=1, boundary=None*)

Bases: *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 Umwaelzthermostate” 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: *DeviceError*

Error 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, OperationModeEnum] =
    OperationModeEnum.AUTO, control_mode: Union[int, ExtControlModeEnum]
    = ExtControlModeEnum.INTERNAL)
```

Bases: object

Configuration for the Lauda RP245E circulation chiller.

```
class ExtControlModeEnum(value)
```

Bases: 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: 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, 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 Galden 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, *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 Galden HT135 cooling liquid)

class LaudaProRp245eTcpCommunication(*configuration*)

Bases: *Tcp*

Implements the Communication Protocol for Lauda PRO RP245E TCP connection.

close() → *None*

Close the Lauda PRO RP245E TCP connection.

static config_cls() → Type[*LaudaProRp245eTcpCommunicationConfig*]

Return the default configdataclass class.

Returns

a reference to the default configdataclass class

open() → *None*

Open the Lauda PRO RP245E TCP connection.

Raises

LaudaProRp245eCommandError – if the connection fails.

query_command(*command*: *LaudaProRp245eCommand*, *param*: *str* = "") → *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() → *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*: *LaudaProRp245eCommand*, *param*: *str* = "") → *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*, *IPv4Address*, *IPv6Address*], *port*: *int* = 54321, *bufsize*: *int* = 1024, *wait_sec_pre_read_or_write*: Union[*int*, *float*] = 0.005, *terminator*: *str* = '\n\n')

Bases: *TcpCommunicationConfig*

Configuration dataclass for *LaudaProRp245eTcpCommunication*.

clean_values() → *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() → 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.

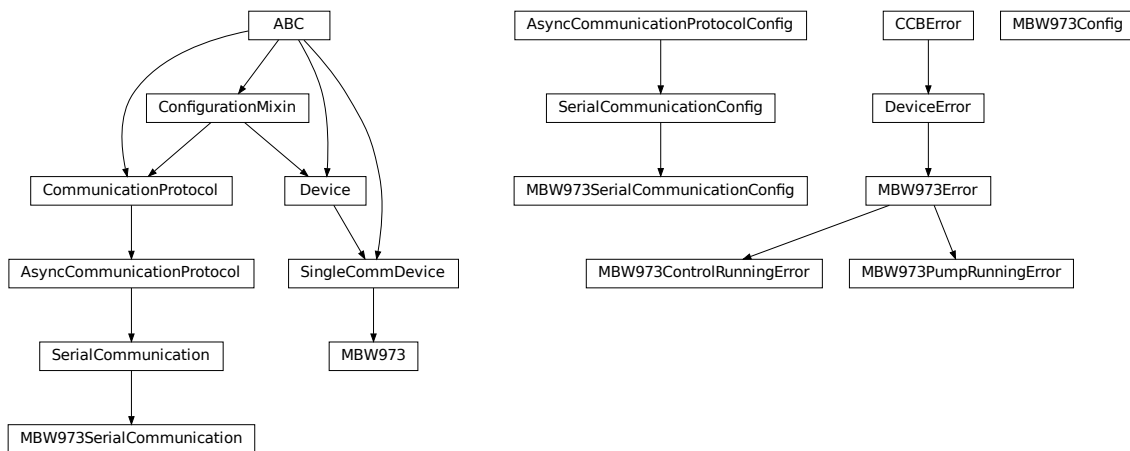
terminator: str = '\r\n'

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: *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 MBW973ControlRunningError

Bases: *MBW973Error*

Error indicating there is still a measurement running, and a new one cannot be started.

exception MBW973Error

Bases: [DeviceError](#)

General error with the MBW973 dew point mirror device.

exception MBW973PumpRunningError

Bases: [MBW973Error](#)

Error indicating the pump of the dew point mirror is still recovering gas, unable to start a new measurement.

class MBW973SerialCommunication(configuration)

Bases: [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: [SerialCommunicationConfig](#)

baudrate: int = 9600

Baudrate for MBW973 is 9600 baud

bytesize: Union[int, [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, [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, [SerialCommunicationStopbits](#)] = 1

MBW973 does use one stop bit

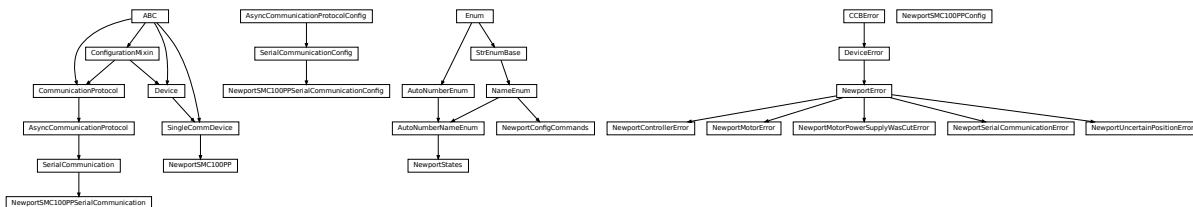
terminator: bytes = b'\r'

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, qualname=None, type=None, start=1, boundary=None)

Bases: [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: *NewportError*

Error with the Newport controller.

exception NewportError

Bases: *DeviceError*

General Error for Newport Device

exception NewportMotorError

Bases: *NewportError*

Error with the Newport motor.

exception NewportMotorPowerSupplyWasCutError

Bases: *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: *SingleCommDevice*

Device class of the Newport motor controller SMC100PP

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

Bases: 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, qualname=None, type=None,
                    start=1, boundary=None)
```

Bases: 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 *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, 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,
                             EspStageConfig] = EspStageConfig.EnableEspStageCheck)
```

Bases: object

Configuration dataclass for the Newport motor controller SMC100PP.

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

Bases: 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, qualname=None, type=None, start=1, boundary=None*)

Bases: 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, 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, [EspStageConfig](#)] = 3

user_position_offset: Union[int, float] = 23.987

velocity: Union[int, float] = 4

class `NewportSMC100PPSerialCommunication`(*configuration*)

Bases: [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*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: 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: *[SerialCommunicationConfig](#)*

baudrate: **int = 57600**

Baudrate for NewportSMC100 controller is 57600 baud

bytesize: **Union[int, [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, *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, *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: *NewportError*

Communication error with the Newport controller.

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

Bases: *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

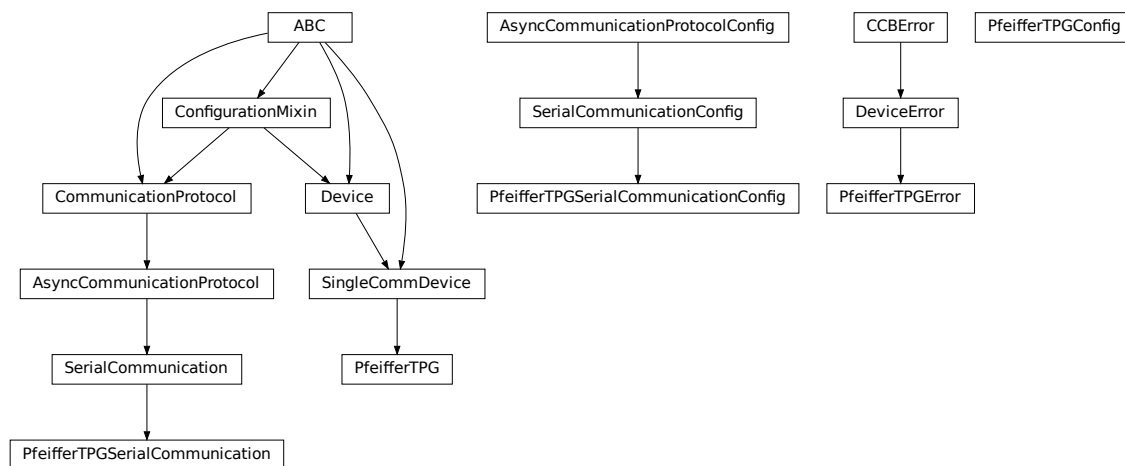
READY = 4

exception `NewportUncertainPositionError`

Bases: `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: `SingleCommDevice`

Pfeiffer TPG control unit device class

class `SensorStatus`(*value*)

Bases: `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
    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 $1 \leq \text{channel} \leq \text{number_of_sensors}$

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_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, Model] = Model.TPG25xA*)

Bases: *object*

Device configuration dataclass for Pfeiffer TPG controllers.

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

Bases: *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, 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: *DeviceError*

Error with the Pfeiffer TPG Controller.

class `PfeifferTPGSerialCommunication(configuration)`

Bases: *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: [SerialCommunicationConfig](#)

baudrate: `int = 9600`

Baudrate for Pfeiffer TPG controllers is 9600 baud

bytesize: `Union[int, 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, 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, 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

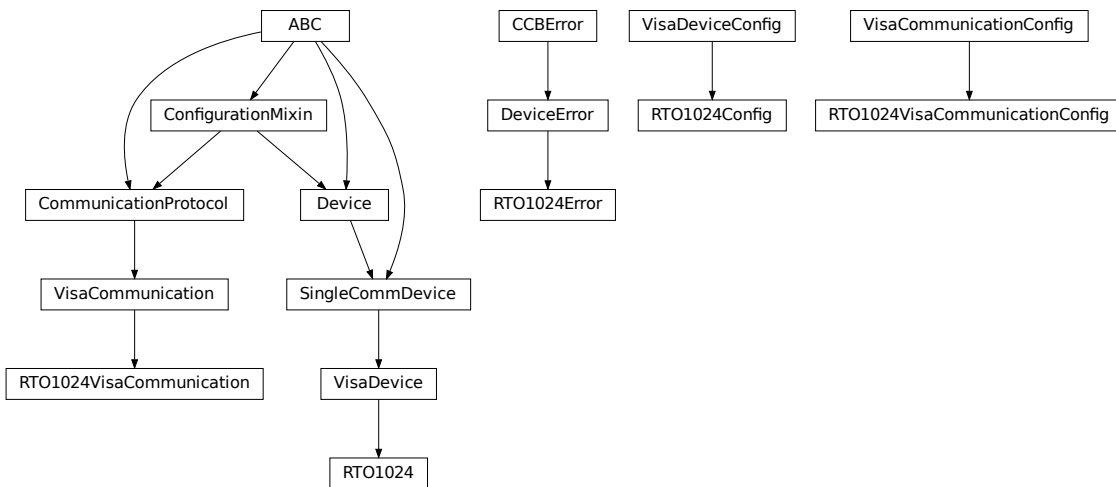
NOTE: PicoSDK Python wrappers already on import attempt to load the PicoSDK library; thus, the API docs can only be generated in a system with the latter installed and are by default disabled.

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[RTO1024VisaCommunication, RTO1024VisaCommunicationConfig, dict],
              dev_config: Union[RTO1024Config, dict])
```

Bases: [VisaDevice](#)

Device class for the Rhode & Schwarz RTO 1024 oscilloscope.

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

Bases: [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 ‘.dfl’ extension.

Information from the manual *ReCall* 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, 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 RT01024 oscilloscope and bring it into a defined state and remote mode.

stop() → *None*

Stop the RT01024 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: *VisaDeviceConfig*, *RT01024ConfigDefaultsBase*, *_RT01024ConfigBase*

Configdataclass for the RT01024 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: `DeviceError`

class `RT01024VisaCommunication(configuration)`

Bases: `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, IPv4Address, IPv6Address], interface_type: Union[str, 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: `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, `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

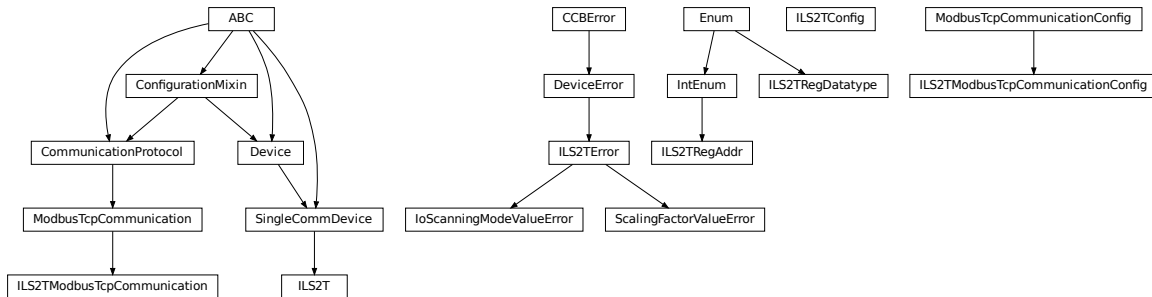
a list of strings containing all optional keys.

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: *SingleCommDevice*

Schneider Electric ILS2T stepper drive class.

```
ACTION_JOG_VALUE = 0
```

The single action value for *ILS2T.Mode.JOG*

```
class ActionsPtp(value)
```

Bases: *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: *IntEnum*

ILS2T device modes

```
JOG = 1
```

```
PTP = 3
```

```
class Ref16Jog(value)
```

Bases: *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 Adresses

alias of *ILS2TRegAddr*

RegDatatype

Modbus Register Datatypes

alias of *ILS2TRegDatatype*

class State(value)

Bases: 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 <= RPM_MAX)

Raises

ILS2TError – 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: 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: **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 **ILS2TError**

Bases: *DeviceError*

Error to indicate problems with the SE ILS2T stepper motor.

class **ILS2TModbusTcpCommunication**(*configuration*)

Bases: *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, IPv4Address, IPv6Address], unit: int = 255,*
port: int = 502)

Bases: *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: IntEnum

Modbus Register Addresses 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*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: 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: [ILS2TError](#)

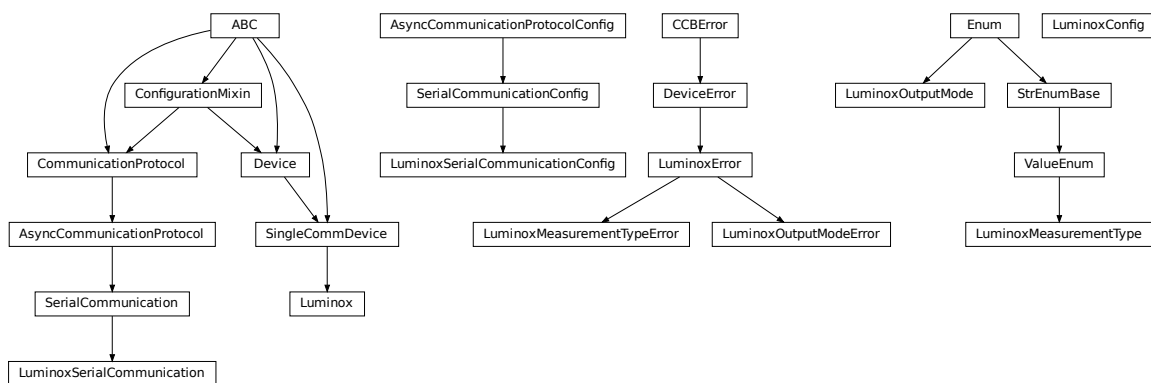
Error to indicate that the selected IO scanning mode is invalid.

exception ScalingFactorValueError

Bases: [ILS2TError](#)

Error 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: *SingleCommDevice*

Luminox oxygen sensor device class.

activate_output(*mode: 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, LuminoxMeasurementType]*) → Union[Dict[Union[str, *LuminoxMeasurementType*], Union[float, int, str]], float, int, str]

Query a value or values of Luminox 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 *LuminoxMeasurementType* is provided
- **LuminoxOutputModeError** – when polling mode is not activated
- **LuminoxMeasurementTypeError** – when expected measurement value is not read

read_streaming() → Dict[Union[str, *LuminoxMeasurementType*], Union[float, int, str]]

Read values of Luminox in the streaming mode. Convert the single string into separate values.

Returns

dictionary with *LuminoxMeasurementType.all_measurements_types()* keys and accordingly type-parsed values.

Raises

- **LuminoxOutputModeError** – when streaming mode is not activated
- **LuminoxMeasurementTypeError** – 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 LuminoxConfig(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 Luminox 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 LuminoxError

Bases: `DeviceError`

General Error for Luminox Device.

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

Bases: `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[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, 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: *LuminoxError*

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

output mode.

```
polling = 1
```

```
streaming = 0
```

exception LuminoxOutputModeError

Bases: *LuminoxError*

Wrong output mode for requested data

class `LuminoxSerialCommunication`(*configuration*)

Bases: `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: `SerialCommunicationConfig`

baudrate: `int = 9600`

Baudrate for SST Luminox is 9600 baud

bytesize: `Union[int, 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, SerialCommunicationParity] = 'N'`

SST Luminox 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, *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 / error 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()

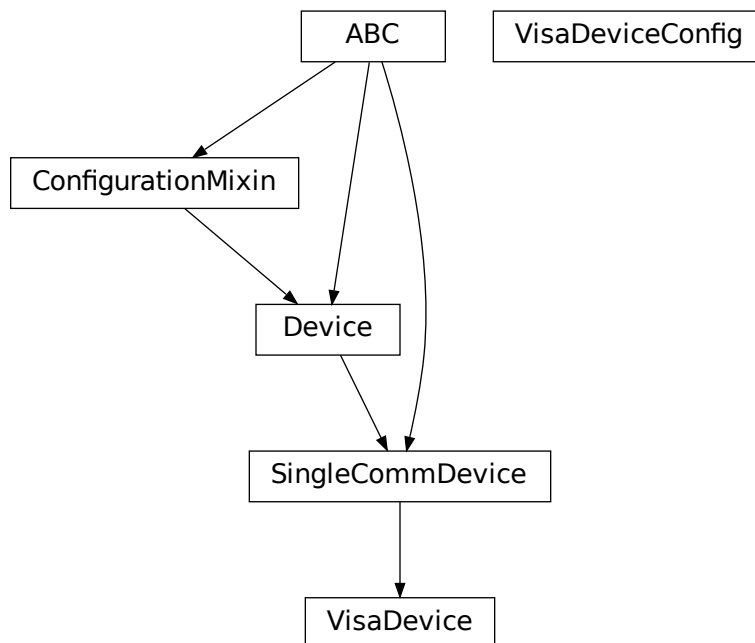
Wait for until polling function returns a result as well as any exception / error that might have been raised within a thread.

Returns

polling function result

Raises

polling function errors

hvl_ccb.dev.visa

```
class VisaDevice(com: Union[VisaCommunication, VisaCommunicationConfig, dict], dev_config: Optional[Union[VisaDeviceConfig, dict]] = None)
```

Bases: `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[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: `_VisaDeviceConfigDefaultsBase`, `_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

Subpackages

hvl_ccb.utils.conversion

Submodules

hvl_ccb.utils.conversion.map_range

MapBitAsymRange

MapBitSymRange

MapRanges

```
class MapBitAsymRange(value: ~typing.Union[int, float], bit: int, dtype_1: ~typing.Union[~numpy.dtype, None,
type, ~numpy.typing._dtype_like._SupportsDType[~numpy.dtype], str,
~typing.Tuple[~typing.Any, int], ~typing.Tuple[~typing.Any,
~typing.Union[~typing_extensions.SupportsIndex,
~typing.Sequence[~typing_extensions.SupportsIndex]]], ~typing.List[~typing.Any],
~numpy.typing._dtype_like._DTypeDict, ~typing.Tuple[~typing.Any, ~typing.Any]] =
<class 'float'>, logger=None)
```

Bases: `_MapBitRange`

Class to convert an asymmetric arbitrary range (0 to value) to a bit-range (0 to 2**bit - 1).

```
class MapBitSymRange(value: ~typing.Union[int, float], bit: int, dtype_1: ~typing.Union[~numpy.dtype, None,
~numpy.typing._dtype_like._SupportsDType[~numpy.dtype], str,
~typing.Tuple[~typing.Any, int], ~typing.Tuple[~typing.Any,
~typing.Union[~typing_extensions.SupportsIndex,
~typing.Sequence[~typing_extensions.SupportsIndex]]], ~typing.List[~typing.Any],
~numpy.typing._dtype_like._DTypeDict, ~typing.Tuple[~typing.Any, ~typing.Any]] =
<class 'float'>, logger=None)
```

Bases: `_MapBitRange`

Class to convert a symmetric arbitrary range (-value to value) to a bit-range (0 to $2^{bit} - 1$).

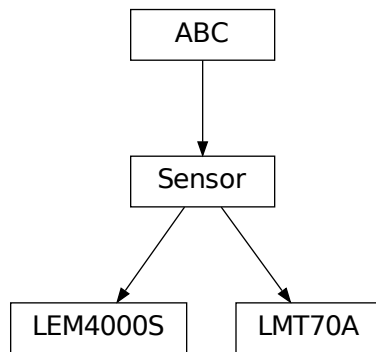
```
class MapRanges(range_1: Tuple[Union[int, float], Union[int, float]], range_2: Tuple[Union[int, float],
Union[int, float]], dtype_1: Union[dtype, None, type, _SupportsDType[dtype], str, Tuple[Any,
int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict,
Tuple[Any, Any]], dtype_2: Union[dtype, None, type, _SupportsDType[dtype], str, Tuple[Any,
int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict,
Tuple[Any, Any]], logger=None)
```

Bases: `object`

`convert_to_range1(value, **kwargs)`

`convert_to_range2(value, **kwargs)`

`hvl_ccb.utils.conversion.sensor`



Sensors that are used by the devices implemented in the CCB

```
class LEM4000S
```

Bases: `Sensor`

`CONVERSION`

`calibration_factor`

`convert(value, **kwargs)`

`shunt`

class LMT70A

Bases: *Sensor*

Converts the output voltage (V) to the measured temperature (default °C) when using a TI Precision Analog Temperature Sensor LMT70(A)

LUT

convert(value, **kwargs)

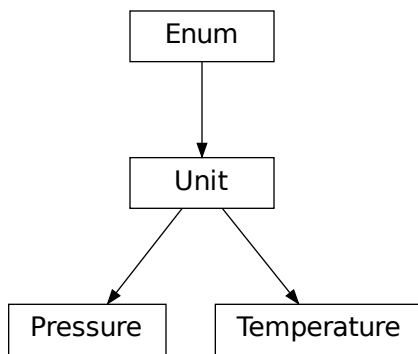
temperature_unit

class Sensor

Bases: ABC

abstract convert(value, **kwargs)

hvl_ccb.utils.conversion.unit



example Kelvin <-> Celsius

class Pressure(value)

Bases: *Unit*

An enumeration.

ATM = 'atm'

ATMOSPHERE = 'atm'

BAR = 'bar'

MILLIMETER_MERCURY = 'mmHg'

MMHG = 'mmHg'

PA = 'Pa'

PASCAL = 'Pa'

Unit conversion, within in the same group of units, for

```
POUNDS_PER_SQUARE_INCH = 'psi'
```

```
PSI = 'psi'
```

```
TORR = 'torr'
```

```
class Temperature(value)
```

```
Bases: Unit
```

```
An enumeration.
```

```
C = 'C'
```

```
CELSIUS = 'C'
```

```
F = 'F'
```

```
FAHRENHEIT = 'F'
```

```
K = 'K'
```

```
KELVIN = 'K'
```

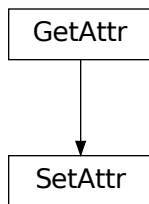
```
class Unit(value)
```

```
Bases: Enum
```

```
An enumeration.
```

```
abstract classmethod convert(value, **kwargs)
```

hvl_ccb.utils.conversion.utils



```
class GetAttr(default, name)
```

```
Bases: object
```

```
class SetAttr(default, name, limits, absolut=False, dtype=(<class 'int'>, <class 'float'>), validator=None)
```

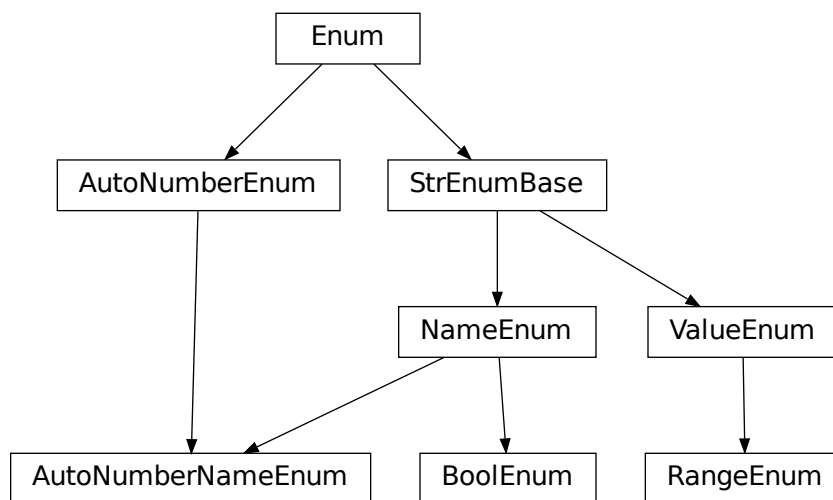
```
Bases: GetAttr
```

```
convert_value_to_str(*value: ndarray[Any, dtype[ScalarType]]) → List[Union[str, List[str]]]
```

Converts two sets of values to strings. This is necessary because a 0-dim array needs different treatment than a 1-dim array :param value: array of values either 0-dim or 1-dim :return:converted se

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!

Module contents**Submodules****hvl_ccb.utils.enum**

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

Bases: [NameEnum](#), 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, qualname=None, type=None, start=1, boundary=None*)

Bases: [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, qualname=None, type=None, start=1, boundary=None*)

Bases: [StrEnumBase](#)

Enum with names used as string representation, and with lookup and equality based on this representation. The lookup is implemented in StrEnumBase with the `_missing_value_` method. The equality is also defined at this place (`__eq__`).

Use-case:

```
class E(NameEnum):
    a = 2
    b = 4

E.a == "a"
E.a != 2
E.a != "2"
```

The access would be normally with E["a"], but E("a") works also. Therefore, E["a"] == E("a")

Attention: to avoid errors, best use together with *unique* enum decorator.

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

Bases: float, [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, qualname=None, type=None, start=1,
                 boundary=None)
```

Bases: Enum

String representation-based equality and lookup.

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

Bases: [StrEnumBase](#)

Enum with string representation of values used as string representation, and with lookup and equality based on this representation. Values do not need to be of type 'str', but they need to have a str-representation to enable this feature. The lookup is implemented in StrEnumBase with the `_missing_value_` method. The equality is also defined at this place (`__eq__`).

Use-case:

```
class E(ValueEnum):
    ONE = 1

E.ONE == "1"
E.ONE != 1
E.ONE != "ONE"
```

The access would be normally with E(1), but E("1") works also. Therefore, E(1) == E("1")

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]], 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, IPv4Address, IPv6Address]*, *logger*: *Optional[Logger] = None*)
→ *str*

validate_bool(*x_name*: *str*, *x*: *object*, *logger*: *Optional[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.

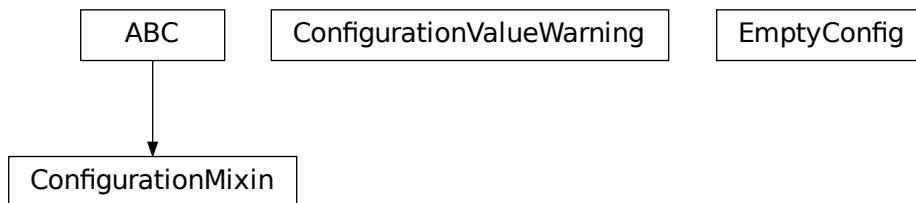
Parameters

- **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[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`

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.error

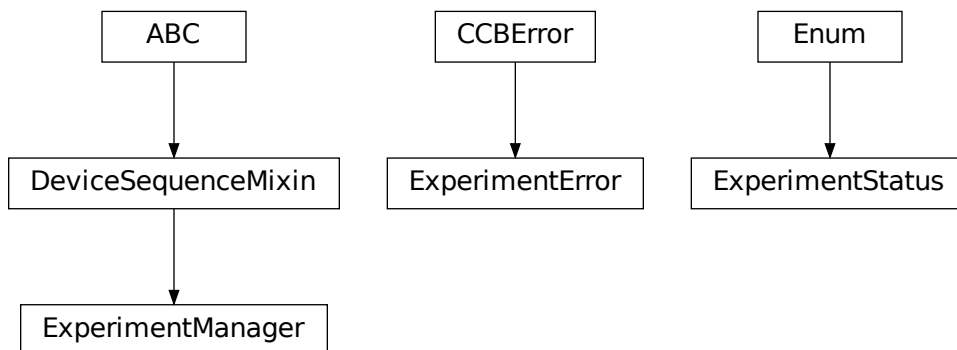
CCBError

Introduce a common code base error for the CCB

exception CCBError

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: [CCBError](#)

Error 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: *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: *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 error.

Parameters

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

Raises

ExperimentError –

devices_failed_start: Dict[str, *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, *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: *ExperimentStatus*

Get experiment status.

Returns

experiment status enum code.

stop() → *None*

Alias for ExperimentManager.finish()

class ExperimentStatus(*value*)

Bases: 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, mypy and the tests, including testing other Python versions with tox:

```
$ flake8 hvl_ccb tests
$ mypy --show-error-codes hvl_ccb
$ python setup.py test or py.test
$ tox
```

You can also use the provided make-like shell script to run flake8 and tests:

```
$ ./make.sh style
$ ./make.sh type
$ ./make.sh test
```

5. As we want to maintain a high quality of coding style we use *black*. This style is checked with the pipelines on gitlab.com. Ensure that your commits include only properly formatted code. One way to comply is to install and use *pre-commit*. This package includes the necessary configuration.
6. 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
```


7. 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 to 3.10. 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.

Make sure all your changes are committed and that all relevant MR are merged. Then switch to `devel`, update it and create `release-N.M.K` branch:

```
$ git switch devel
$ git pull
$ git checkout -b release-N.M.K
```

- Update or create entry in `HISTORY.rst` (commit message: Update HISTORY.rst: release N.M.K).
- Update, if applicable, `AUTHORS.rst` (commit message: Update AUTHORS.rst: release N.M.K)
- Update features tables in `README.rst` file (commit message: Update README.rst: release N.M.K)
- Update API docs (commit message: Update API-docs: release N.M.K)

```
$ ./make.sh docs # windows
$ make docs # unix-based-os
```

Commit all of the above, except for

- `docs/hvl_ccb.dev.picotech_pt104.rst`
- `docs/hvl_ccb.dev.tiepie.base.rst`
- `docs/hvl_ccb.dev.tiepie.channel.rst`
- `docs/hvl_ccb.dev.tiepie.device.rst`
- `docs/hvl_ccb.dev.tiepie.generator.rst`
- `docs/hvl_ccb.dev.tiepie.i2c.rst`
- `docs/hvl_ccb.dev.tiepie.oscilloscope.rst`
- `docs/hvl_ccb.dev.tiepie.utils.rst`.

Before you continue revert the changes in this file.

Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push --set-upstream origin release-N.M.K
$ 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.

The two following commands are best executed in a WSL or Unix based OS. 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
```

After this you can/should clean your folder (with WSL/Unix command):

```
$ make clean
```

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>
- Chi-Ching Hsu <hsu@eeh.ee.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>
- Chi-Ching Hsu <hsu@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>

HISTORY

7.1 0.12.0 (2022-10-17)

- Last release for Python 3.7 and 3.8
- **Repository maintenance**
 - update Labjack LJM software installer link in the pipeline
 - fix dependencies to the fixed version
 - fix `asyncua` to 0.9.95 and `pymodbus` to 2.5.3 (newer versions break the code)
 - fix PICube checker for slope as it is always positive

7.2 0.11.1 (2022-09-15)

- **Repository maintenance**
 - fix issue with `mypy` and Python 3.10.7
 - update code style to `black` 22.8.0
 - project configurations merged into `setup.cfg`
 - fix coverage indicator

7.3 0.11.0 (2022-06-22)

- New device: Fluke 884X Bench 6.5 Digit Precision Multimeter
- `RangeEnum` is a new enum for e.g. measurement ranges which also finds a suitable range object
- **smaller changes of device `tiepie`:**
 - introduce status method `is_measurement_running()` to check if the device is armed
 - introduce `stop_measurement()` to disarm the trigger of the device
 - fix bug with docs due to change of `libtiepie`
- `NameEnum` and inherited enums can only have unique entries

7.4 0.10.3 (2022-03-21)

- fix bug in the Labjack pulse feature that occurred when the start time was set to 0s
- new conversion utility to map two ranges on each other
- update CONTRIBUTING.RST
- update makefile and make.sh
- improve the mockup telnet test server

7.5 0.10.2 (2022-02-28)

- introduction of `black` as code formatter
- increase the required version of the package `aenum`
- remove device `supercube2015` - as it is no longer used
- remove unused package `openpyxl` requirement
- fix bug in highland logging
- improve handling for communication error with `picotech`

7.6 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`

7.7 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)

7.8 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`.

7.9 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.

7.10 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.

7.11 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.

7.12 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.

7.13 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.

7.14 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.

7.15 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.

7.16 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.

7.17 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.

7.18 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>=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`.

7.19 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`.

7.20 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 Luminos 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.

7.21 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).

7.22 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.

7.23 0.3.3 (2019-05-08)

- Use PyPI labjack-ljm (no external dependencies)

7.24 0.3.2 (2019-05-08)

- INSTALLATION.rst with LJMPython prerequisite info

7.25 0.3.1 (2019-05-02)

- readthedocs.org support

7.26 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)

7.27 0.2.1 (2019-04-01)

- Fix issue with LJMPython not being installed automatically with setuptools.

7.28 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.

7.29 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](#), 185
- [hvl_ccb.comm](#), 35
 - [hvl_ccb.comm.base](#), 11
 - [hvl_ccb.comm.labjack_ljm](#), 16
 - [hvl_ccb.comm.modbus_tcp](#), 19
 - [hvl_ccb.comm.opc](#), 21
 - [hvl_ccb.comm.serial](#), 24
 - [hvl_ccb.comm.tcp](#), 28
 - [hvl_ccb.comm.telnet](#), 30
 - [hvl_ccb.comm.visa](#), 32
- [hvl_ccb.configuration](#), 181
- [hvl_ccb.dev](#), 174
 - [hvl_ccb.dev.base](#), 72
 - [hvl_ccb.dev.crylas](#), 75
 - [hvl_ccb.dev.cube](#), 48
 - [hvl_ccb.dev.cube.base](#), 36
 - [hvl_ccb.dev.cube.constants](#), 41
 - [hvl_ccb.dev.cube.picube](#), 45
 - [hvl_ccb.dev.ea_psi9000](#), 85
 - [hvl_ccb.dev.fluke884x](#), 57
 - [hvl_ccb.dev.fluke884x.base](#), 48
 - [hvl_ccb.dev.fluke884x.constants](#), 53
 - [hvl_ccb.dev.fluke884x.ranges](#), 54
 - [hvl_ccb.dev.fug](#), 90
 - [hvl_ccb.dev.heinzinger](#), 103
 - [hvl_ccb.dev.highland_t560](#), 62
 - [hvl_ccb.dev.highland_t560.base](#), 57
 - [hvl_ccb.dev.highland_t560.channel](#), 59
 - [hvl_ccb.dev.highland_t560.device](#), 60
 - [hvl_ccb.dev.labjack](#), 110
 - [hvl_ccb.dev.lauda](#), 118
 - [hvl_ccb.dev.mbw973](#), 125
 - [hvl_ccb.dev.newport](#), 129
 - [hvl_ccb.dev.pfeiffer_tpg](#), 144
 - [hvl_ccb.dev.rs_rto1024](#), 150
 - [hvl_ccb.dev.se_ils2t](#), 159
 - [hvl_ccb.dev.sst_luminos](#), 166
 - [hvl_ccb.dev.technix](#), 68
 - [hvl_ccb.dev.technix.base](#), 63
 - [hvl_ccb.dev.technix.device](#), 66
 - [hvl_ccb.dev.utils](#), 171
 - [hvl_ccb.dev.visa](#), 172
 - [hvl_ccb.error](#), 183
 - [hvl_ccb.experiment_manager](#), 183
 - [hvl_ccb.utils](#), 181
 - [hvl_ccb.utils.conversion](#), 178
 - [hvl_ccb.utils.conversion.map_range](#), 174
 - [hvl_ccb.utils.conversion.sensor](#), 175
 - [hvl_ccb.utils.conversion.unit](#), 176
 - [hvl_ccb.utils.conversion.utils](#), 177
 - [hvl_ccb.utils.enum](#), 178
 - [hvl_ccb.utils.typing](#), 180
 - [hvl_ccb.utils.validation](#), 180

A

A (*HeinzingerPNC.UnitCurrent* attribute), 107
ABSOLUTE_POSITION (*ILS2T.ActionsPtp* attribute), 159
AC (*NewportConfigCommands* attribute), 129
AC_100KV (*PowerSetup* attribute), 44
AC_150KV (*PowerSetup* attribute), 44
AC_200KV (*PowerSetup* attribute), 44
AC_50KV (*PowerSetup* attribute), 44
ac_current_range (*Fluke8845a* attribute), 49
ac_voltage_range (*Fluke8845a* attribute), 49
acceleration (*NewportSMC100PPConfig* attribute), 138
ACCESS_ENABLE (*ILS2TRegAddr* attribute), 165
access_lock (*CommunicationProtocol* attribute), 14
ACCurrentRange (class in *hvl_ccb.dev.fluke884x.ranges*), 54
ACTION_JOG_VALUE (*ILS2T* attribute), 159
activate_clock_output() (*T560* method), 60
activate_measurements() (*RTO1024* method), 151
activate_output() (*Luminox* method), 167
activate_remote_mode() (*Fluke8845a* method), 49
ACTIVE (*CryLasLaser.AnswersStatus* attribute), 78
active_alarms() (*BaseCube* method), 36
ACTIVE_HIGH (*Polarity* attribute), 58
ACTIVE_LOW (*Polarity* attribute), 58
actualsetvalue (*FuGProbusVSetRegisters* property), 100
ACVoltageRange (class in *hvl_ccb.dev.fluke884x.ranges*), 55
adc_mode (*FuGProbusVMonitorRegisters* property), 99
add_device() (*DeviceSequenceMixin* method), 73
add_device() (*ExperimentManager* method), 184
ADDR_INCORRECT (*NewportSMC100PPSerialCommunication.ControllerBase* attribute), 139
address (*NewportSMC100PPConfig* attribute), 138
address (*VisaCommunicationConfig* property), 34
address() (*VisaCommunicationConfig.InterfaceType* method), 33
ADMODE (*FuGProbusIVCommands* attribute), 96
all_measurements (*LuminoxMeasurementType* attribute), 168

all_measurements_types() (*LuminoxMeasurementType* class method), 169
ANALOG (*LaudaProRp245eConfig.ExtControlModeEnum* attribute), 122
analog_control (*FuGProbusVDIRegisters* property), 98
ANY (*LabJack.DeviceType* attribute), 112
ANY (*LJMCommunicationConfig.ConnectionType* attribute), 17
ANY (*LJMCommunicationConfig.DeviceType* attribute), 17
ApertureRange (class in *hvl_ccb.dev.fluke884x.ranges*), 55
AsyncCommunicationProtocol (class in *hvl_ccb.comm.base*), 11
AsyncCommunicationProtocolConfig (class in *hvl_ccb.comm.base*), 13
ATM (*Pressure* attribute), 176
ATMOSPHERE (*Pressure* attribute), 176
attenuation (*CryLasAttenuator* property), 75
AUTO (*EarthingStickOperatingStatus* attribute), 42
AUTO (*LaudaProRp245eConfig.OperationModeEnum* attribute), 122
AUTO (*RTO1024.TriggerModes* attribute), 151
auto_install_mode (*T560* property), 60
auto_install_mode (*T560Config* attribute), 61
auto_laser_on (*CryLasLaserConfig* attribute), 82
AutoInstallMode (class in *hvl_ccb.dev.highland_t560.base*), 57
AutoNumberNameEnum (class in *hvl_ccb.utils.enum*), 178

B

BA (*NewportConfigCommands* attribute), 129
BacklashCompensation (*NewportSMC100PPConfig* attribute), 138
backup_waveform() (*RTO1024* method), 151
BAR (*Pressure* attribute), 176
barometric_pressure (*LuminoxMeasurementType* attribute), 169
base_velocity (*NewportSMC100PPConfig* attribute), 138
BaseCube (class in *hvl_ccb.dev.cube.base*), 36

BaseCubeConfiguration (class in **C**
hvl_ccb.dev.cube.base), 39

BaseCubeOpcUaCommunication (class in
hvl_ccb.dev.cube.base), 40

BaseCubeOpcUaCommunicationConfig (class in
hvl_ccb.dev.cube.base), 40

BATH_TEMP (*LaudaProRp245eCommand* attribute), 120

baudrate (*CryLasAttenuatorSerialCommunicationConfig* attribute), 77

baudrate (*CryLasLaserSerialCommunicationConfig* attribute), 84

baudrate (*FuGSerialCommunicationConfig* attribute), 102

baudrate (*HeinzingerSerialCommunicationConfig* attribute), 109

baudrate (*LuminoxSerialCommunicationConfig* attribute), 170

baudrate (*MBW973SerialCommunicationConfig* attribute), 128

baudrate (*NewportSMC100PPSerialCommunicationConfig* attribute), 142

baudrate (*PfeifferTPGSerialCommunicationConfig* attribute), 149

baudrate (*SerialCommunicationConfig* attribute), 26

BH (*NewportConfigCommands* attribute), 129

board (*VisaCommunicationConfig* attribute), 34

BoolEnum (class in *hvl_ccb.utils.enum*), 178

breakdown_detection_active (*BaseCube* property), 36

breakdown_detection_reset() (*BaseCube* method), 36

breakdown_detection_triggered (*BaseCube* property), 36

bufsize (*TcpCommunicationConfig* attribute), 29

build_str() (*LaudaProRp245eCommand* method), 121

BUS (*TriggerSource* attribute), 54

bytesize (*CryLasAttenuatorSerialCommunicationConfig* attribute), 77

bytesize (*CryLasLaserSerialCommunicationConfig* attribute), 84

bytesize (*FuGSerialCommunicationConfig* attribute), 102

bytesize (*HeinzingerSerialCommunicationConfig* attribute), 109

bytesize (*LuminoxSerialCommunicationConfig* attribute), 170

bytesize (*MBW973SerialCommunicationConfig* attribute), 128

bytesize (*NewportSMC100PPSerialCommunicationConfig* attribute), 142

bytesize (*PfeifferTPGSerialCommunicationConfig* attribute), 149

Bytesize (*SerialCommunicationConfig* attribute), 26

bytesize (*SerialCommunicationConfig* attribute), 26

C (*LabJack.TemperatureUnit* attribute), 113

C (*LabJack.ThermocoupleType* attribute), 113

C (*Temperature* attribute), 177

calibration_factor (*CryLasLaserConfig* attribute), 82

calibration_factor (*LEM4000S* attribute), 175

calibration_mode (*FuGProbusVDIRegisters* property), 98

cc_mode (*FuGProbusVDIRegisters* property), 98

CCBError, 183

cee16_socket (*BaseCube* property), 36

CELSIUS (*Temperature* attribute), 177

ch_a (*T560* property), 60

ch_b (*T560* property), 60

ch_c (*T560* property), 60

ch_d (*T560* property), 60

check_for_error() (NewportSMC100PPSerialCommunication method), 140

check_generic_type() (in module *hvl_ccb.utils.typing*), 180

check_master_slave_config() (*PSI9000* method), 86

chunk_size (*VisaCommunicationConfig* attribute), 34

clean_values() (*AsyncCommunicationProtocolConfig* method), 13

clean_values() (*BaseCubeConfiguration* method), 39

clean_values() (*CryLasAttenuatorConfig* method), 76

clean_values() (*CryLasLaserConfig* method), 82

clean_values() (*EmptyConfig* method), 74, 182

clean_values() (*Fluke8845aConfig* method), 51

clean_values() (*FuGConfig* method), 92

clean_values() (*HeinzingerConfig* method), 104

clean_values() (*ILS2TConfig* method), 163

clean_values() (*LaudaProRp245eConfig* method), 122

clean_values() (*LaudaProRp245eTcpCommunicationConfig* method), 124

clean_values() (*LJMCommunicationConfig* method), 18

clean_values() (*LuminoxConfig* method), 168

clean_values() (*MBW973Config* method), 127

clean_values() (*ModbusTcpCommunicationConfig* method), 20

clean_values() (*NewportSMC100PPConfig* method), 138

clean_values() (*OpcUaCommunicationConfig* method), 22

clean_values() (*PfeifferTPGConfig* method), 147

clean_values() (*PICubeConfiguration* method), 46

clean_values() (*PSI9000Config* method), 88

`clean_values()` (*SerialCommunicationConfig method*), 26
`clean_values()` (*T560Config method*), 61
`clean_values()` (*TcpCommunicationConfig method*), 29
`clean_values()` (*TechnixConfig method*), 67
`clean_values()` (*TelnetCommunicationConfig method*), 31
`clean_values()` (*VisaCommunicationConfig method*), 34
`clear_display_message()` (*Fluke8845a method*), 49
`clear_error_queue()` (*Fluke8845a method*), 49
`Client` (*class in hvl_ccb.comm.opc*), 21
`CLOSE` (*EarthingStickOperation attribute*), 42
`close()` (*CommunicationProtocol method*), 14
`close()` (*LaudaProRp245eTcpCommunication method*), 123
`close()` (*LJMCommunication method*), 16
`close()` (*ModbusTcpCommunication method*), 19
`close()` (*NullCommunicationProtocol method*), 14
`close()` (*OpcUaCommunication method*), 21
`close()` (*SerialCommunication method*), 24
`close()` (*Tcp method*), 28
`close()` (*TelnetCommunication method*), 30
`close()` (*VisaCommunication method*), 32
`close_shutter()` (*CryLasLaser method*), 79
`CLOSED` (*CryLasLaser.AnswersShutter attribute*), 78
`CLOSED` (*CryLasLaserShutterStatus attribute*), 85
`CLOSED` (*DoorStatus attribute*), 42
`CLOSED` (*EarthingStickStatus attribute*), 43
`CMD_EXEC_ERROR` (*New-portSMC100PPSerialCommunication.ControllerErrors attribute*), 140
`CMD_NOT_ALLOWED` (*New-portSMC100PPSerialCommunication.ControllerErrors attribute*), 140
`CMD_NOT_ALLOWED_CC` (*New-portSMC100PPSerialCommunication.ControllerErrors attribute*), 140
`CMD_NOT_ALLOWED_CONFIGURATION` (*New-portSMC100PPSerialCommunication.ControllerErrors attribute*), 140
`CMD_NOT_ALLOWED_DISABLE` (*New-portSMC100PPSerialCommunication.ControllerErrors attribute*), 140
`CMD_NOT_ALLOWED_HOMING` (*New-portSMC100PPSerialCommunication.ControllerErrors attribute*), 140
`CMD_NOT_ALLOWED_MOVING` (*New-portSMC100PPSerialCommunication.ControllerErrors attribute*), 140
`CMD_NOT_ALLOWED_NOT_REFERENCED` (*New-portSMC100PPSerialCommunication.ControllerErrors attribute*), 140
`CMD_NOT_ALLOWED_PP` (*New-portSMC100PPSerialCommunication.ControllerErrors attribute*), 140
`CMD_NOT_ALLOWED_READY` (*New-portSMC100PPSerialCommunication.ControllerErrors attribute*), 140
`CMR` (*PfeifferTPG.SensorTypes attribute*), 145
`CODE_OR_ADDR_INVALID` (*New-portSMC100PPSerialCommunication.ControllerErrors attribute*), 140
`com` (*SingleCommDevice property*), 74
`COM_TIME_OUT` (*LaudaProRp245eCommand attribute*), 120
`com_time_out` (*LaudaProRp245eConfig attribute*), 122
`COM_TIMEOUT` (*NewportSMC100PPSerialCommunication.ControllerErrors attribute*), 140
`command` (*LuminexMeasurementType property*), 169
`COMMAND` (*TriggerMode attribute*), 59
`command()` (*FuGProbusIV method*), 95
`communication_channel` (*TechnixConfig attribute*), 67
`CommunicationError`, 14
`CommunicationProtocol` (*class in hvl_ccb.comm.base*), 14
`config` (*ConfigurationMixin property*), 181
`CONFIG` (*FuGProbusVRegisterGroups attribute*), 99
`CONFIG` (*NewportSMC100PP.StateMessages attribute*), 131
`CONFIG` (*NewportStates attribute*), 143
`config_cls()` (*AsyncCommunicationProtocol static method*), 11
`config_cls()` (*BaseCube static method*), 37
`config_cls()` (*BaseCubeOpcUaCommunication static method*), 40
`config_cls()` (*ConfigurationMixin static method*), 181
`config_cls()` (*CryLasAttenuator static method*), 75
`config_cls()` (*CryLasAttenuatorSerialCommunication static method*), 77
`config_cls()` (*CryLasLaser static method*), 79
`config_cls()` (*CryLasLaserSerialCommunication static method*), 83
`config_cls()` (*Device static method*), 72
`config_cls()` (*Fluke8845a static method*), 49
`config_cls()` (*Fluke8845aTelnetCommunication static method*), 51
`config_cls()` (*FuGProbusIV static method*), 96
`config_cls()` (*FuGSerialCommunication static method*), 101
`config_cls()` (*HeinzingerDI static method*), 105
`config_cls()` (*HeinzingerSerialCommunication static method*), 108
`config_cls()` (*ILS2T static method*), 160
`config_cls()` (*ILS2TModbusTcpCommunication static method*), 164
`config_cls()` (*LaudaProRp245e static method*), 118

`config_cls()` (*LaudaProRp245eTcpCommunication static method*), 123
`config_cls()` (*LJMCommunication static method*), 16
`config_cls()` (*Luminos static method*), 167
`config_cls()` (*LuminosSerialCommunication static method*), 170
`config_cls()` (*MBW973 static method*), 125
`config_cls()` (*MBW973SerialCommunication static method*), 128
`config_cls()` (*ModbusTcpCommunication static method*), 19
`config_cls()` (*NewportSMC100PP static method*), 132
`config_cls()` (*NewportSMC100PPSerialCommunication static method*), 140
`config_cls()` (*NullCommunicationProtocol static method*), 15
`config_cls()` (*OpCuaCommunication static method*), 21
`config_cls()` (*PfeifferTPG static method*), 145
`config_cls()` (*PfeifferTPGSerialCommunication static method*), 148
`config_cls()` (*PICube static method*), 45
`config_cls()` (*PICubeOpCuaCommunication static method*), 47
`config_cls()` (*PSI9000 static method*), 86
`config_cls()` (*PSI9000VisaCommunication static method*), 89
`config_cls()` (*RTO1024 static method*), 151
`config_cls()` (*RTO1024VisaCommunication static method*), 158
`config_cls()` (*SerialCommunication static method*), 25
`config_cls()` (*SyncCommunicationProtocol static method*), 15
`config_cls()` (*T560 static method*), 60
`config_cls()` (*T560Communication static method*), 58
`config_cls()` (*Tcp static method*), 28
`config_cls()` (*Technix static method*), 66
`config_cls()` (*TechnixSerialCommunication static method*), 63
`config_cls()` (*TechnixTelnetCommunication static method*), 64
`config_cls()` (*TelnetCommunication static method*), 30
`config_cls()` (*VisaCommunication static method*), 32
`config_cls()` (*VisaDevice static method*), 172
`config_high_pulse()` (*LabJack method*), 113
`config_status` (*FuG property*), 90
`configdataclass()` (*in module hvl_ccb.configuration*), 182
`configuration_save_json()` (*ConfigurationMixin method*), 181
`ConfigurationMixin` (*class in hvl_ccb.configuration*), 181
`ConfigurationValueWarning`, 182
`connection_type` (*LJMCommunicationConfig attribute*), 18
`CONT_MODE` (*LaudaProRp245eCommand attribute*), 120
`continue_ramp()` (*LaudaProRp245e method*), 118
`control_mode` (*LaudaProRp245eConfig attribute*), 122
`CONVERSION` (*LEM4000S attribute*), 175
`convert()` (*LEM4000S method*), 175
`convert()` (*LMT70A method*), 176
`convert()` (*Sensor method*), 176
`convert()` (*Unit class method*), 177
`convert_to_range1()` (*MapRanges method*), 175
`convert_to_range2()` (*MapRanges method*), 175
`convert_value_to_str()` (*in module hvl_ccb.utils.conversion.utils*), 177
`ConvertibleTypes` (*in module hvl_ccb.utils.typing*), 180
`COOLOFF` (*LaudaProRp245eConfig.OperationModeEnum attribute*), 122
`COOLON` (*LaudaProRp245eConfig.OperationModeEnum attribute*), 122
`CR` (*FuGTerminators attribute*), 103
`create_serial_port()` (*SerialCommunicationConfig method*), 26
`create_telnet()` (*TelnetCommunicationConfig method*), 31
`CRLF` (*FuGTerminators attribute*), 103
`CryLasAttenuator` (*class in hvl_ccb.dev.crylas*), 75
`CryLasAttenuatorConfig` (*class in hvl_ccb.dev.crylas*), 76
`CryLasAttenuatorError`, 77
`CryLasAttenuatorSerialCommunication` (*class in hvl_ccb.dev.crylas*), 77
`CryLasAttenuatorSerialCommunicationConfig` (*class in hvl_ccb.dev.crylas*), 77
`CryLasLaser` (*class in hvl_ccb.dev.crylas*), 78
`CryLasLaser.AnswersShutter` (*class in hvl_ccb.dev.crylas*), 78
`CryLasLaser.AnswersStatus` (*class in hvl_ccb.dev.crylas*), 78
`CryLasLaser.LaserStatus` (*class in hvl_ccb.dev.crylas*), 79
`CryLasLaser.RepetitionRates` (*class in hvl_ccb.dev.crylas*), 79
`CryLasLaserConfig` (*class in hvl_ccb.dev.crylas*), 81
`CryLasLaserError`, 82
`CryLasLaserNotReadyError`, 82
`CryLasLaserPoller` (*class in hvl_ccb.dev.crylas*), 83
`CryLasLaserSerialCommunication` (*class in hvl_ccb.dev.crylas*), 83
`CryLasLaserSerialCommunicationConfig` (*class in hvl_ccb.dev.crylas*), 84
`CryLasLaserShutterStatus` (*class in hvl_ccb.dev.crylas*), 85
`CubeEarthingStickOperationError`, 41

- CubeError, 41
 CubeRemoteControlError, 42
 CubeStatusChangeError, 42
 CubeStopError, 42
 current (*FuG* property), 91
 CURRENT (*FuGProbusIVCommands* attribute), 96
 CURRENT (*FuGReadbackChannels* attribute), 101
 current (*Technix* property), 66
 CURRENT_AC (*MeasurementFunction* attribute), 53
 CURRENT_DC (*MeasurementFunction* attribute), 53
 current_filter (*Fluke8845a* attribute), 49
 current_lower_limit (*PSI9000Config* attribute), 88
 current_monitor (*FuG* property), 91
 current_primary (*PICube* property), 45
 current_upper_limit (*PSI9000Config* attribute), 88
 CurrentPosition (*New-portSMC100PPConfig.HomeSearch* attribute), 138
 cv_mode (*FuGProbusVDIRegisters* property), 98
- ## D
- datachange_notification() (*OpcUaSubHandler* method), 24
 date_of_manufacture (*LuminoxMeasurementType* attribute), 169
 datetime_to_opc() (*BaseCube* class method), 37
 DC_140KV (*PowerSetup* attribute), 44
 DC_280KV (*PowerSetup* attribute), 44
 dc_current_range (*Fluke8845a* attribute), 49
 dc_voltage_range (*Fluke8845a* attribute), 49
 DC_VOLTAGE_TOO_LOW (*New-portSMC100PP.MotorErrors* attribute), 130
 DCCurrentRange (class in *hvl_ccb.dev.fluke884x.ranges*), 55
 DCVoltageRange (class in *hvl_ccb.dev.fluke884x.ranges*), 55
 default_com_cls() (*BaseCube* static method), 37
 default_com_cls() (*CryLasAttenuator* static method), 75
 default_com_cls() (*CryLasLaser* static method), 79
 default_com_cls() (*Fluke8845a* static method), 49
 default_com_cls() (*FuGProbusIV* static method), 96
 default_com_cls() (*HeinzingerDI* static method), 105
 default_com_cls() (*ILS2T* static method), 160
 default_com_cls() (*LabJack* static method), 113
 default_com_cls() (*LaudaProRp245e* static method), 118
 default_com_cls() (*Luminox* static method), 167
 default_com_cls() (*MBW973* static method), 125
 default_com_cls() (*NewportSMC100PP* static method), 132
 default_com_cls() (*PfeifferTPG* static method), 145
 default_com_cls() (*PICube* static method), 45
 default_com_cls() (*PSI9000* static method), 86
 default_com_cls() (*RTO1024* static method), 151
 default_com_cls() (*SingleCommDevice* static method), 74
 default_com_cls() (*T560* static method), 61
 default_com_cls() (*Technix* method), 66
 default_com_cls() (*VisaDevice* static method), 172
 DEFAULT_IO_SCANNING_CONTROL_VALUES (*ILS2T* attribute), 159
 default_n_attempts_read_text_nonempty (*AsyncCommunicationProtocolConfig* attribute), 13
 default_n_attempts_read_text_nonempty (*FuGSerialCommunicationConfig* attribute), 102
 default_n_attempts_read_text_nonempty (*HeinzingerSerialCommunicationConfig* attribute), 109
 default_number_of_recordings (*HeinzingerConfig* attribute), 104
 Device (class in *hvl_ccb.dev.base*), 72
 DEVICE_TYPE (*LaudaProRp245eCommand* attribute), 121
 device_type (*LJMCommunicationConfig* attribute), 18
 DeviceError, 72
 DeviceExistingError, 72
 DeviceFailuresError, 72
 devices_failed_start (*DeviceSequenceMixin* attribute), 73
 devices_failed_start (*ExperimentManager* attribute), 184
 devices_failed_stop (*DeviceSequenceMixin* attribute), 73
 devices_failed_stop (*ExperimentManager* attribute), 184
 DeviceSequenceMixin (class in *hvl_ccb.dev.base*), 72
 di (*FuG* property), 91
 digital_control (*FuGProbusVDIRegisters* property), 98
 DIOChannel (*LabJack* attribute), 112
 DIODE (*MeasurementFunction* attribute), 53
 DISABLE (*NewportStates* attribute), 143
 disable() (*ILS2T* method), 160
 DISABLE_FROM_JOGGING (*New-portSMC100PP.StateMessages* attribute), 131
 DISABLE_FROM_MOVING (*New-portSMC100PP.StateMessages* attribute), 131
 DISABLE_FROM_READY (*New-portSMC100PP.StateMessages* attribute), 131
 disable_pulses() (*LabJack* method), 114
 DisableEspStageCheck (*New-portSMC100PPConfig.EspStageConfig* attribute), 137

disarm_trigger() (*T560 method*), 61
 disconnect() (*Client method*), 21
 DISPLACEMENT_OUT_OF_LIMIT (New-
 portSMC100PPSerialCommunication.ControllerErrors
 attribute), 140
 display_enable (*Fluke8845a property*), 49
 DISPLAY_MAX_LENGTH (*Fluke8845a attribute*), 49
 display_message (*Fluke8845a property*), 49
 display_message_board() (*BaseCube method*), 37
 display_status_board() (*BaseCube method*), 37
 do_ioscanning_write() (*ILS2T method*), 160
 door_1_status (*BaseCube attribute*), 37
 door_2_status (*BaseCube attribute*), 37
 door_3_status (*BaseCube attribute*), 37
 DoorStatus (class in *hvl_ccb.dev.cube.constants*), 42

E

E (*LabJack.ThermocoupleType attribute*), 113
 E0 (*FuGErrorcodes attribute*), 93
 E1 (*FuGErrorcodes attribute*), 93
 E10 (*FuGErrorcodes attribute*), 93
 E100 (*FuGErrorcodes attribute*), 93
 E106 (*FuGErrorcodes attribute*), 94
 E11 (*FuGErrorcodes attribute*), 94
 E115 (*FuGErrorcodes attribute*), 94
 E12 (*FuGErrorcodes attribute*), 94
 E125 (*FuGErrorcodes attribute*), 94
 E13 (*FuGErrorcodes attribute*), 94
 E135 (*FuGErrorcodes attribute*), 94
 E14 (*FuGErrorcodes attribute*), 94
 E145 (*FuGErrorcodes attribute*), 94
 E15 (*FuGErrorcodes attribute*), 94
 E155 (*FuGErrorcodes attribute*), 94
 E16 (*FuGErrorcodes attribute*), 94
 E165 (*FuGErrorcodes attribute*), 94
 E2 (*FuGErrorcodes attribute*), 94
 E206 (*FuGErrorcodes attribute*), 94
 E306 (*FuGErrorcodes attribute*), 94
 E4 (*FuGErrorcodes attribute*), 94
 E5 (*FuGErrorcodes attribute*), 94
 E504 (*FuGErrorcodes attribute*), 94
 E505 (*FuGErrorcodes attribute*), 94
 E6 (*FuGErrorcodes attribute*), 94
 E666 (*FuGErrorcodes attribute*), 95
 E7 (*FuGErrorcodes attribute*), 95
 E8 (*FuGErrorcodes attribute*), 95
 E9 (*FuGErrorcodes attribute*), 95
 earthing_rod_1_status (*BaseCube attribute*), 37
 earthing_rod_2_status (*BaseCube attribute*), 37
 earthing_rod_3_status (*BaseCube attribute*), 37
 EarthingRodStatus (class
 hvl_ccb.dev.cube.constants), 42
 EarthingStickOperatingStatus (class
 hvl_ccb.dev.cube.constants), 42

EarthingStickOperation (class
 hvl_ccb.dev.cube.constants), 42
 EarthingStickStatus (class
 hvl_ccb.dev.cube.constants), 43
 EEPROM_ACCESS_ERROR (New-
 portSMC100PPSerialCommunication.ControllerErrors
 attribute), 140
 EIGHT (*HeinzingerConfig.RecordingsEnum attribute*),
 104
 EIGHTBITS (*SerialCommunicationBytesize attribute*), 25
 EmptyConfig (class in *hvl_ccb.configuration*), 182
 EmptyConfig (class in *hvl_ccb.dev.base*), 74
 enable() (*ILS2T method*), 161
 enable_clock() (*LabJack method*), 114
 EnableEspStageCheck (New-
 portSMC100PPConfig.EspStageConfig at-
 tribute), 138
 encoding (*AsyncCommunicationProtocolConfig at-*
 tribute), 13
 encoding (NewportSMC100PPSerialCommunicationConfig
 attribute), 142
 encoding_error_handling (*AsyncCommunication-*
 ProtocolConfig attribute), 13
 encoding_error_handling (New-
 portSMC100PPSerialCommunicationConfig
 attribute), 142
 EndOfRunSwitch (New-
 portSMC100PPConfig.HomeSearch attribute),
 138
 EndOfRunSwitch_and_Index (New-
 portSMC100PPConfig.HomeSearch attribute),
 138
 endpoint_name (*BaseCubeOpcUaCommunicationCon-*
 fig attribute), 41
 endpoint_name (*OpcUaCommunicationConfig at-*
 tribute), 23
 endpoint_name (*PICubeOpcUaCommunicationConfig*
 attribute), 47
 ERROR (*DoorStatus attribute*), 42
 ERROR (*EarthingStickStatus attribute*), 43
 ERROR (*ExperimentStatus attribute*), 185
 ERROR (*SafetyStatus attribute*), 44
 errorcode (*FuGError attribute*), 93
 ESP_STAGE_NAME_INVALID (New-
 portSMC100PPSerialCommunication.ControllerErrors
 attribute), 140
 ETH (*LaudaProRp245eConfig.ExtControlModeEnum at-*
 tribute), 122
 ETHERNET (*LJMCommunicationConfig.ConnectionType*
 attribute), 17
 EVEN (*SerialCommunicationParity attribute*), 27
 event_notification() (*OpcUaSubHandler method*),
 24
 EXECUTE (*FuGProbusIVCommands attribute*), 96

- execute_absolute_position() (*ILS2T method*), 161
- execute_on_x (*FuGProbusVConfigRegisters property*), 97
- execute_relative_step() (*ILS2T method*), 161
- EXECUTEONX (*FuGProbusIVCommands attribute*), 96
- exit_configuration() (*NewportSMC100PP method*), 132
- exit_configuration_wait_sec (*NewportSMC100PPConfig attribute*), 138
- EXPERIMENT_BLOCKED (*EarthingRodStatus attribute*), 42
- EXPERIMENT_READY (*EarthingRodStatus attribute*), 42
- ExperimentError, 183
- ExperimentManager (class in *hvl_ccb.experiment_manager*), 183
- ExperimentStatus (class in *hvl_ccb.experiment_manager*), 185
- EXPT100 (*LaudaProRp245eConfig.ExtControlModeEnum attribute*), 122
- EXT_FALLING_EDGE (*TriggerMode attribute*), 59
- EXT_RISING_EDGE (*TriggerMode attribute*), 59
- EXTERNAL (*TriggerSource attribute*), 54
- EXTERNAL_SOURCE (*PowerSetup attribute*), 44
- EXTERNAL_TEMP (*LaudaProRp245eCommand attribute*), 121
- ## F
- F (*LabJack.TemperatureUnit attribute*), 113
- F (*Temperature attribute*), 177
- FAHRENHEIT (*Temperature attribute*), 177
- failures (*DeviceFailuresError attribute*), 72
- FAST (*ILS2T.Ref16Jog attribute*), 159
- FAST_FILTER (*FilterRange attribute*), 56
- fetch() (*Fluke8845a method*), 49
- file_copy() (*RTO1024 method*), 151
- FilterRange (class in *hvl_ccb.dev.fluke884x.ranges*), 56
- finish() (*ExperimentManager method*), 184
- FINISHED (*ExperimentStatus attribute*), 185
- FINISHING (*ExperimentStatus attribute*), 185
- fire_trigger() (*T560 method*), 61
- FIRMWARE (*FuGReadbackChannels attribute*), 101
- FIVE_MHZ (*LabJack.ClockFrequency attribute*), 112
- FIVEBITS (*SerialCommunicationBytesize attribute*), 25
- FLT_INFO (*ILS2TRegAddr attribute*), 165
- FLT_MEM_DEL (*ILS2TRegAddr attribute*), 165
- FLT_MEM_RESET (*ILS2TRegAddr attribute*), 165
- Fluke8845a (class in *hvl_ccb.dev.fluke884x.base*), 48
- Fluke8845aCheckError, 53
- Fluke8845aConfig (class in *hvl_ccb.dev.fluke884x.base*), 51
- Fluke8845aError, 53
- Fluke8845aTelnetCommunication (class in *hvl_ccb.dev.fluke884x.base*), 51
- Fluke8845aTelnetCommunicationConfig (class in *hvl_ccb.dev.fluke884x.base*), 52
- Fluke8845aUnknownCommandError, 53
- FOLLOWING_ERROR (*NewportSMC100PP.MotorErrors attribute*), 130
- FOLLOWRAMP (*FuGRampModes attribute*), 101
- force_value() (*AsyncCommunicationProtocolConfig method*), 13
- force_value() (*BaseCubeConfiguration method*), 39
- force_value() (*BaseCubeOpcUaCommunicationConfig method*), 41
- force_value() (*CryLasAttenuatorConfig method*), 76
- force_value() (*CryLasAttenuatorSerialCommunicationConfig method*), 77
- force_value() (*CryLasLaserConfig method*), 82
- force_value() (*CryLasLaserSerialCommunicationConfig method*), 84
- force_value() (*EmptyConfig method*), 74, 182
- force_value() (*Fluke8845aConfig method*), 51
- force_value() (*Fluke8845aTelnetCommunicationConfig method*), 52
- force_value() (*FuGConfig method*), 92
- force_value() (*FuGSerialCommunicationConfig method*), 102
- force_value() (*HeinzingerConfig method*), 104
- force_value() (*HeinzingerSerialCommunicationConfig method*), 109
- force_value() (*ILS2TConfig method*), 163
- force_value() (*ILS2TModbusTcpCommunicationConfig method*), 164
- force_value() (*LaudaProRp245eConfig method*), 122
- force_value() (*LaudaProRp245eTcpCommunicationConfig method*), 124
- force_value() (*LJMCommunicationConfig method*), 18
- force_value() (*LuminoxConfig method*), 168
- force_value() (*LuminoxSerialCommunicationConfig method*), 170
- force_value() (*MBW973Config method*), 127
- force_value() (*MBW973SerialCommunicationConfig method*), 128
- force_value() (*ModbusTcpCommunicationConfig method*), 20
- force_value() (*NewportSMC100PPConfig method*), 138
- force_value() (*NewportSMC100PPSerialCommunicationConfig method*), 142
- force_value() (*OpcUaCommunicationConfig method*), 23
- force_value() (*PfeifferTPGConfig method*), 147
- force_value() (*PfeifferTPGSerialCommunicationConfig method*), 149
- force_value() (*PICubeConfiguration method*), 46
- force_value() (*PICubeOpcUaCommunicationConfig method*), 47

force_value() (*PSI9000Config* method), 88
 force_value() (*PSI9000VisaCommunicationConfig* method), 89
 force_value() (*RTO1024Config* method), 157
 force_value() (*RTO1024VisaCommunicationConfig* method), 158
 force_value() (*SerialCommunicationConfig* method), 26
 force_value() (*T560CommunicationConfig* method), 58
 force_value() (*T560Config* method), 61
 force_value() (*TcpCommunicationConfig* method), 29
 force_value() (*TechnixConfig* method), 67
 force_value() (*TechnixSerialCommunicationConfig* method), 64
 force_value() (*TechnixTelnetCommunicationConfig* method), 65
 force_value() (*TelnetCommunicationConfig* method), 31
 force_value() (*VisaCommunicationConfig* method), 34
 force_value() (*VisaDeviceConfig* method), 173
 FORTY_MHZ (*LabJack.ClockFrequency* attribute), 112
 FOUR (*HeinzingerConfig.RecordingsEnum* attribute), 104
 FOUR_HUNDRED_MILLI_AMPERE (*ACCCurrentRange* attribute), 54
 FOUR_HUNDRED_MILLI_AMPERE (*DCCCurrentRange* attribute), 55
 FOUR_WIRE_RESISTANCE (*MeasurementFunction* attribute), 53
 four_wire_resistance_range (*Fluke8845a* attribute), 50
 FREERUN (*RTO1024.TriggerModes* attribute), 151
 FREQUENCY (*MeasurementFunction* attribute), 53
 frequency (*PICube* property), 45
 frequency (*T560* property), 61
 frequency_aperture (*Fluke8845a* attribute), 50
 FRM (*NewportConfigCommands* attribute), 129
 from_json() (*ConfigurationMixin* class method), 182
 FRS (*NewportConfigCommands* attribute), 129
 FuG (class in *hvl_ccb.dev.fug*), 90
 FuGConfig (class in *hvl_ccb.dev.fug*), 92
 FuGDigitalVal (class in *hvl_ccb.dev.fug*), 93
 FuGError, 93
 FuGErrorcodes (class in *hvl_ccb.dev.fug*), 93
 FuGMonitorModes (class in *hvl_ccb.dev.fug*), 95
 FuGPolarities (class in *hvl_ccb.dev.fug*), 95
 FuGProbusIV (class in *hvl_ccb.dev.fug*), 95
 FuGProbusIVCommands (class in *hvl_ccb.dev.fug*), 96
 FuGProbusV (class in *hvl_ccb.dev.fug*), 97
 FuGProbusVConfigRegisters (class in *hvl_ccb.dev.fug*), 97
 FuGProbusVDIRegisters (class in *hvl_ccb.dev.fug*), 98
 FuGProbusVDORegisters (class in *hvl_ccb.dev.fug*), 99

FuGProbusVMonitorRegisters (class in *hvl_ccb.dev.fug*), 99
 FuGProbusVRegisterGroups (class in *hvl_ccb.dev.fug*), 99
 FuGProbusVSetRegisters (class in *hvl_ccb.dev.fug*), 100
 FuGRampModes (class in *hvl_ccb.dev.fug*), 100
 FuGReadbackChannels (class in *hvl_ccb.dev.fug*), 101
 FuGSerialCommunication (class in *hvl_ccb.dev.fug*), 101
 FuGSerialCommunicationConfig (class in *hvl_ccb.dev.fug*), 102
 FuGTerminators (class in *hvl_ccb.dev.fug*), 103

G

gate_mode (*T560* property), 61
 gate_polarity (*T560* property), 61
 GateMode (class in *hvl_ccb.dev.highland_t560.base*), 57
 get_acceleration() (*NewportSMC100PP* method), 132
 get_acquire_length() (*RTO1024* method), 152
 get_ain() (*LabJack* method), 114
 get_bath_temp() (*LaudaProRp245e* method), 118
 get_by_p_id() (*LabJack.DeviceType* class method), 112
 get_by_p_id() (*LJMCommunicationConfig.DeviceType* class method), 18
 get_cal_current_source() (*LabJack* method), 114
 get_channel_offset() (*RTO1024* method), 152
 get_channel_position() (*RTO1024* method), 152
 get_channel_range() (*RTO1024* method), 152
 get_channel_scale() (*RTO1024* method), 152
 get_channel_state() (*RTO1024* method), 152
 get_clock() (*LabJack* method), 114
 get_controller_information() (*NewportSMC100PP* method), 132
 get_current() (*HeinzingerDI* method), 105
 get_dc_volt() (*ILS2T* method), 161
 get_device() (*DeviceSequenceMixin* method), 73
 get_device_type() (*LaudaProRp245e* method), 118
 get_devices() (*DeviceSequenceMixin* method), 73
 get_digital_input() (*LabJack* method), 114
 get_error_code() (*ILS2T* method), 161
 get_error_queue() (*VisaDevice* method), 173
 get_full_scale_mbar() (*PfeifferTPG* method), 145
 get_full_scale_unitless() (*PfeifferTPG* method), 145
 get_identification() (*VisaDevice* method), 173
 get_interface_version() (*HeinzingerDI* method), 105
 get_motor_configuration() (*NewportSMC100PP* method), 132
 get_move_duration() (*NewportSMC100PP* method), 133

get_negative_software_limit() (NewportSMC100PP method), 133
 get_number_of_recordings() (HeinzingerDI method), 105
 get_objects_node() (Client method), 21
 get_objects_node() (Server method), 24
 get_output() (PSI9000 method), 86
 get_position() (ILS2T method), 161
 get_position() (NewportSMC100PP method), 133
 get_positive_software_limit() (NewportSMC100PP method), 133
 get_product_id() (LabJack method), 114
 get_product_name() (LabJack method), 114
 get_product_type() (LabJack method), 115
 get_pulse_energy_and_rate() (CryLasLaser method), 79
 get_reference_point() (RTO1024 method), 152
 get_register() (FuGProbusV method), 97
 get_repetitions() (RTO1024 method), 153
 get_sbus_rh() (LabJack method), 115
 get_sbus_temp() (LabJack method), 115
 get_serial_number() (HeinzingerDI method), 105
 get_serial_number() (LabJack method), 115
 get_state() (NewportSMC100PP method), 134
 get_status() (ILS2T method), 161
 get_system_lock() (PSI9000 method), 86
 get_temperature() (ILS2T method), 161
 get_timestamps() (RTO1024 method), 153
 get_ui_lower_limits() (PSI9000 method), 86
 get_uip_upper_limits() (PSI9000 method), 86
 get_voltage() (HeinzingerDI method), 105
 get_voltage_current_setpoint() (PSI9000 method), 86
 GetAttr (class in *hvl_ccb.utils.conversion.utils*), 177
 go_home() (NewportSMC100PP method), 134
 go_to_configuration() (NewportSMC100PP method), 134
 GREEN_NOT_READY (SafetyStatus attribute), 44
 GREEN_READY (SafetyStatus attribute), 44

H

HARDWARE (CryLasLaser.RepetitionRates attribute), 79
 HEAD (CryLasLaser.AnswersStatus attribute), 78
 HeinzingerConfig (class in *hvl_ccb.dev.heinzinger*), 103
 HeinzingerConfig.RecordingsEnum (class in *hvl_ccb.dev.heinzinger*), 103
 HeinzingerDI (class in *hvl_ccb.dev.heinzinger*), 104
 HeinzingerDI.OutputStatus (class in *hvl_ccb.dev.heinzinger*), 105
 HeinzingerPNC (class in *hvl_ccb.dev.heinzinger*), 107
 HeinzingerPNC.UnitCurrent (class in *hvl_ccb.dev.heinzinger*), 107
 HeinzingerPNC.UnitVoltage (class in *hvl_ccb.dev.heinzinger*), 107
 HeinzingerPNCDeviceNotRecognizedError, 108
 HeinzingerPNCError, 108
 HeinzingerPNCMaxCurrentExceededError, 108
 HeinzingerPNCMaxVoltageExceededError, 108
 HeinzingerSerialCommunication (class in *hvl_ccb.dev.heinzinger*), 108
 HeinzingerSerialCommunicationConfig (class in *hvl_ccb.dev.heinzinger*), 108
 HIGH (LabJack.DIOStatus attribute), 112
 high_resolution (FuGProbusVSetRegisters property), 100
 home_search_polling_interval (NewportSMC100PPConfig attribute), 138
 home_search_timeout (NewportSMC100PPConfig attribute), 138
 home_search_type (NewportSMC100PPConfig attribute), 138
 home_search_velocity (NewportSMC100PPConfig attribute), 138
 HOME_STARTED (NewportSMC100PPSerialCommunication.ControllerError attribute), 140
 HomeSwitch (NewportSMC100PPConfig.HomeSearch attribute), 138
 HomeSwitch_and_Index (NewportSMC100PPConfig.HomeSearch attribute), 138
 HOMING (NewportStates attribute), 143
 HOMING_FROM_RS232 (NewportSMC100PP.StateMessages attribute), 131
 HOMING_FROM_SMC (NewportSMC100PP.StateMessages attribute), 131
 HOMING_TIMEOUT (NewportSMC100PP.MotorErrors attribute), 130
 host (ModbusTcpCommunicationConfig attribute), 20
 host (OpcUaCommunicationConfig attribute), 23
 host (TcpCommunicationConfig attribute), 29
 host (TelnetCommunicationConfig attribute), 31
 host (VisaCommunicationConfig attribute), 34
 HT (NewportConfigCommands attribute), 130
 HUNDRED_VOLT (ACVoltageRange attribute), 55
 HUNDRED_VOLT (DCVoltageRange attribute), 56
 hvl_ccb module, 185
 hvl_ccb.comm module, 35
 hvl_ccb.comm.base module, 11
 hvl_ccb.comm.labjack_ljm module, 16
 hvl_ccb.comm.modbus_tcp module, 19

<code>hvl_ccb.comm.opc</code>	<code>hvl_ccb.dev.newport</code>
module, 21	module, 129
<code>hvl_ccb.comm.serial</code>	<code>hvl_ccb.dev.pfeiffer_tpg</code>
module, 24	module, 144
<code>hvl_ccb.comm.tcp</code>	<code>hvl_ccb.dev.rs_rto1024</code>
module, 28	module, 150
<code>hvl_ccb.comm.telnet</code>	<code>hvl_ccb.dev.se_ils2t</code>
module, 30	module, 159
<code>hvl_ccb.comm.visa</code>	<code>hvl_ccb.dev.sst_luminox</code>
module, 32	module, 166
<code>hvl_ccb.configuration</code>	<code>hvl_ccb.dev.technix</code>
module, 181	module, 68
<code>hvl_ccb.dev</code>	<code>hvl_ccb.dev.technix.base</code>
module, 174	module, 63
<code>hvl_ccb.dev.base</code>	<code>hvl_ccb.dev.technix.device</code>
module, 72	module, 66
<code>hvl_ccb.dev.crylas</code>	<code>hvl_ccb.dev.utils</code>
module, 75	module, 171
<code>hvl_ccb.dev.cube</code>	<code>hvl_ccb.dev.visa</code>
module, 48	module, 172
<code>hvl_ccb.dev.cube.base</code>	<code>hvl_ccb.error</code>
module, 36	module, 183
<code>hvl_ccb.dev.cube.constants</code>	<code>hvl_ccb.experiment_manager</code>
module, 41	module, 183
<code>hvl_ccb.dev.cube.picube</code>	<code>hvl_ccb.utils</code>
module, 45	module, 181
<code>hvl_ccb.dev.ea_psi9000</code>	<code>hvl_ccb.utils.conversion</code>
module, 85	module, 178
<code>hvl_ccb.dev.fluke884x</code>	<code>hvl_ccb.utils.conversion.map_range</code>
module, 57	module, 174
<code>hvl_ccb.dev.fluke884x.base</code>	<code>hvl_ccb.utils.conversion.sensor</code>
module, 48	module, 175
<code>hvl_ccb.dev.fluke884x.constants</code>	<code>hvl_ccb.utils.conversion.unit</code>
module, 53	module, 176
<code>hvl_ccb.dev.fluke884x.ranges</code>	<code>hvl_ccb.utils.conversion.utils</code>
module, 54	module, 177
<code>hvl_ccb.dev.fug</code>	<code>hvl_ccb.utils.enum</code>
module, 90	module, 178
<code>hvl_ccb.dev.heinzinger</code>	<code>hvl_ccb.utils.typing</code>
module, 103	module, 180
<code>hvl_ccb.dev.highland_t560</code>	<code>hvl_ccb.utils.validation</code>
module, 62	module, 180
<code>hvl_ccb.dev.highland_t560.base</code>	<code>hysteresis_compensation</code>
module, 57	(<i>New-portSMC100PPConfig attribute</i>), 138
<code>hvl_ccb.dev.highland_t560.channel</code>	
module, 59	
<code>hvl_ccb.dev.highland_t560.device</code>	ID (<i>FuGProbusIVCommands attribute</i>), 96
module, 60	identification (<i>Fluke8845a property</i>), 50
<code>hvl_ccb.dev.labjack</code>	Identification_error (<i>PfeifferTPG.SensorStatus attribute</i>), 144
module, 110	identifier (<i>LJMCommunicationConfig attribute</i>), 18
<code>hvl_ccb.dev.lauda</code>	identify_device() (<i>FuG method</i>), 91
module, 118	identify_device() (<i>HeinzingerPNC method</i>), 107
<code>hvl_ccb.dev.mbw973</code>	identify_sensors() (<i>PfeifferTPG method</i>), 146
module, 125	

- IKR (*PfeifferTPG.SensorTypes* attribute), 145
 - IKR11 (*PfeifferTPG.SensorTypes* attribute), 145
 - IKR9 (*PfeifferTPG.SensorTypes* attribute), 145
 - ILS2T (class in *hvl_ccb.dev.se_ils2t*), 159
 - ILS2T.ActionsPtp (class in *hvl_ccb.dev.se_ils2t*), 159
 - ILS2T.Mode (class in *hvl_ccb.dev.se_ils2t*), 159
 - ILS2T.Ref16Jog (class in *hvl_ccb.dev.se_ils2t*), 159
 - ILS2T.State (class in *hvl_ccb.dev.se_ils2t*), 160
 - ILS2TConfig (class in *hvl_ccb.dev.se_ils2t*), 163
 - ILS2TError, 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
 - IMMEDIATE (*TriggerSource* attribute), 54
 - IMMEDIATELY (*FuGRampModes* attribute), 101
 - IMPULSE_140KV (*PowerSetup* attribute), 44
 - IMR (*PfeifferTPG.SensorTypes* attribute), 145
 - INACTIVE (*CryLasLaser.AnswersStatus* attribute), 78
 - INACTIVE (*DoorStatus* attribute), 42
 - INACTIVE (*EarthingStickStatus* attribute), 43
 - inhibit (*Technix* property), 66
 - init_attenuation (*CryLasAttenuatorConfig* attribute), 76
 - init_monitored_nodes() (*OpcUaCommunication* method), 21
 - init_shutter_status (*CryLasLaserConfig* attribute), 82
 - initialize() (*NewportSMC100PP* method), 134
 - INITIALIZED (*ExperimentStatus* attribute), 185
 - INITIALIZING (*ExperimentStatus* attribute), 185
 - INITIALIZING (*SafetyStatus* attribute), 44
 - initiate_trigger() (*Fluke8845a* method), 50
 - INPUT (*FuGProbusVRegisterGroups* attribute), 99
 - INPUT (*GateMode* attribute), 57
 - INSTALL (*AutoInstallMode* attribute), 57
 - INT32 (*ILS2TRegDatatype* attribute), 166
 - INT_SYNTHESIZER (*TriggerMode* attribute), 59
 - interface_type (*PSI9000VisaCommunicationConfig* attribute), 89
 - interface_type (*RTO1024VisaCommunicationConfig* attribute), 158
 - interface_type (*VisaCommunicationConfig* attribute), 34
 - internal (*LabJack.CjcType* attribute), 111
 - INTERNAL (*LaudaProRp245eConfig.ExtControlModeEnum* attribute), 122
 - IO_SCANNING (*ILS2TRegAddr* attribute), 165
 - IoScanningModeValueError, 166
 - is_configdataclass (*AsyncCommunicationProtocolConfig* attribute), 14
 - is_configdataclass (*BaseCubeConfiguration* attribute), 39
 - is_configdataclass (*CryLasAttenuatorConfig* attribute), 76
 - is_configdataclass (*CryLasLaserConfig* attribute), 82
 - is_configdataclass (*EmptyConfig* attribute), 74, 182
 - is_configdataclass (*Fluke8845aConfig* attribute), 51
 - is_configdataclass (*FuGConfig* attribute), 92
 - is_configdataclass (*HeinzingerConfig* attribute), 104
 - is_configdataclass (*ILS2TConfig* attribute), 163
 - is_configdataclass (*LaudaProRp245eConfig* attribute), 123
 - is_configdataclass (*LJMCommunicationConfig* attribute), 18
 - is_configdataclass (*LuminoxConfig* attribute), 168
 - is_configdataclass (*MBW973Config* attribute), 127
 - is_configdataclass (*ModbusTcpCommunicationConfig* attribute), 20
 - is_configdataclass (*NewportSMC100PPConfig* attribute), 138
 - is_configdataclass (*OpcUaCommunicationConfig* attribute), 23
 - is_configdataclass (*PfeifferTPGConfig* attribute), 147
 - is_configdataclass (*T560Config* attribute), 62
 - is_configdataclass (*TcpCommunicationConfig* attribute), 29
 - is_configdataclass (*TechnixConfig* attribute), 68
 - is_configdataclass (*VisaCommunicationConfig* attribute), 34
 - is_done() (*MBW973* method), 125
 - is_error() (*ExperimentManager* method), 184
 - is_finished() (*ExperimentManager* method), 184
 - is_generic_type_hint() (in module *hvl_ccb.utils.typing*), 180
 - is_in_range() (*ILS2TRegDatatype* method), 166
 - is_inactive (*CryLasLaser.LaserStatus* property), 79
 - is_open (*Client* property), 21
 - is_open (*LJMCommunication* property), 16
 - is_open (*OpcUaCommunication* property), 22
 - is_open (*SerialCommunication* property), 25
 - is_open (*TelnetCommunication* property), 30
 - is_polling() (*Poller* method), 171
 - is_ready (*CryLasLaser.LaserStatus* property), 79
 - is_running() (*ExperimentManager* method), 184
 - is_started (*Technix* property), 66
 - is_valid_scale_range_reversed_str() (*PfeifferTPGConfig.Model* method), 147
- ## J
- J (*LabJack.ThermocoupleType* attribute), 113
 - jerk_time (*NewportSMC100PPConfig* attribute), 138
 - JOG (*ILS2T.Mode* attribute), 159

jog_run() (*ILS2T* method), 162
 jog_stop() (*ILS2T* method), 162
 JOGGING (*NewportStates* attribute), 143
 JOGGING_FROM_DISABLE (*NewportSMC100PP.StateMessages* attribute), 131
 JOGGING_FROM_READY (*NewportSMC100PP.StateMessages* attribute), 131
 JOGN_FAST (*ILS2TRegAddr* attribute), 165
 JOGN_SLOW (*ILS2TRegAddr* attribute), 165
 JR (*NewportConfigCommands* attribute), 130

K

K (*LabJack.TemperatureUnit* attribute), 113
 K (*LabJack.ThermocoupleType* attribute), 113
 K (*Temperature* attribute), 177
 KELVIN (*Temperature* attribute), 177
 keys() (*AsyncCommunicationProtocolConfig* class method), 14
 keys() (*BaseCubeConfiguration* class method), 39
 keys() (*BaseCubeOpcUaCommunicationConfig* class method), 41
 keys() (*CryLasAttenuatorConfig* class method), 76
 keys() (*CryLasAttenuatorSerialCommunicationConfig* class method), 77
 keys() (*CryLasLaserConfig* class method), 82
 keys() (*CryLasLaserSerialCommunicationConfig* class method), 84
 keys() (*EmptyConfig* class method), 74, 182
 keys() (*Fluke8845aConfig* class method), 51
 keys() (*Fluke8845aTelnetCommunicationConfig* class method), 52
 keys() (*FuGConfig* class method), 92
 keys() (*FuGSerialCommunicationConfig* class method), 102
 keys() (*HeinzingerConfig* class method), 104
 keys() (*HeinzingerSerialCommunicationConfig* class method), 109
 keys() (*ILS2TConfig* class method), 163
 keys() (*ILS2TModbusTcpCommunicationConfig* class method), 164
 keys() (*LaudaProRp245eConfig* class method), 123
 keys() (*LaudaProRp245eTcpCommunicationConfig* class method), 124
 keys() (*LJMCommunicationConfig* class method), 18
 keys() (*LuminoxConfig* class method), 168
 keys() (*LuminoxSerialCommunicationConfig* class method), 170
 keys() (*MBW973Config* class method), 127
 keys() (*MBW973SerialCommunicationConfig* class method), 128
 keys() (*ModbusTcpCommunicationConfig* class method), 20

keys() (*NewportSMC100PPConfig* class method), 138
 keys() (*NewportSMC100PPSerialCommunicationConfig* class method), 143
 keys() (*OpcUaCommunicationConfig* class method), 23
 keys() (*PfeifferTPGConfig* class method), 147
 keys() (*PfeifferTPGSerialCommunicationConfig* class method), 149
 keys() (*PICubeConfiguration* class method), 47
 keys() (*PICubeOpcUaCommunicationConfig* class method), 47
 keys() (*PSI9000Config* class method), 88
 keys() (*PSI9000VisaCommunicationConfig* class method), 90
 keys() (*RTO1024Config* class method), 157
 keys() (*RTO1024VisaCommunicationConfig* class method), 158
 keys() (*SerialCommunicationConfig* class method), 26
 keys() (*T560CommunicationConfig* class method), 58
 keys() (*T560Config* class method), 62
 keys() (*TcpCommunicationConfig* class method), 29
 keys() (*TechnixConfig* class method), 68
 keys() (*TechnixSerialCommunicationConfig* class method), 64
 keys() (*TechnixTelnetCommunicationConfig* class method), 65
 keys() (*TelnetCommunicationConfig* class method), 31
 keys() (*VisaCommunicationConfig* class method), 34
 keys() (*VisaDeviceConfig* class method), 173
 kV (*HeinzingerPNC.UnitVoltage* attribute), 107

L

LabJack (class in *hvl_ccb.dev.labjack*), 111
 LabJack.AInRange (class in *hvl_ccb.dev.labjack*), 111
 LabJack.BitLimit (class in *hvl_ccb.dev.labjack*), 111
 LabJack.CalMicroAmpere (class in *hvl_ccb.dev.labjack*), 111
 LabJack.CjcType (class in *hvl_ccb.dev.labjack*), 111
 LabJack.ClockFrequency (class in *hvl_ccb.dev.labjack*), 112
 LabJack.DeviceType (class in *hvl_ccb.dev.labjack*), 112
 LabJack.DIOStatus (class in *hvl_ccb.dev.labjack*), 112
 LabJack.TemperatureUnit (class in *hvl_ccb.dev.labjack*), 113
 LabJack.ThermocoupleType (class in *hvl_ccb.dev.labjack*), 113
 LabJackError, 117
 LabJackIdentifierDIOError, 117
 laser_off() (*CryLasLaser* method), 80
 laser_on() (*CryLasLaser* method), 80
 LaudaProRp245e (class in *hvl_ccb.dev.lauda*), 118
 LaudaProRp245eCommand (class in *hvl_ccb.dev.lauda*), 120
 LaudaProRp245eCommandError, 122

LaudaProRp245eConfig (class in *hvl_ccb.dev.lauda*), 122
 LaudaProRp245eConfig.ExtControlModeEnum (class in *hvl_ccb.dev.lauda*), 122
 LaudaProRp245eConfig.OperationModeEnum (class in *hvl_ccb.dev.lauda*), 122
 LaudaProRp245eTcpCommunication (class in *hvl_ccb.dev.lauda*), 123
 LaudaProRp245eTcpCommunicationConfig (class in *hvl_ccb.dev.lauda*), 124
 LEM4000S (class in *hvl_ccb.utils.conversion.sensor*), 175
 LF (*FuGTerminators* attribute), 103
 LFCR (*FuGTerminators* attribute), 103
 LINE_1 (*MessageBoard* attribute), 43
 LINE_10 (*MessageBoard* attribute), 43
 LINE_11 (*MessageBoard* attribute), 43
 LINE_12 (*MessageBoard* attribute), 43
 LINE_13 (*MessageBoard* attribute), 43
 LINE_14 (*MessageBoard* attribute), 43
 LINE_15 (*MessageBoard* attribute), 43
 LINE_2 (*MessageBoard* attribute), 43
 LINE_3 (*MessageBoard* attribute), 43
 LINE_4 (*MessageBoard* attribute), 43
 LINE_5 (*MessageBoard* attribute), 43
 LINE_6 (*MessageBoard* attribute), 43
 LINE_7 (*MessageBoard* attribute), 43
 LINE_8 (*MessageBoard* attribute), 43
 LINE_9 (*MessageBoard* attribute), 43
 list_directory() (*RTO1024* method), 153
 LJMCommunication (class in *hvl_ccb.comm.labjack_ljm*), 16
 LJMCommunicationConfig (class in *hvl_ccb.comm.labjack_ljm*), 17
 LJMCommunicationConfig.ConnectionType (class in *hvl_ccb.comm.labjack_ljm*), 17
 LJMCommunicationConfig.DeviceType (class in *hvl_ccb.comm.labjack_ljm*), 17
 LJMCommunicationError, 18
 lm34 (*LabJack.CjcType* attribute), 112
 LMT70A (class in *hvl_ccb.utils.conversion.sensor*), 175
 load_configuration() (*RTO1024* method), 153
 load_device_configuration() (*T560* method), 61
 local_display() (*RTO1024* method), 153
 LOCKED (*DoorStatus* attribute), 42
 LOW (*LabJack.DIOStatus* attribute), 112
 LOWER_TEMP (*LaudaProRp245eCommand* attribute), 121
 lower_temp (*LaudaProRp245eConfig* attribute), 123
 Luminox (class in *hvl_ccb.dev.sst_luminox*), 167
 LuminoxConfig (class in *hvl_ccb.dev.sst_luminox*), 167
 LuminoxError, 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*), 170
 LUT (*LMT70A* attribute), 176

M

mA (*HeinzingerPNC.UnitCurrent* attribute), 107
 MANUAL (*EarthingStickOperatingStatus* attribute), 42
 MapBitAsymRange (class in *hvl_ccb.utils.conversion.map_range*), 174
 MapBitSymRange (class in *hvl_ccb.utils.conversion.map_range*), 175
 MapRanges (class in *hvl_ccb.utils.conversion.map_range*), 175
 MARK (*SerialCommunicationParity* attribute), 27
 max_current (*FuG* property), 91
 max_current (*HeinzingerPNC* property), 107
 max_current (*Technix* property), 66
 max_current (*TechnixConfig* attribute), 68
 max_current_hardware (*FuG* property), 91
 max_current_hardware (*HeinzingerPNC* property), 107
 max_pr_number (*LaudaProRp245eConfig* attribute), 123
 max_pump_level (*LaudaProRp245eConfig* attribute), 123
 max_timeout_retry_nr (*OpCuaCommunicationConfig* attribute), 23
 max_voltage (*FuG* property), 91
 max_voltage (*HeinzingerPNC* property), 107
 max_voltage (*Technix* property), 67
 max_voltage (*TechnixConfig* attribute), 68
 max_voltage_hardware (*FuG* property), 91
 max_voltage_hardware (*HeinzingerPNC* property), 107
 MAXIMUM (*LabJack.ClockFrequency* attribute), 112
 MBW973 (class in *hvl_ccb.dev.mbw973*), 125
 MBW973Config (class in *hvl_ccb.dev.mbw973*), 127
 MBW973ControlRunningError, 127
 MBW973Error, 127
 MBW973PumpRunningError, 128
 MBW973SerialCommunication (class in *hvl_ccb.dev.mbw973*), 128
 MBW973SerialCommunicationConfig (class in *hvl_ccb.dev.mbw973*), 128
 measure() (*Fluke8845a* method), 50
 measure() (*PfeifferTPG* method), 146
 measure_all() (*PfeifferTPG* method), 146

`measure_current()` (*HeinzingerDI method*), 106
`measure_voltage()` (*HeinzingerDI method*), 106
`measure_voltage_current()` (*PSI9000 method*), 87
`measurement_function` (*Fluke8845a property*), 50
`MeasurementFunction` (class in *hvl_ccb.dev.fluke884x.constants*), 53
`MEDIUM_FILTER` (*FilterRange attribute*), 56
`MessageBoard` (class in *hvl_ccb.dev.cube.constants*), 43
`micro_step_per_full_step_factor` (*NewportSMC100PPConfig attribute*), 139
`MILLIMETER_MERCURY` (*Pressure attribute*), 176
`MINIMUM` (*LabJack.ClockFrequency attribute*), 112
`MMHG` (*Pressure attribute*), 176
`ModbusTcpCommunication` (class in *hvl_ccb.comm.modbus_tcp*), 19
`ModbusTcpCommunicationConfig` (class in *hvl_ccb.comm.modbus_tcp*), 20
`ModbusTcpConnectionFailedError`, 21
`model` (*PfeifferTPGConfig attribute*), 147
`module`
 hvl_ccb, 185
 hvl_ccb.comm, 35
 hvl_ccb.comm.base, 11
 hvl_ccb.comm.labjack_ljm, 16
 hvl_ccb.comm.modbus_tcp, 19
 hvl_ccb.comm.opc, 21
 hvl_ccb.comm.serial, 24
 hvl_ccb.comm.tcp, 28
 hvl_ccb.comm.telnet, 30
 hvl_ccb.comm.visa, 32
 hvl_ccb.configuration, 181
 hvl_ccb.dev, 174
 hvl_ccb.dev.base, 72
 hvl_ccb.dev.crylas, 75
 hvl_ccb.dev.cube, 48
 hvl_ccb.dev.cube.base, 36
 hvl_ccb.dev.cube.constants, 41
 hvl_ccb.dev.cube.picube, 45
 hvl_ccb.dev.ea_psi9000, 85
 hvl_ccb.dev.fluke884x, 57
 hvl_ccb.dev.fluke884x.base, 48
 hvl_ccb.dev.fluke884x.constants, 53
 hvl_ccb.dev.fluke884x.ranges, 54
 hvl_ccb.dev.fug, 90
 hvl_ccb.dev.heinzinger, 103
 hvl_ccb.dev.highland_t560, 62
 hvl_ccb.dev.highland_t560.base, 57
 hvl_ccb.dev.highland_t560.channel, 59
 hvl_ccb.dev.highland_t560.device, 60
 hvl_ccb.dev.labjack, 110
 hvl_ccb.dev.lauda, 118
 hvl_ccb.dev.mbw973, 125
 hvl_ccb.dev.newport, 129
 hvl_ccb.dev.pfeiffer_tpg, 144
 hvl_ccb.dev.rs_rto1024, 150
 hvl_ccb.dev.se_ils2t, 159
 hvl_ccb.dev.sst_luminos, 166
 hvl_ccb.dev.technix, 68
 hvl_ccb.dev.technix.base, 63
 hvl_ccb.dev.technix.device, 66
 hvl_ccb.dev.utils, 171
 hvl_ccb.dev.visa, 172
 hvl_ccb.error, 183
 hvl_ccb.experiment_manager, 183
 hvl_ccb.utils, 181
 hvl_ccb.utils.conversion, 178
 hvl_ccb.utils.conversion.map_range, 174
 hvl_ccb.utils.conversion.sensor, 175
 hvl_ccb.utils.conversion.unit, 176
 hvl_ccb.utils.conversion.utils, 177
 hvl_ccb.utils.enum, 178
 hvl_ccb.utils.typing, 180
 hvl_ccb.utils.validation, 180
`MONITOR_I` (*FuGProbusVRegisterGroups attribute*), 99
`MONITOR_V` (*FuGProbusVRegisterGroups attribute*), 99
`most_recent_error` (*FuGProbusVConfigRegisters property*), 97
`motion_distance_per_full_step` (*NewportSMC100PPConfig attribute*), 139
`motor_config` (*NewportSMC100PPConfig property*), 139
`move_to_absolute_position()` (*NewportSMC100PP method*), 135
`move_to_relative_position()` (*NewportSMC100PP method*), 135
`move_wait_sec` (*NewportSMC100PPConfig attribute*), 139
`MOVING` (*NewportSMC100PP.StateMessages attribute*), 131
`MOVING` (*NewportStates attribute*), 143
`MS_NOMINAL_CURRENT` (*PSI9000 attribute*), 86
`MS_NOMINAL_VOLTAGE` (*PSI9000 attribute*), 86
`MULTI_COMMANDS_MAX` (*VisaCommunication attribute*), 32
`MULTI_COMMANDS_SEPARATOR` (*VisaCommunication attribute*), 32

N

`name` (*Fluke8845aConfig attribute*), 51
`NameEnum` (class in *hvl_ccb.utils.enum*), 178
`NAMES` (*SerialCommunicationParity attribute*), 27
`names()` (*RTO1024.TriggerModes class method*), 151
`namespace_index` (*BaseCubeConfiguration attribute*), 39
`NED_END_OF_TURN` (*NewportSMC100PP.MotorErrors attribute*), 130
`NEG` (*ILS2T.Ref16Jog attribute*), 160
`NEG_FAST` (*ILS2T.Ref16Jog attribute*), 160

NEGATIVE (*FuGPolarities* attribute), 95
 NEGATIVE (*Polarity* attribute), 44
 negative_software_limit (NewportSMC100PPConfig attribute), 139
 NewportConfigCommands (class in *hvl_ccb.dev.newport*), 129
 NewportControllerError, 130
 NewportError, 130
 NewportMotorError, 130
 NewportMotorPowerSupplyWasCutError, 130
 NewportSerialCommunicationError, 143
 NewportSMC100PP (class in *hvl_ccb.dev.newport*), 130
 NewportSMC100PP.MotorErrors (class in *hvl_ccb.dev.newport*), 130
 NewportSMC100PP.StateMessages (class in *hvl_ccb.dev.newport*), 131
 NewportSMC100PPConfig (class in *hvl_ccb.dev.newport*), 137
 NewportSMC100PPConfig.EspStageConfig (class in *hvl_ccb.dev.newport*), 137
 NewportSMC100PPConfig.HomeSearch (class in *hvl_ccb.dev.newport*), 138
 NewportSMC100PPSerialCommunication (class in *hvl_ccb.dev.newport*), 139
 NewportSMC100PPSerialCommunication.ControllerErrors (class in *hvl_ccb.dev.newport*), 139
 NewportSMC100PPSerialCommunicationConfig (class in *hvl_ccb.dev.newport*), 142
 NewportStates (class in *hvl_ccb.dev.newport*), 143
 NewportUncertainPositionError, 144
 NO (*FuGDigitalVal* attribute), 93
 NO_ERROR (NewportSMC100PPSerialCommunication.ControllerErrors attribute), 140
 NO_REF (NewportStates attribute), 143
 NO_REF_ESP_STAGE_ERROR (NewportSMC100PP.StateMessages attribute), 131
 NO_REF_FROM_CONFIG (NewportSMC100PP.StateMessages attribute), 131
 NO_REF_FROM_DISABLED (NewportSMC100PP.StateMessages attribute), 131
 NO_REF_FROM_HOMING (NewportSMC100PP.StateMessages attribute), 131
 NO_REF_FROM_JOGGING (NewportSMC100PP.StateMessages attribute), 131
 NO_REF_FROM_MOVING (NewportSMC100PP.StateMessages attribute), 131
 NO_REF_FROM_READY (NewportSMC100PP.StateMessages attribute), 131
 NO_REF_FROM_RESET (NewportSMC100PP.StateMessages attribute), 131
 NO_SENSOR (*PfeifferTPG.SensorStatus* attribute), 144
 NO_SOURCE (*PowerSetup* attribute), 44
 noise_level_measurement_channel_1 (*BaseCube-Configuration* attribute), 40
 noise_level_measurement_channel_2 (*BaseCube-Configuration* attribute), 40
 noise_level_measurement_channel_3 (*BaseCube-Configuration* attribute), 40
 noise_level_measurement_channel_4 (*BaseCube-Configuration* attribute), 40
 NONE (*ILS2T.Ref16Jog* attribute), 160
 NONE (*LabJack.ThermocoupleType* attribute), 113
 None (*PfeifferTPG.SensorTypes* attribute), 145
 NONE (*SerialCommunicationParity* attribute), 27
 NORMAL (*RTO1024.TriggerModes* attribute), 151
 noSen (*PfeifferTPG.SensorTypes* attribute), 145
 noSENSOR (*PfeifferTPG.SensorTypes* attribute), 145
 nr_trials_activate (*LuminosConfig* attribute), 168
 NullCommunicationProtocol (class in *hvl_ccb.comm.base*), 14
 Number (in module *hvl_ccb.utils.typing*), 180
 number_of_decimals (*HeinzingerConfig* attribute), 104
 number_of_sensors (*PfeifferTPG* property), 146
 O
 ODD (*SerialCommunicationParity* attribute), 27
 OFF (*AutoInstallMode* attribute), 57
 OFF (*FuGDigitalVal* attribute), 93
 OFF (*GateMode* attribute), 57
 OFF (*HeinzingerDI.OutputStatus* attribute), 105
 OFF (*TriggerMode* attribute), 59
 OH (NewportConfigCommands attribute), 130
 Ok (*PfeifferTPG.SensorStatus* attribute), 144
 on (*FuG* property), 91
 ON (*FuGDigitalVal* attribute), 93
 on (*FuGProbusVDIRegisters* property), 98
 ON (*HeinzingerDI.OutputStatus* attribute), 105
 ON (*ILS2T.State* attribute), 160
 ONE (*HeinzingerConfig.RecordingsEnum* attribute), 104
 ONE (*LabJack.AInRange* attribute), 111
 ONE (*SerialCommunicationStopbits* attribute), 27
 ONE_AMPERE (*ACCurrentRange* attribute), 54
 ONE_AMPERE (*DCCurrentRange* attribute), 55
 ONE_HUNDRED_MICRO_AMPERE (*DCCurrentRange* attribute), 55
 ONE_HUNDRED_MILLI_AMPERE (*ACCurrentRange* attribute), 54
 ONE_HUNDRED_MILLI_AMPERE (*DCCurrentRange* attribute), 55

ONE_HUNDRED_MILLI_SECOND (*ApertureRange* attribute), 55
 ONE_HUNDRED_MILLI_VOLT (*ACVoltageRange* attribute), 55
 ONE_HUNDRED_MILLI_VOLT (*DCVoltageRange* attribute), 56
 ONE_HUNDRED_MILLION_OHM (*ResistanceRange* attribute), 56
 ONE_HUNDRED_OHM (*ResistanceRange* attribute), 56
 ONE_HUNDRED_THOUSAND_OHM (*ResistanceRange* attribute), 56
 ONE_HUNDREDTH (*LabJack.AInRange* attribute), 111
 ONE_MILLI_AMPERE (*DCCurrentRange* attribute), 55
 ONE_MILLION_OHM (*ResistanceRange* attribute), 56
 ONE_POINT_FIVE (*SerialCommunicationStopbits* attribute), 27
 ONE_SECOND (*ApertureRange* attribute), 55
 ONE_TENTH (*LabJack.AInRange* attribute), 111
 ONE_THOUSAND_OHM (*ResistanceRange* attribute), 56
 ONE_THOUSAND_VOLT (*DCVoltageRange* attribute), 56
 ONE_VOLT (*ACVoltageRange* attribute), 55
 ONE_VOLT (*DCVoltageRange* attribute), 56
 ONLYUPWARDSOFFTOZERO (*FuGRampModes* attribute), 101
 OPC_MAX_YEAR (*BaseCube* attribute), 36
 OPC_MIN_YEAR (*BaseCube* attribute), 36
 OpcUaCommunication (class in *hvl_ccb.comm.opc*), 21
 OpcUaCommunicationConfig (class in *hvl_ccb.comm.opc*), 22
 OpcUaCommunicationIOError, 24
 OpcUaCommunicationTimeoutError, 24
 OpcUaSubHandler (class in *hvl_ccb.comm.opc*), 24
 OPEN (*DoorStatus* attribute), 42
 OPEN (*EarthingStickOperation* attribute), 43
 OPEN (*EarthingStickStatus* attribute), 43
 open() (*CommunicationProtocol* method), 14
 open() (*LaudaProRp245eTcpCommunication* method), 124
 open() (*LJMCommunication* method), 16
 open() (*ModbusTcpCommunication* method), 19
 open() (*NullCommunicationProtocol* method), 15
 open() (*OpcUaCommunication* method), 22
 open() (*SerialCommunication* method), 25
 open() (*Tcp* method), 28
 open() (*TelnetCommunication* method), 30
 open() (*VisaCommunication* method), 33
 open_interlock (*Technix* property), 67
 open_shutter() (*CryLasLaser* method), 80
 open_timeout (*VisaCommunicationConfig* attribute), 34
 OPENED (*CryLasLaser.AnswersShutter* attribute), 78
 OPENED (*CryLasLaserShutterStatus* attribute), 85
 operate (*BaseCube* property), 37
 operate (*PICube* property), 46
 OPERATION_MODE (*LaudaProRp245eCommand* attribute), 121
 operation_mode (*LaudaProRp245eConfig* attribute), 123
 optional_defaults() (*AsyncCommunicationProtocolConfig* class method), 14
 optional_defaults() (*BaseCubeConfiguration* class method), 40
 optional_defaults() (*BaseCubeOpcUaCommunicationConfig* class method), 41
 optional_defaults() (*CryLasAttenuatorConfig* class method), 76
 optional_defaults() (*CryLasAttenuatorSerialCommunicationConfig* class method), 78
 optional_defaults() (*CryLasLaserConfig* class method), 82
 optional_defaults() (*CryLasLaserSerialCommunicationConfig* class method), 84
 optional_defaults() (*EmptyConfig* class method), 74, 182
 optional_defaults() (*Fluke8845aConfig* class method), 51
 optional_defaults() (*Fluke8845aTelnetCommunicationConfig* class method), 52
 optional_defaults() (*FuGConfig* class method), 93
 optional_defaults() (*FuGSerialCommunicationConfig* class method), 102
 optional_defaults() (*HeinzingerConfig* class method), 104
 optional_defaults() (*HeinzingerSerialCommunicationConfig* class method), 109
 optional_defaults() (*ILS2TConfig* class method), 163
 optional_defaults() (*ILS2TModbusTcpCommunicationConfig* class method), 165
 optional_defaults() (*LaudaProRp245eConfig* class method), 123
 optional_defaults() (*LaudaProRp245eTcpCommunicationConfig* class method), 124
 optional_defaults() (*LJMCommunicationConfig* class method), 18
 optional_defaults() (*LuminoxConfig* class method), 168
 optional_defaults() (*LuminoxSerialCommunicationConfig* class method), 170
 optional_defaults() (*MBW973Config* class method), 127
 optional_defaults() (*MBW973SerialCommunicationConfig* class method), 128
 optional_defaults() (*ModbusTcpCommunication-*

- Config* class method), 20
- optional_defaults() (*NewportSMC100PPConfig* class method), 139
- optional_defaults() (*NewportSMC100PPSerialCommunicationConfig* class method), 143
- optional_defaults() (*OpcUaCommunicationConfig* class method), 23
- optional_defaults() (*PfeifferTPGConfig* class method), 148
- optional_defaults() (*PfeifferTPGSerialCommunicationConfig* class method), 149
- optional_defaults() (*PICubeConfiguration* class method), 47
- optional_defaults() (*PICubeOpcUaCommunicationConfig* class method), 48
- optional_defaults() (*PSI9000Config* class method), 88
- optional_defaults() (*PSI9000VisaCommunicationConfig* class method), 90
- optional_defaults() (*RTO1024Config* class method), 157
- optional_defaults() (*RTO1024VisaCommunicationConfig* class method), 158
- optional_defaults() (*SerialCommunicationConfig* class method), 26
- optional_defaults() (*T560CommunicationConfig* class method), 58
- optional_defaults() (*T560Config* class method), 62
- optional_defaults() (*TcpCommunicationConfig* class method), 29
- optional_defaults() (*TechnixConfig* class method), 68
- optional_defaults() (*TechnixSerialCommunicationConfig* class method), 64
- optional_defaults() (*TechnixTelnetCommunicationConfig* class method), 65
- optional_defaults() (*TelnetCommunicationConfig* class method), 31
- optional_defaults() (*VisaCommunicationConfig* class method), 34
- optional_defaults() (*VisaDeviceConfig* class method), 174
- OT (*NewportConfigCommands* attribute), 130
- out (*FuGProbusVDORegisters* property), 99
- OUTPUT (*FuGProbusIVCommands* attribute), 96
- OUTPUT (*GateMode* attribute), 57
- output (*Technix* property), 67
- output_off() (*FuGProbusIV* method), 96
- output_off() (*HeinzingerDI* method), 106
- output_on() (*HeinzingerDI* method), 106
- OUTPUT_POWER_EXCEEDED (*NewportSMC100PP.MotorErrors* attribute), 131
- output_status (*HeinzingerDI* property), 106
- OUTPUTONCMD (*FuGProbusVRegisterGroups* attribute), 99
- OUTPUTX0 (*FuGProbusVRegisterGroups* attribute), 99
- OUTPUTX1 (*FuGProbusVRegisterGroups* attribute), 100
- OUTPUTX2 (*FuGProbusVRegisterGroups* attribute), 100
- OUTPUTXCMD (*FuGProbusVRegisterGroups* attribute), 100
- outX0 (*FuG* property), 91
- outX1 (*FuG* property), 91
- outX2 (*FuG* property), 92
- outXCMD (*FuG* property), 92
- Overrange (*PfeifferTPG.SensorStatus* attribute), 144
- ## P
- PA (*Pressure* attribute), 176
- PARAM_MISSING_OR_INVALID (*NewportSMC100PPSerialCommunication.ControllerErrors* attribute), 140
- parity (*CryLasAttenuatorSerialCommunicationConfig* attribute), 78
- parity (*CryLasLaserSerialCommunicationConfig* attribute), 85
- parity (*FuGSerialCommunicationConfig* attribute), 102
- parity (*HeinzingerSerialCommunicationConfig* attribute), 109
- parity (*LuminoxSerialCommunicationConfig* attribute), 170
- parity (*MBW973SerialCommunicationConfig* attribute), 129
- parity (*NewportSMC100PPSerialCommunicationConfig* attribute), 143
- parity (*PfeifferTPGSerialCommunicationConfig* attribute), 149
- Parity (*SerialCommunicationConfig* attribute), 26
- parity (*SerialCommunicationConfig* attribute), 27
- parse_read_measurement_value() (*LuminoxMeasurementType* method), 169
- partial_pressure_o2 (*LuminoxMeasurementType* attribute), 169
- PASCAL (*Pressure* attribute), 176
- pause() (*LaudaProRp245e* method), 118
- pause_ramp() (*LaudaProRp245e* method), 118
- PBR (*PfeifferTPG.SensorTypes* attribute), 145
- PEAK_CURRENT_LIMIT (*NewportSMC100PP.MotorErrors* attribute), 131
- peak_output_current_limit (*NewportSMC100PPConfig* attribute), 139
- percent_o2 (*LuminoxMeasurementType* attribute), 169
- PERIOD (*MeasurementFunction* attribute), 53
- period (*T560* property), 61

period_aperture (*Fluke8845a* attribute), 50
 PfeifferTPG (class in *hvl_ccb.dev.pfeiffer_tpg*), 144
 PfeifferTPG.SensorStatus (class in *hvl_ccb.dev.pfeiffer_tpg*), 144
 PfeifferTPG.SensorTypes (class in *hvl_ccb.dev.pfeiffer_tpg*), 145
 PfeifferTPGConfig (class in *hvl_ccb.dev.pfeiffer_tpg*), 147
 PfeifferTPGConfig.Model (class in *hvl_ccb.dev.pfeiffer_tpg*), 147
 PfeifferTPGError, 148
 PfeifferTPGSerialCommunication (class in *hvl_ccb.dev.pfeiffer_tpg*), 148
 PfeifferTPGSerialCommunicationConfig (class in *hvl_ccb.dev.pfeiffer_tpg*), 148
 PICube (class in *hvl_ccb.dev.cube.picube*), 45
 PICubeConfiguration (class in *hvl_ccb.dev.cube.picube*), 46
 PICubeOpcUaCommunication (class in *hvl_ccb.dev.cube.picube*), 47
 PICubeOpcUaCommunicationConfig (class in *hvl_ccb.dev.cube.picube*), 47
 PICubeTestParameterError, 43
 PKR (*PfeifferTPG.SensorTypes* attribute), 145
 Polarity (class in *hvl_ccb.dev.cube.constants*), 43
 Polarity (class in *hvl_ccb.dev.highland_t560.base*), 57
 POLARITY (*FuGProbusIVCommands* attribute), 96
 polarity (*PICube* property), 46
 Poller (class in *hvl_ccb.dev.utils*), 171
 polling (*LuminosOutputMode* attribute), 169
 polling_delay_sec (*BaseCubeConfiguration* attribute), 40
 polling_interval (*MBW973Config* attribute), 127
 polling_interval_sec (*BaseCubeConfiguration* attribute), 40
 polling_interval_sec (*TechnixConfig* attribute), 68
 polling_period (*CryLasLaserConfig* attribute), 82
 polling_timeout (*CryLasLaserConfig* attribute), 82
 port (*Fluke8845aTelnetCommunicationConfig* attribute), 52
 port (*ModbusTcpCommunicationConfig* attribute), 20
 port (*OpcUaCommunicationConfig* attribute), 23
 port (*SerialCommunicationConfig* attribute), 27
 port (*T560CommunicationConfig* attribute), 59
 port (*TcpCommunicationConfig* attribute), 29
 port (*TechnixTelnetCommunicationConfig* attribute), 65
 port (*TelnetCommunicationConfig* attribute), 31
 port (*VisaCommunicationConfig* attribute), 34
 POS (*ILS2T.Ref16Jog* attribute), 160
 POS_END_OF_TURN (*NewportSMC100PP.MotorErrors* attribute), 131
 POS_FAST (*ILS2T.Ref16Jog* attribute), 160
 POSITION (*ILS2TRegAddr* attribute), 165
 POSITION_OUT_OF_LIMIT (New-
 portSMC100PPSerialCommunication.ControllerErrors
 attribute), 140
 POSITIVE (*FuGPolarities* attribute), 95
 POSITIVE (*Polarity* attribute), 44
 positive_software_limit (New-
 portSMC100PPConfig attribute), 139
 post_force_value() (New*portSMC100PPConfig*
 method), 139
 post_stop_pause_sec (*TechnixConfig* attribute), 68
 POUNDS_PER_SQUARE_INCH (*Pressure* attribute), 176
 POWER_INVERTER_220V (*PowerSetup* attribute), 44
 power_limit (*PSI9000Config* attribute), 88
 power_setup (*PICube* property), 46
 PowerSetup (class in *hvl_ccb.dev.cube.constants*), 44
 prepare_ultra_segmentation() (*RTO1024* method),
 153
 preserve_type() (in module
 hvl_ccb.utils.conversion.utils), 177
 Pressure (class in *hvl_ccb.utils.conversion.unit*), 176
 PSI (*Pressure* attribute), 177
 PSI9000 (class in *hvl_ccb.dev.ea_psi9000*), 85
 PSI9000Config (class in *hvl_ccb.dev.ea_psi9000*), 88
 PSI9000Error, 89
 PSI9000VisaCommunication (class in
 hvl_ccb.dev.ea_psi9000), 89
 PSI9000VisaCommunicationConfig (class in
 hvl_ccb.dev.ea_psi9000), 89
 PT100 (*LabJack.ThermocoupleType* attribute), 113
 PT1000 (*LabJack.ThermocoupleType* attribute), 113
 PT500 (*LabJack.ThermocoupleType* attribute), 113
 PTP (*ILS2T.Mode* attribute), 159
 pump_init (*LaudaProRp245eConfig* attribute), 123
 PUMP_LEVEL (*LaudaProRp245eCommand* attribute), 121

Q

QIL (*NewportConfigCommands* attribute), 130
 QUERY (*FuGProbusIVCommands* attribute), 96
 query() (*CryLasLaserSerialCommunication* method),
 83
 query() (*Fluke8845aTelnetCommunication* method), 52
 query() (*FuGSerialCommunication* method), 101
 query() (*NewportSMC100PPSerialCommunication*
 method), 140
 query() (*PfeifferTPGSerialCommunication* method),
 148
 query() (*SyncCommunicationProtocol* method), 15
 query() (*T560Communication* method), 58
 query() (*VisaCommunication* method), 33
 query_all() (*CryLasLaserSerialCommunication*
 method), 83
 query_command() (*Lau-
 daProRp245eTcpCommunication* method),
 124

- query_multiple() (*NewportSMC100PPSerialCommunication method*), 141
 query_polling() (*Luminox method*), 167
 query_status() (*Technix method*), 67
 QUEUE (*AutoInstallMode attribute*), 57
 QUICK_STOP (*SafetyStatus attribute*), 44
 QUICKSTOP (*ILS2T.State attribute*), 160
 quickstop() (*ILS2T method*), 162
 quit_error() (*BaseCube method*), 38
- ## R
- R (*LabJack.ThermocoupleType attribute*), 113
 raise() (*FuGErrorcodes method*), 95
 RAMP_ACC (*ILS2TRegAddr attribute*), 165
 RAMP_CONTINUE (*LaudaProRp245eCommand attribute*), 121
 RAMP_DECEL (*ILS2TRegAddr attribute*), 165
 RAMP_DELETE (*LaudaProRp245eCommand attribute*), 121
 RAMP_ITERATIONS (*LaudaProRp245eCommand attribute*), 121
 RAMP_N_MAX (*ILS2TRegAddr attribute*), 165
 RAMP_PAUSE (*LaudaProRp245eCommand attribute*), 121
 RAMP_SELECT (*LaudaProRp245eCommand attribute*), 121
 RAMP_SET (*LaudaProRp245eCommand attribute*), 121
 RAMP_START (*LaudaProRp245eCommand attribute*), 121
 RAMP_STOP (*LaudaProRp245eCommand attribute*), 121
 RAMP_TYPE (*ILS2TRegAddr attribute*), 165
 rampmode (*FuGProbusVSetRegisters property*), 100
 ramprate (*FuGProbusVSetRegisters property*), 100
 rampstate (*FuGProbusVSetRegisters property*), 100
 RAMPUPWARDS (*FuGRampModes attribute*), 101
 RangeEnum (*class in hvl_ccb.utils.enum*), 179
 RATEDCURRENT (*FuGReadbackChannels attribute*), 101
 RATEDVOLTAGE (*FuGReadbackChannels attribute*), 101
 read() (*AsyncCommunicationProtocol method*), 11
 read() (*BaseCube method*), 38
 read() (*CryLasLaserSerialCommunication method*), 84
 read() (*LaudaProRp245eTcpCommunication method*), 124
 read() (*MBW973 method*), 126
 read() (*OpcUaCommunication method*), 22
 read() (*Tcp method*), 28
 read_all() (*AsyncCommunicationProtocol method*), 12
 read_bytes() (*AsyncCommunicationProtocol method*), 12
 read_bytes() (*SerialCommunication method*), 25
 read_bytes() (*TelnetCommunication method*), 30
 read_float() (*MBW973 method*), 126
 read_holding_registers() (*ModbusTcpCommunication method*), 19
 read_input_registers() (*ModbusTcpCommunication method*), 19
 read_int() (*MBW973 method*), 126
 read_measurement() (*RTO1024 method*), 154
 read_measurements() (*MBW973 method*), 126
 read_name() (*LJMCommunication method*), 16
 read_nonempty() (*AsyncCommunicationProtocol method*), 12
 read_output_while_polling (*TechnixConfig attribute*), 68
 read_resistance() (*LabJack method*), 115
 read_single_bytes() (*SerialCommunication method*), 25
 read_streaming() (*Luminox method*), 167
 read_termination (*VisaCommunicationConfig attribute*), 35
 read_text() (*AsyncCommunicationProtocol method*), 12
 read_text() (*NewportSMC100PPSerialCommunication method*), 141
 read_text_nonempty() (*AsyncCommunicationProtocol method*), 12
 READ_TEXT_SKIP_PREFIXES (*CryLasLaserSerialCommunication attribute*), 83
 read_thermocouple() (*LabJack method*), 115
 readback_data (*FuGProbusVConfigRegisters property*), 97
 READBACKCHANNEL (*FuGProbusIVCommands attribute*), 96
 ready (*BaseCube property*), 38
 READY (*CryLasLaser.AnswersStatus attribute*), 78
 READY (*ILS2T.State attribute*), 160
 READY (*NewportStates attribute*), 144
 READY_ACTIVE (*CryLasLaser.LaserStatus attribute*), 79
 READY_FROM_DISABLE (*NewportSMC100PP.StateMessages attribute*), 131
 READY_FROM_HOMING (*NewportSMC100PP.StateMessages attribute*), 131
 READY_FROM_JOGGING (*NewportSMC100PP.StateMessages attribute*), 131
 READY_FROM_MOVING (*NewportSMC100PP.StateMessages attribute*), 131
 READY_INACTIVE (*CryLasLaser.LaserStatus attribute*), 79
 RED_OPERATE (*SafetyStatus attribute*), 44
 RED_READY (*SafetyStatus attribute*), 45
 reg_3 (*FuGProbusVDIRegisters property*), 98
 RegAddr (*ILS2T attribute*), 160
 RegDatatype (*ILS2T attribute*), 160
 register_pulse_time (*TechnixConfig attribute*), 68

RELATIVE_POSITION_MOTOR (*ILS2T.ActionsPtp* attribute), 159

RELATIVE_POSITION_TARGET (*ILS2T.ActionsPtp* attribute), 159

remote (*Technix* property), 67

remove_device() (*DeviceSequenceMixin* method), 73

required_keys() (*AsyncCommunicationProtocolConfig* class method), 14

required_keys() (*BaseCubeConfiguration* class method), 40

required_keys() (*BaseCubeOpcUaCommunicationConfig* class method), 41

required_keys() (*CryLasAttenuatorConfig* class method), 76

required_keys() (*CryLasAttenuatorSerialCommunicationConfig* class method), 78

required_keys() (*CryLasLaserConfig* class method), 82

required_keys() (*CryLasLaserSerialCommunicationConfig* class method), 85

required_keys() (*EmptyConfig* class method), 74, 182

required_keys() (*Fluke8845aConfig* class method), 51

required_keys() (*Fluke8845aTelnetCommunicationConfig* class method), 52

required_keys() (*FuGConfig* class method), 93

required_keys() (*FuGSerialCommunicationConfig* class method), 102

required_keys() (*HeinzingerConfig* class method), 104

required_keys() (*HeinzingerSerialCommunicationConfig* class method), 109

required_keys() (*ILS2TConfig* class method), 164

required_keys() (*ILS2TModbusTcpCommunicationConfig* class method), 165

required_keys() (*LaudaProRp245eConfig* class method), 123

required_keys() (*LaudaProRp245eTcpCommunicationConfig* class method), 124

required_keys() (*LJMCommunicationConfig* class method), 18

required_keys() (*LuminosConfig* class method), 168

required_keys() (*LuminosSerialCommunicationConfig* class method), 170

required_keys() (*MBW973Config* class method), 127

required_keys() (*MBW973SerialCommunicationConfig* class method), 129

required_keys() (*ModbusTcpCommunicationConfig* class method), 20

required_keys() (*NewportSMC100PPConfig* class method), 139

required_keys() (*NewportSMC100PPSerialCommunicationConfig* class method), 143

required_keys() (*OpcUaCommunicationConfig* class method), 23

required_keys() (*PfeifferTPGConfig* class method), 148

required_keys() (*PfeifferTPGSerialCommunicationConfig* class method), 149

required_keys() (*PICubeConfiguration* class method), 47

required_keys() (*PICubeOpcUaCommunicationConfig* class method), 48

required_keys() (*PSI9000Config* class method), 89

required_keys() (*PSI9000VisaCommunicationConfig* class method), 90

required_keys() (*RTO1024Config* class method), 158

required_keys() (*RTO1024VisaCommunicationConfig* class method), 158

required_keys() (*SerialCommunicationConfig* class method), 27

required_keys() (*T560CommunicationConfig* class method), 59

required_keys() (*T560Config* class method), 62

required_keys() (*TcpCommunicationConfig* class method), 29

required_keys() (*TechnixConfig* class method), 68

required_keys() (*TechnixSerialCommunicationConfig* class method), 64

required_keys() (*TechnixTelnetCommunicationConfig* class method), 65

required_keys() (*TelnetCommunicationConfig* class method), 31

required_keys() (*VisaCommunicationConfig* class method), 35

required_keys() (*VisaDeviceConfig* class method), 174

RESET (*FuGProbusIVCommands* attribute), 96

reset() (*Fluke8845a* method), 50

reset() (*FuGProbusIV* method), 96

reset() (*NewportSMC100PP* method), 135

reset() (*VisaDevice* method), 173

reset_error() (*ILS2T* method), 162

reset_interface() (*HeinzingerDI* method), 106

reset_ramp() (*LaudaProRp245e* method), 118

ResistanceRange (class in *hvl_ccb.dev.fluke884x.ranges*), 56

response_sleep_time (*CryLasAttenuatorConfig* attribute), 77

RMS_CURRENT_LIMIT (*NewportSMC100PP.MotorErrors* attribute), 131

rpm_max_init (*ILS2TConfig* attribute), 164

rs485_address (*NewportSMC100PPConfig* attribute), 139

RT01024 (class in *hvl_ccb.dev.rs_rto1024*), 150

RT01024.TriggerModes (class in *hvl_ccb.dev.rs_rto1024*), 151

- RTO1024Config (*class in hvl_ccb.dev.rs_rto1024*), 157
 RTO1024Error, 158
 RTO1024VisaCommunication (*class in hvl_ccb.dev.rs_rto1024*), 158
 RTO1024VisaCommunicationConfig (*class in hvl_ccb.dev.rs_rto1024*), 158
 run() (*ExperimentManager method*), 184
 run() (*LaudaProRp245e method*), 119
 run_continuous_acquisition() (*RTO1024 method*), 154
 run_single_acquisition() (*RTO1024 method*), 154
 RUNNING (*ExperimentStatus attribute*), 185
- ## S
- S (*LabJack.ThermocoupleType attribute*), 113
 SA (*NewportConfigCommands attribute*), 130
 SafetyStatus (*class in hvl_ccb.dev.cube.constants*), 44
 save_configuration() (*RTO1024 method*), 154
 save_device_configuration() (*T560 method*), 61
 save_waveform_history() (*RTO1024 method*), 154
 SCALE (*ILS2TRegAddr attribute*), 165
 ScalingFactorValueError, 166
 screw_scaling (*NewportSMC100PPConfig attribute*), 139
 send_command() (*NewportSMC100PPSerialCommunication method*), 141
 send_command() (*PfeifferTPGSerialCommunication method*), 148
 send_hello() (*Client method*), 21
 send_pulses() (*LabJack method*), 115
 send_stop() (*NewportSMC100PPSerialCommunication method*), 142
 Sensor (*class in hvl_ccb.utils.conversion.sensor*), 176
 Sensor_error (*PfeifferTPG.SensorStatus attribute*), 144
 Sensor_off (*PfeifferTPG.SensorStatus attribute*), 144
 sensor_status (*LuminosMeasurementType attribute*), 169
 SERIAL (*LaudaProRp245eConfig.ExtControlModeEnum attribute*), 122
 serial_number (*LuminosMeasurementType attribute*), 169
 SerialCommunication (*class in hvl_ccb.comm.serial*), 24
 SerialCommunicationBytesize (*class in hvl_ccb.comm.serial*), 25
 SerialCommunicationConfig (*class in hvl_ccb.comm.serial*), 26
 SerialCommunicationIOError, 27
 SerialCommunicationParity (*class in hvl_ccb.comm.serial*), 27
 SerialCommunicationStopbits (*class in hvl_ccb.comm.serial*), 27
 Server (*class in hvl_ccb.comm.opc*), 24
 set_acceleration() (*NewportSMC100PP method*), 135
 set_acquire_length() (*RTO1024 method*), 154
 set_ain_differential() (*LabJack method*), 116
 set_ain_range() (*LabJack method*), 116
 set_ain_resistance() (*LabJack method*), 116
 set_ain_resolution() (*LabJack method*), 116
 set_ain_thermocouple() (*LabJack method*), 116
 set_analog_output() (*LabJack method*), 117
 set_attenuation() (*CryLasAttenuator method*), 75
 set_channel_offset() (*RTO1024 method*), 154
 set_channel_position() (*RTO1024 method*), 155
 set_channel_range() (*RTO1024 method*), 155
 set_channel_scale() (*RTO1024 method*), 155
 set_channel_state() (*RTO1024 method*), 156
 set_clock() (*LabJack method*), 117
 set_control_mode() (*LaudaProRp245e method*), 119
 set_current() (*HeinzingerDI method*), 106
 set_current() (*HeinzingerPNC method*), 108
 set_digital_output() (*LabJack method*), 117
 set_external_temp() (*LaudaProRp245e method*), 119
 set_full_scale_mbar() (*PfeifferTPG method*), 146
 set_full_scale_unitless() (*PfeifferTPG method*), 146
 set_init_attenuation() (*CryLasAttenuator method*), 75
 set_init_shutter_status() (*CryLasLaser method*), 80
 set_jog_speed() (*ILS2T method*), 162
 set_lower_limits() (*PSI9000 method*), 87
 set_max_acceleration() (*ILS2T method*), 162
 set_max_deceleration() (*ILS2T method*), 162
 set_max_rpm() (*ILS2T method*), 162
 set_measuring_options() (*MBW973 method*), 126
 set_message_board() (*BaseCube method*), 38
 set_motor_configuration() (*NewportSMC100PP method*), 136
 set_negative_software_limit() (*NewportSMC100PP method*), 136
 set_number_of_recordings() (*HeinzingerDI method*), 106
 set_output() (*PSI9000 method*), 87
 set_positive_software_limit() (*NewportSMC100PP method*), 136
 set_pulse_energy() (*CryLasLaser method*), 80
 set_pump_level() (*LaudaProRp245e method*), 119
 set_ramp_iterations() (*LaudaProRp245e method*), 119
 set_ramp_program() (*LaudaProRp245e method*), 119
 set_ramp_segment() (*LaudaProRp245e method*), 119
 set_ramp_type() (*ILS2T method*), 162
 set_reference_point() (*RTO1024 method*), 156
 set_register() (*FuGProbusV method*), 97
 set_repetition_rate() (*CryLasLaser method*), 80

set_repetitions() (*RTO1024 method*), 156
set_status_board() (*BaseCube method*), 38
set_system_lock() (*PSI9000 method*), 87
set_temp_set_point() (*LaudaProRp245e method*), 120
set_transmission() (*CryLasAttenuator method*), 76
set_trigger_level() (*RTO1024 method*), 156
set_trigger_mode() (*RTO1024 method*), 157
set_trigger_source() (*RTO1024 method*), 157
set_upper_limits() (*PSI9000 method*), 87
set_voltage() (*HeinzingerDI method*), 106
set_voltage() (*HeinzingerPNC method*), 108
set_voltage_current() (*PSI9000 method*), 87
SetAttr (*class in hvl_ccb.utils.conversion.utils*), 177
SETCURRENT (*FuGProbusVRegisterGroups attribute*), 100
setvalue (*FuGProbusVSetRegisters property*), 100
SETVOLTAGE (*FuGProbusVRegisterGroups attribute*), 100
SEVEN_HUNDRED_FIFTY_VOLT (*ACVoltageRange attribute*), 55
SEVENBITS (*SerialCommunicationBytesize attribute*), 26
SHORT_CIRCUIT (*NewportSMC100PP.MotorErrors attribute*), 131
shunt (*LEM4000S attribute*), 175
SHUTDOWN_CURRENT_LIMIT (*PSI9000 attribute*), 86
SHUTDOWN_VOLTAGE_LIMIT (*PSI9000 attribute*), 86
ShutterStatus (*CryLasLaser attribute*), 79
ShutterStatus (*CryLasLaserConfig attribute*), 82
SingleCommDevice (*class in hvl_ccb.dev.base*), 74
SIXBITS (*SerialCommunicationBytesize attribute*), 26
SIXTEEN (*HeinzingerConfig.RecordingsEnum attribute*), 104
SL (*NewportConfigCommands attribute*), 130
SLOW_FILTER (*FilterRange attribute*), 56
SN (*FuGReadbackChannels attribute*), 101
SOFTWARE_INTERNAL_SIXTY (*CryLasLaser.RepetitionRates attribute*), 79
SOFTWARE_INTERNAL_TEN (*CryLasLaser.RepetitionRates attribute*), 79
SOFTWARE_INTERNAL_TWENTY (*CryLasLaser.RepetitionRates attribute*), 79
software_revision (*LuminosMeasurementType attribute*), 169
SPACE (*SerialCommunicationParity attribute*), 27
SPECIALRAMPUPWARDS (*FuGRampModes attribute*), 101
spoll() (*VisaCommunication method*), 33
spoll_handler() (*VisaDevice method*), 173
SR (*NewportConfigCommands attribute*), 130
srq_mask (*FuGProbusVConfigRegisters property*), 97
srq_status (*FuGProbusVConfigRegisters property*), 97
stage_configuration (*NewportSMC100PPConfig attribute*), 139
START (*LaudaProRp245eCommand attribute*), 121
start() (*BaseCube method*), 38
start() (*CryLasAttenuator method*), 76
start() (*CryLasLaser method*), 81
start() (*Device method*), 72
start() (*DeviceSequenceMixin method*), 73
start() (*ExperimentManager method*), 184
start() (*Fluke8845a method*), 50
start() (*FuG method*), 92
start() (*FuGProbusIV method*), 96
start() (*HeinzingerDI method*), 107
start() (*HeinzingerPNC method*), 108
start() (*ILS2T method*), 162
start() (*LabJack method*), 117
start() (*LaudaProRp245e method*), 120
start() (*Luminos method*), 167
start() (*MBW973 method*), 126
start() (*NewportSMC100PP method*), 136
start() (*PfeifferTPG method*), 147
start() (*PSI9000 method*), 88
start() (*RTO1024 method*), 157
start() (*SingleCommDevice method*), 75
start() (*Technix method*), 67
start() (*VisaDevice method*), 173
start_control() (*MBW973 method*), 126
start_polling() (*Poller method*), 171
start_ramp() (*LaudaProRp245e method*), 120
STARTING (*ExperimentStatus attribute*), 185
States (*NewportSMC100PP attribute*), 131
status (*BaseCube property*), 38
status (*ExperimentManager property*), 184
status (*FuGProbusVConfigRegisters property*), 98
status (*FuGProbusVDOResources property*), 99
status (*Technix property*), 67
STATUSBYTE (*FuGReadbackChannels attribute*), 101
STOP (*LaudaProRp245eCommand attribute*), 121
stop() (*BaseCube method*), 39
stop() (*CryLasLaser method*), 81
stop() (*Device method*), 72
stop() (*DeviceSequenceMixin method*), 73
stop() (*ExperimentManager method*), 184
stop() (*Fluke8845a method*), 50
stop() (*FuGProbusIV method*), 96
stop() (*HeinzingerDI method*), 107
stop() (*ILS2T method*), 162
stop() (*LabJack method*), 117
stop() (*LaudaProRp245e method*), 120
stop() (*Luminos method*), 167
stop() (*MBW973 method*), 126
stop() (*NewportSMC100PP method*), 137
stop() (*PfeifferTPG method*), 147
stop() (*PSI9000 method*), 88
stop() (*RTO1024 method*), 157
stop() (*SingleCommDevice method*), 75
stop() (*Technix method*), 67

stop() (*VisaDevice* method), 173
 stop_acquisition() (*RTO1024* method), 157
 stop_motion() (*NewportSMC100PP* method), 137
 stop_polling() (*Poller* method), 171
 stop_ramp() (*LaudaProRp245e* method), 120
 STOP_SAFETY_STATUSES (in *hvl_ccb.dev.cube.constants*), 44
 stopbits (*CryLasAttenuatorSerialCommunicationConfig* attribute), 78
 stopbits (*CryLasLaserSerialCommunicationConfig* attribute), 85
 stopbits (*FuGSerialCommunicationConfig* attribute), 102
 stopbits (*HeinzingerSerialCommunicationConfig* attribute), 109
 stopbits (*LuminosSerialCommunicationConfig* attribute), 171
 stopbits (*MBW973SerialCommunicationConfig* attribute), 129
 stopbits (*NewportSMC100PPSerialCommunicationConfig* attribute), 143
 stopbits (*PfeifferTPGSerialCommunicationConfig* attribute), 149
 Stopbits (*SerialCommunicationConfig* attribute), 26
 stopbits (*SerialCommunicationConfig* attribute), 27
 streaming (*LuminosOutputMode* attribute), 169
 StrEnumBase (class in *hvl_ccb.utils.enum*), 179
 sub_handler (*BaseCubeOpcUaCommunicationConfig* attribute), 41
 sub_handler (*OpcUaCommunicationConfig* attribute), 23
 SyncCommunicationProtocol (class in *hvl_ccb.comm.base*), 15
 SyncCommunicationProtocolConfig (class in *hvl_ccb.comm.base*), 15

T

T (*LabJack.ThermocoupleType* attribute), 113
 t13_socket_1 (*BaseCube* attribute), 39
 t13_socket_2 (*BaseCube* attribute), 39
 t13_socket_3 (*BaseCube* attribute), 39
 T1MS (*FuGMonitorModes* attribute), 95
 T200MS (*FuGMonitorModes* attribute), 95
 T20MS (*FuGMonitorModes* attribute), 95
 T256US (*FuGMonitorModes* attribute), 95
 T4 (*LabJack.DeviceType* attribute), 112
 T4 (*LJMCommunicationConfig.DeviceType* attribute), 17
 T40MS (*FuGMonitorModes* attribute), 95
 T4MS (*FuGMonitorModes* attribute), 95
 T560 (class in *hvl_ccb.dev.highland_t560.device*), 60
 T560Communication (class in *hvl_ccb.dev.highland_t560.base*), 58
 T560CommunicationConfig (class in *hvl_ccb.dev.highland_t560.base*), 58
 T560Config (class in *hvl_ccb.dev.highland_t560.device*), 61
 T560Error, 59
 T7 (*LabJack.DeviceType* attribute), 112
 T7 (*LJMCommunicationConfig.DeviceType* attribute), 17
 T7_PRO (*LabJack.DeviceType* attribute), 112
 T7_PRO (*LJMCommunicationConfig.DeviceType* attribute), 17
 T800MS (*FuGMonitorModes* attribute), 95
 T80MS (*FuGMonitorModes* attribute), 95
 target_pulse_energy (*CryLasLaser* property), 81
 Tcp (class in *hvl_ccb.comm.tcp*), 28
 TCP (*LJMCommunicationConfig.ConnectionType* attribute), 17
 TcpCommunicationConfig (class in *hvl_ccb.comm.tcp*), 28
 TCPIP_INSTR (*VisaCommunicationConfig.InterfaceType* attribute), 33
 TCPIP_SOCKET (*VisaCommunicationConfig.InterfaceType* attribute), 33
 TEC1 (*CryLasLaser.AnswersStatus* attribute), 78
 TEC2 (*CryLasLaser.AnswersStatus* attribute), 78
 Technix (class in *hvl_ccb.dev.technix.device*), 66
 TechnixConfig (class in *hvl_ccb.dev.technix.device*), 67
 TechnixError, 63
 TechnixFaultError, 63
 TechnixSerialCommunication (class in *hvl_ccb.dev.technix.base*), 63
 TechnixSerialCommunicationConfig (class in *hvl_ccb.dev.technix.base*), 63
 TechnixTelnetCommunication (class in *hvl_ccb.dev.technix.base*), 64
 TechnixTelnetCommunicationConfig (class in *hvl_ccb.dev.technix.base*), 64
 TelnetCommunication (class in *hvl_ccb.comm.telnet*), 30
 TelnetCommunicationConfig (class in *hvl_ccb.comm.telnet*), 31
 TelnetError, 32
 TEMP (*ILS2TRegAddr* attribute), 165
 TEMP_SET_POINT (*LaudaProRp245eCommand* attribute), 121
 temp_set_point_init (*LaudaProRp245eConfig* attribute), 123
 Temperature (class in *hvl_ccb.utils.conversion.unit*), 177
 temperature_sensor (*LuminosMeasurementType* attribute), 169
 temperature_unit (*LMT70A* attribute), 176
 TEN (*LabJack.AInRange* attribute), 111
 TEN (*LabJack.CalMicroAmpere* attribute), 111
 TEN_AMPERE (*ACCCurrentRange* attribute), 54
 TEN_AMPERE (*DCCurrentRange* attribute), 55
 TEN_MHZ (*LabJack.ClockFrequency* attribute), 112

- TEN_MILLI_AMPERE (*ACCurrentRange* attribute), 54
 - TEN_MILLI_AMPERE (*DCCurrentRange* attribute), 55
 - TEN_MILLI_SECOND (*ApertureRange* attribute), 55
 - TEN_MILLION_OHM (*ResistanceRange* attribute), 56
 - TEN_THOUSAND_OHM (*ResistanceRange* attribute), 56
 - TEN_VOLT (*ACVoltageRange* attribute), 55
 - TEN_VOLT (*DCVoltageRange* attribute), 56
 - terminator (*AsyncCommunicationProtocolConfig* attribute), 14
 - terminator (*CryLasAttenuatorSerialCommunicationConfig* attribute), 78
 - terminator (*CryLasLaserSerialCommunicationConfig* attribute), 85
 - terminator (*Fluke8845aTelnnetCommunicationConfig* attribute), 52
 - TERMINATOR (*FuGProbusIVCommands* attribute), 96
 - terminator (*FuGProbusVConfigRegisters* property), 98
 - terminator (*FuGSerialCommunicationConfig* attribute), 103
 - terminator (*HeinzingerSerialCommunicationConfig* attribute), 110
 - terminator (*LaudaProRp245eTcpCommunicationConfig* attribute), 125
 - terminator (*LuminosSerialCommunicationConfig* attribute), 171
 - terminator (*MBW973SerialCommunicationConfig* attribute), 129
 - terminator (*NewportSMC100PPSerialCommunicationConfig* attribute), 143
 - terminator (*PfeifferTPGSerialCommunicationConfig* attribute), 150
 - terminator (*T560CommunicationConfig* attribute), 59
 - terminator_str() (*SerialCommunicationConfig* method), 27
 - THIRTY_TWO_BIT (*LabJack.BitLimit* attribute), 111
 - THREE_AMPERE (*ACCurrentRange* attribute), 54
 - THREE_AMPERE (*DCCurrentRange* attribute), 55
 - timeout (*CryLasAttenuatorSerialCommunicationConfig* attribute), 78
 - timeout (*CryLasLaserSerialCommunicationConfig* attribute), 85
 - timeout (*FuGSerialCommunicationConfig* attribute), 103
 - timeout (*HeinzingerSerialCommunicationConfig* attribute), 110
 - timeout (*LuminosSerialCommunicationConfig* attribute), 171
 - timeout (*MBW973SerialCommunicationConfig* attribute), 129
 - timeout (*NewportSMC100PPSerialCommunicationConfig* attribute), 143
 - timeout (*PfeifferTPGSerialCommunicationConfig* attribute), 150
 - timeout (*SerialCommunicationConfig* attribute), 27
 - timeout (*TelnnetCommunicationConfig* attribute), 31
 - timeout (*VisaCommunicationConfig* attribute), 35
 - timeout_interval (*BaseCubeConfiguration* attribute), 40
 - timeout_status_change (*BaseCubeConfiguration* attribute), 40
 - timeout_test_parameters (*PICubeConfiguration* attribute), 47
 - TORR (*Pressure* attribute), 177
 - TPG25xA (*PfeifferTPGConfig.Model* attribute), 147
 - TPGx6x (*PfeifferTPGConfig.Model* attribute), 147
 - TPR (*PfeifferTPG.SensorTypes* attribute), 145
 - transmission (*CryLasAttenuator* property), 76
 - trigger() (*Fluke8845a* method), 50
 - trigger_delay (*Fluke8845a* property), 50
 - trigger_level (*T560* property), 61
 - trigger_mode (*T560* property), 61
 - trigger_source (*Fluke8845a* property), 50
 - TriggerMode (class in *hvl_ccb.dev.highland_t560.base*), 59
 - TriggerSource (class in *hvl_ccb.dev.fluke884x.constants*), 54
 - TWELVE_HUNDRED_FIFTY_KHZ (*LabJack.ClockFrequency* attribute), 112
 - TWENTY_FIVE_HUNDRED_KHZ (*LabJack.ClockFrequency* attribute), 112
 - TWENTY_MHZ (*LabJack.ClockFrequency* attribute), 112
 - TYPE (*HeinzingerConfig.RecordingsEnum* attribute), 104
 - TWO (*SerialCommunicationStopbits* attribute), 28
 - TWO_HUNDRED (*LabJack.CalMicroAmpere* attribute), 111
 - TWO_WIRE_RESISTANCE (*MeasurementFunction* attribute), 53
 - two_wire_resistance_range (*Fluke8845a* attribute), 51
- ## U
- Underrange (*PfeifferTPG.SensorStatus* attribute), 145
 - Unit (class in *hvl_ccb.utils.conversion.unit*), 177
 - unit (*ILS2TModbusTcpCommunicationConfig* attribute), 165
 - unit (*ModbusTcpCommunicationConfig* attribute), 20
 - unit (*PfeifferTPG* property), 147
 - unit() (*RangeEnum* class method), 179
 - unit_current (*HeinzingerPNC* property), 108
 - unit_voltage (*HeinzingerPNC* property), 108
 - UNKNOWN (*HeinzingerDI.OutputStatus* attribute), 105
 - UNKNOWN (*HeinzingerPNC.UnitCurrent* attribute), 107
 - UNKNOWN (*HeinzingerPNC.UnitVoltage* attribute), 107
 - UNREADY_INACTIVE (*CryLasLaser.LaserStatus* attribute), 79
 - update_laser_status() (*CryLasLaser* method), 81
 - update_parameter (*OpcUaCommunicationConfig* attribute), 23

update_repetition_rate() (*CryLasLaser* method), 81
 update_shutter_status() (*CryLasLaser* method), 81
 update_target_pulse_energy() (*CryLasLaser* method), 81
 UpdateEspStageInfo (*NewportSMC100PPConfig.EspStageConfig* attribute), 138
 UPPER_TEMP (*LaudaProRp245eCommand* attribute), 121
 upper_temp (*LaudaProRp245eConfig* attribute), 123
 USB (*LaudaProRp245eConfig.ExtControlModeEnum* attribute), 122
 USB (*LJMCommunicationConfig.ConnectionType* attribute), 17
 use_external_clock() (*T560* method), 61
 user_position_offset (*NewportSMC100PPConfig* attribute), 139
 user_steps() (*ILS2T* method), 163

V

V (*HeinzingerPNC.UnitVoltage* attribute), 107
 VA (*NewportConfigCommands* attribute), 130
 validate_and_resolve_host() (in module *hvl_ccb.utils.validation*), 180
 validate_bool() (in module *hvl_ccb.utils.validation*), 180
 validate_number() (in module *hvl_ccb.utils.validation*), 180
 validate_pump_level() (*LaudaProRp245e* method), 120
 validate_tcp_port() (in module *hvl_ccb.utils.validation*), 181
 value (*FuGProbusVMonitorRegisters* property), 99
 value (*LabJack.AInRange* property), 111
 value_raw (*FuGProbusVMonitorRegisters* property), 99
 ValueEnum (class in *hvl_ccb.utils.enum*), 179
 VB (*NewportConfigCommands* attribute), 130
 velocity (*NewportSMC100PPConfig* attribute), 139
 visa_backend (*VisaCommunicationConfig* attribute), 35
 VisaCommunication (class in *hvl_ccb.comm.visa*), 32
 VisaCommunicationConfig (class in *hvl_ccb.comm.visa*), 33
 VisaCommunicationConfig.InterfaceType (class in *hvl_ccb.comm.visa*), 33
 VisaCommunicationError, 35
 VisaDevice (class in *hvl_ccb.dev.visa*), 172
 VisaDeviceConfig (class in *hvl_ccb.dev.visa*), 173
 VOLT (*ILS2TRegAddr* attribute), 165
 voltage (*FuG* property), 92
 VOLTAGE (*FuGProbusIVCommands* attribute), 97
 VOLTAGE (*FuGReadbackChannels* attribute), 101
 voltage (*Technix* property), 67
 VOLTAGE_AC (*MeasurementFunction* attribute), 53
 voltage_actual (*PICube* property), 46

VOLTAGE_DC (*MeasurementFunction* attribute), 53
 voltage_filter (*Fluke8845a* attribute), 51
 voltage_lower_limit (*PSI9000Config* attribute), 89
 voltage_max (*PICube* property), 46
 voltage_monitor (*FuG* property), 92
 voltage_primary (*PICube* property), 46
 voltage_regulation (*Technix* property), 67
 voltage_upper_limit (*PSI9000Config* attribute), 89

W

WAIT_AFTER_WRITE (*VisaCommunication* attribute), 32
 wait_for_polling_result() (*Poller* method), 171
 wait_operation_complete() (*VisaDevice* method), 173
 wait_sec_initialisation (*PSI9000Config* attribute), 89
 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 (*LaudaProRp245eTcpCommunicationConfig* attribute), 125
 wait_sec_read_text_nonempty (*AsyncCommunicationProtocolConfig* attribute), 14
 wait_sec_read_text_nonempty (*FuGSerialCommunicationConfig* attribute), 103
 wait_sec_read_text_nonempty (*HeinzingerSerialCommunicationConfig* attribute), 110
 wait_sec_settings_effect (*PSI9000Config* attribute), 89
 wait_sec_stop_commands (*FuGConfig* attribute), 93
 wait_sec_stop_commands (*HeinzingerConfig* attribute), 104
 wait_sec_system_lock (*PSI9000Config* attribute), 89
 wait_sec_trials_activate (*LuminoxConfig* attribute), 168
 wait_timeout_retry_sec (*OpcUaCommunicationConfig* attribute), 23
 wait_until_motor_initialized() (*NewportSMC100PP* method), 137
 wait_until_ready() (*CryLasLaser* method), 81
 WIFI (*LJMCommunicationConfig.ConnectionType* attribute), 17
 write() (*AsyncCommunicationProtocol* method), 13
 write() (*BaseCube* method), 39
 write() (*MBW973* method), 127
 write() (*OpcUaCommunication* method), 22

`write()` (*Tcp method*), 28
`write()` (*VisaCommunication method*), 33
`write_absolute_position()` (*ILS2T method*), 163
`write_bytes()` (*AsyncCommunicationProtocol method*), 13
`write_bytes()` (*SerialCommunication method*), 25
`write_bytes()` (*TelnetCommunication method*), 30
`write_command()` (*Lau-daProRp245eTcpCommunication method*), 124
`write_name()` (*LJMCommunication method*), 17
`write_names()` (*LJMCommunication method*), 17
`write_registers()` (*ModbusTcpCommunication method*), 20
`write_relative_step()` (*ILS2T method*), 163
`write_termination` (*VisaCommunicationConfig attribute*), 35
`write_text()` (*AsyncCommunicationProtocol method*), 13
`WRONG_ESP_STAGE` (*NewportSMC100PP.MotorErrors attribute*), 131

X

`x_stat` (*FuGProbusVDIRegisters property*), 98
`XOUTPUTS` (*FuGProbusIVCommands attribute*), 97

Y

`YES` (*FuGDigitalVal attribute*), 93

Z

`ZX` (*NewportConfigCommands attribute*), 130